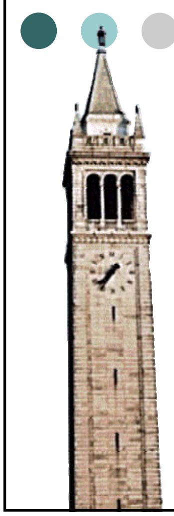
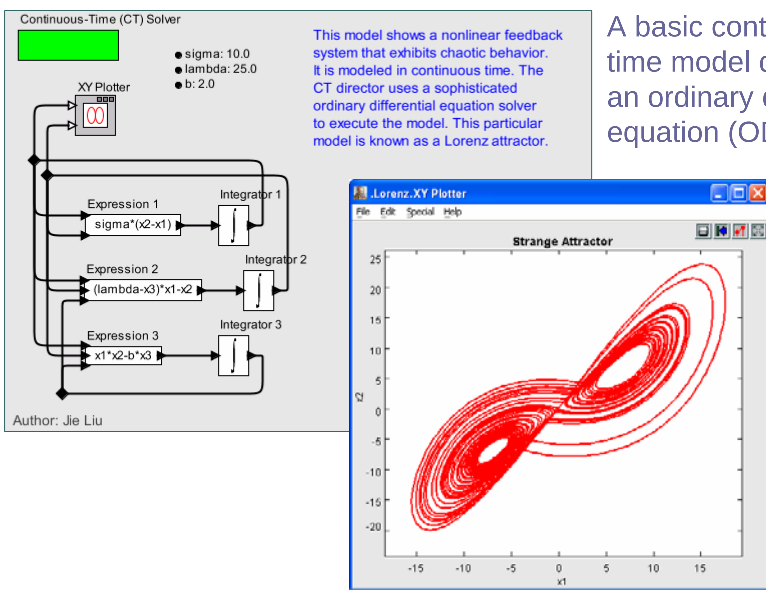


Introduction to Embedded Systems

Edward A. Lee & Sanjit Seshia
UC Berkeley
EECS 124
Spring 2008

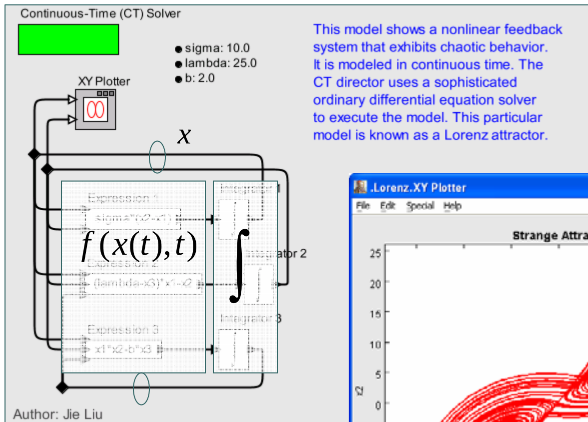
Lecture 12: Simulation Strategies for Continuous and Hybrid Models

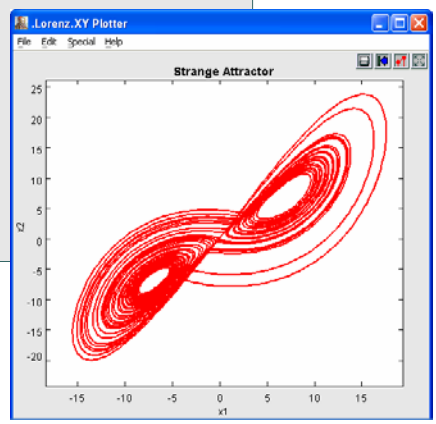
Basic Continuous-Time Modeling



A basic continuous-time model describes an ordinary differential equation (ODE).

Basic Continuous-Time Modeling





A basic continuous-time model describes an ordinary differential equation (ODE).

$$\dot{x}(t) = f(x(t), t)$$

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)\, d\tau$$

Basic Continuous-Time Modeling

The state trajectory is modeled as a vector function of time,

$$x : T \to R^n \qquad T = [t_0, \infty) \subset R$$

$$f(x(t),t) \xrightarrow{\dot{x}} x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)\, d\tau \xrightarrow{x}$$

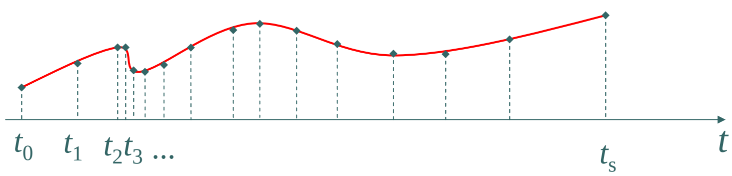
$$\dot{x}(t) = f(x(t), t)$$

$$f : R^m \times T \to R^m$$

ODE Solvers

Numerical solution approximates the state trajectory of the ODE by estimating its value at discrete time points:

$$\{t_0, t_1, \ldots\} \subset T$$



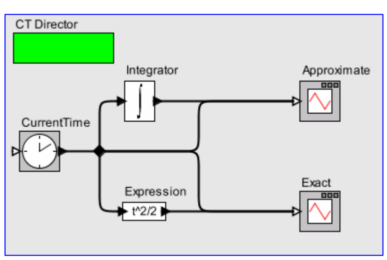
Reasonable choices for these points depend on the function *f*.

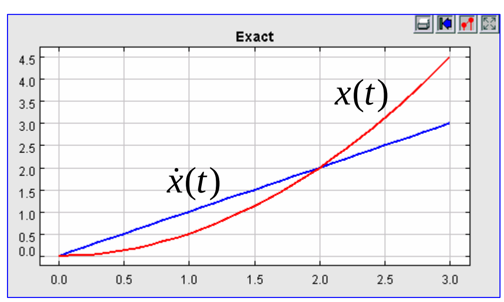
Using such solvers, signals are discrete-event signals.

Simple Example

This simple example integrates a ramp. In this case, it is easy to find a closed form solution,

$$\dot{x}(t) = t \;\Rightarrow\; x(t) = t^2/2$$

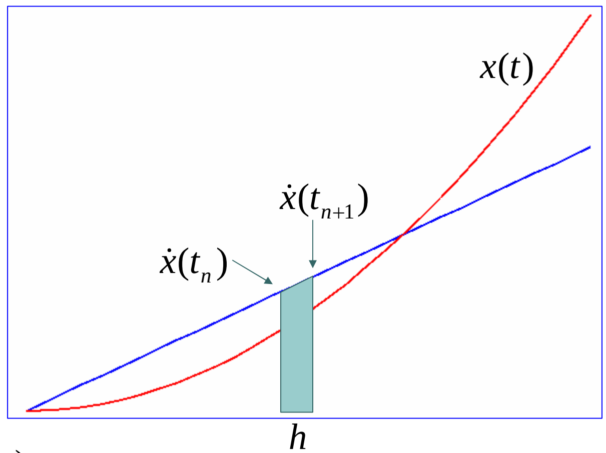




Trapezoidal Method: An example of an "implicit" method

Classical method estimates the area under the curve by calculating the area of trapezoids.

However, this method requires knowing $\dot{x}(t_{n+1})$, which in a feedback system isn't known until $x(t_{n+1})$ is known.

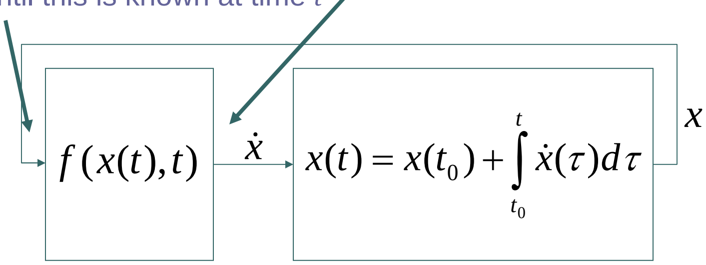


$$x(t_{n+1}) = x(t_n) + h(\dot{x}(t_n) + \dot{x}(t_{n+1}))/2$$

Implicit Methods are Challenging with Feedback

Can't know this at any time t

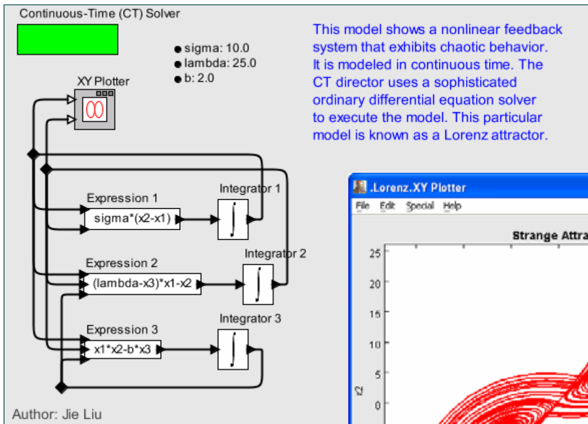
Until this is known at time t

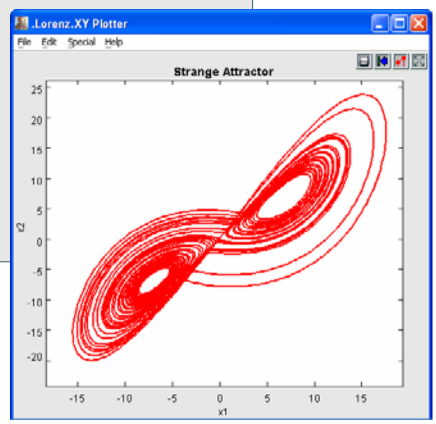


$$\dot{x}(t) = f(x(t),t)$$

$$f : R^m \times T \to R^m$$

Implicit Methods are Challenging with Feedback





We have a "causality loop."

One possible approach is to iterate to a solution. Convergence and uniqueness are not always guaranteed.

Forward Euler Solver: An example of an "explicit method"

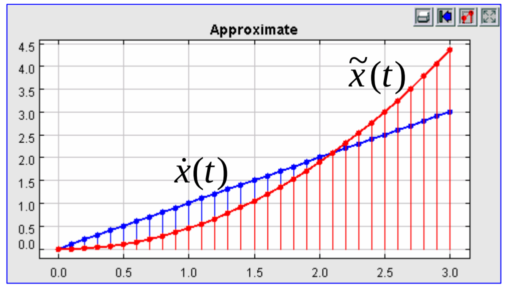
Given $x(t_n)$ and a time increment h, calculate:

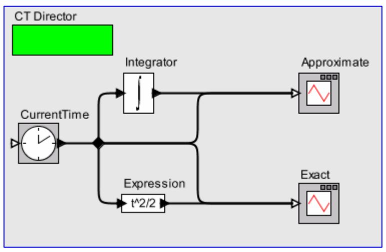
$$t_{n+1} = t_n + h$$
$$x(t_{n+1}) = x(t_n) + h\, f(x(t_n), t_n)$$

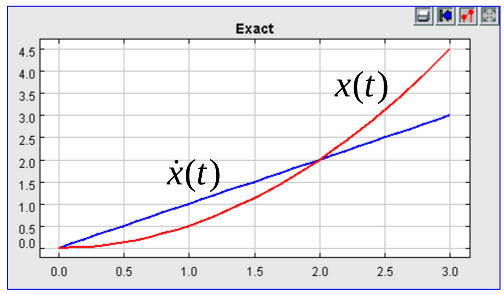
This method can be used in feedback systems. The solution is unique.

Forward Euler on Simple Example

In this case, we have used a fixed step size $h = 0.1$. The result is close, but diverges over time.







"Stiff" systems require small step sizes

Force due to spring extension:

$$F_1(t) = k(p - x(t))$$

Force due to viscous damping:

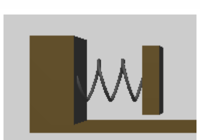
$$F_2(t) = -c\dot{x}(t)$$

Newton's second law:

$$F_1(t) + F_2(t) = M\ddot{x}(t)$$

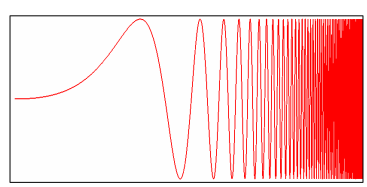
or

$$M\ddot{x}(t) + c\dot{x}(t) + kx(t) = kp.$$



For spring-mass damper, large stiffness constant *k* makes the system "stiff."

Variable step-size methods will dynamically modify the step size *h* in response to estimates of the integration error. Even these, however, run into trouble when stiffness varies over time. Extreme case of increasing stiffness results in Zeno behavior:



Runge-Kutta 2-3 Solver (RK2-3): Improving on Forward Euler

Given $x(t_n)$ and a time increment h, calculate

$$K_0 = f(x(t_n), t_n) \quad \dot{x}(t_n)$$

$$K_1 = f(x(t_n) + 0.5hK_0, t_n + 0.5h) \quad \text{estimate of } \dot{x}(t_n + 0.5h)$$

$$K_2 = f(x(t_n) + 0.75hK_1, t_n + 0.75h) \quad \text{estimate of } \dot{x}(t_n + 0.75h)$$

then let

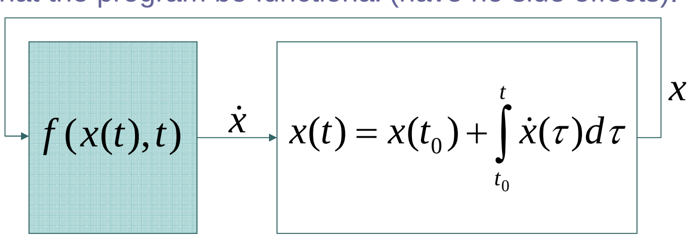
$$t_{n+1} = t_n + h$$

$$x(t_{n+1}) = x(t_n) + (2/9)hK_0 + (3/9)hK_1 + (4/9)hK_2$$

Note that this requires three evaluations of f at three different times with three different inputs.

Operational Requirements

In a software system, the blue box below can be specified by a program that, given $x(t)$ and t calculates $f(x(t), t)$. But this requires that the program be functional (have no side effects).



$$\dot{x}(t) = f(x(t), t)$$

$$f : R^m \times T \to R^m$$

For variable-step size RK2-3, have to be able to evaluate f at t_n , $t_n + 0.5h$, and $t_n + 0.75h$ without committing to the step size h . (Evaluation must have no side effects).

Adjusting the Time Steps

For time step given by $t_{n+1} = t_n + h$, let

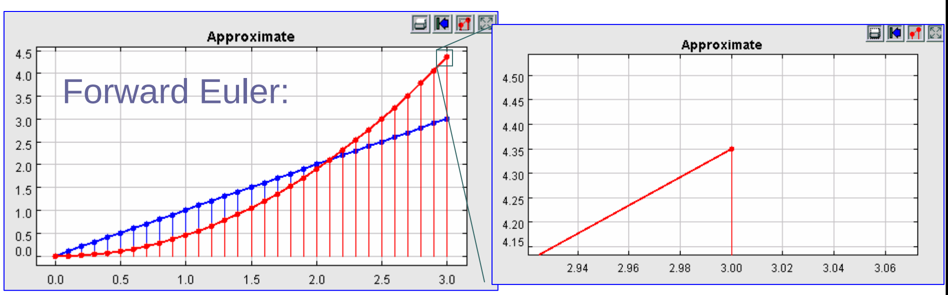
$$K_3 = f(x(t_{n+1}), t_{n+1})$$
$$\varepsilon = h((-5/72)K_0 + (1/12)K_1 + (1/9)K_2 + (-1/8)K_3)$$

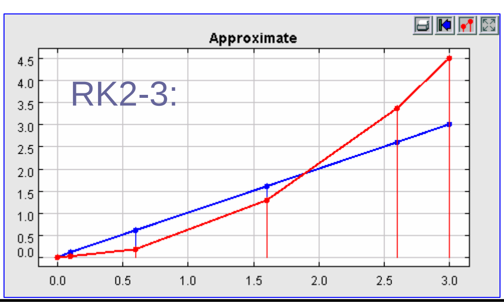
If ε is less than the “error tolerance” e, then the step is deemed “successful” and the next time step is estimated at:

$$h' = 0.8\sqrt[3]{e/\varepsilon}$$

If ε is greater than the “error tolerance,” then the time step h is reduced and the whole thing is tried again.

Comparing RK2-3 to Forward Euler

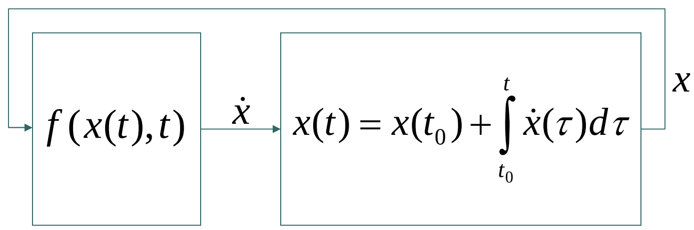




For this example, RK2-3 with variable step size computes far fewer points and is exact at 3.0, while Forward Euler undershoots by a significant amount.

Accumulating Errors

In feedback systems, the errors of forward Euler accumulate more rapidly than those of RK2-3.

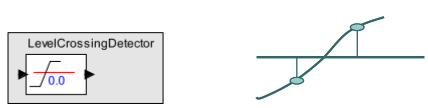


$$\dot{x}(t) = f(x(t),t)$$

$$f : R^m \times T \to R^m$$

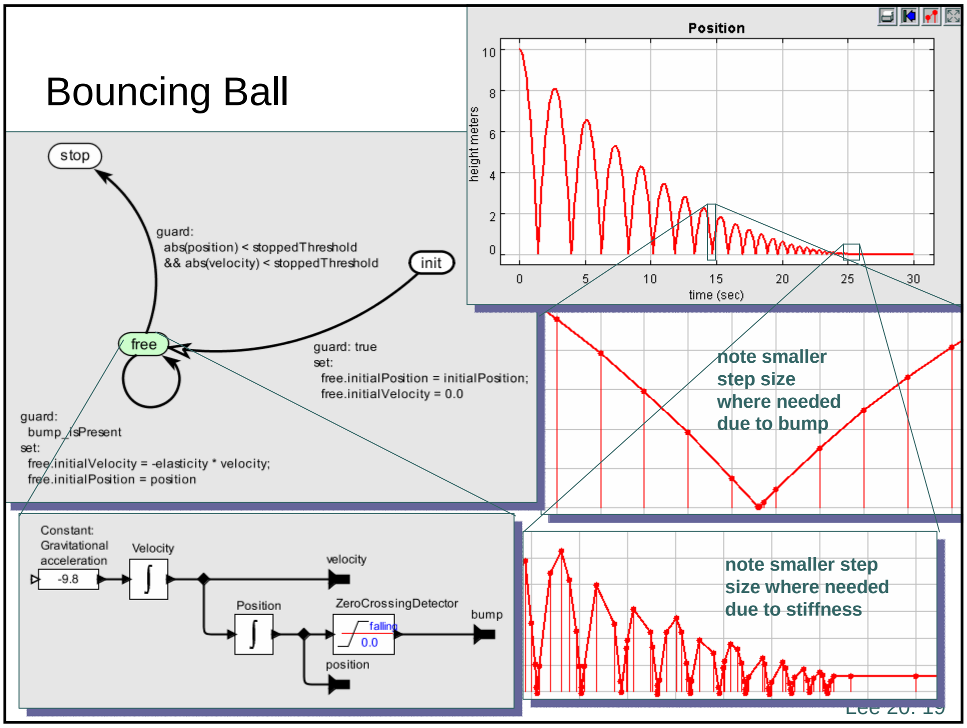
Adjusting the Time Steps due to Discrete Events

A step size *h* may cause the model to skip over a point where the behavior of the system changes abruptly:

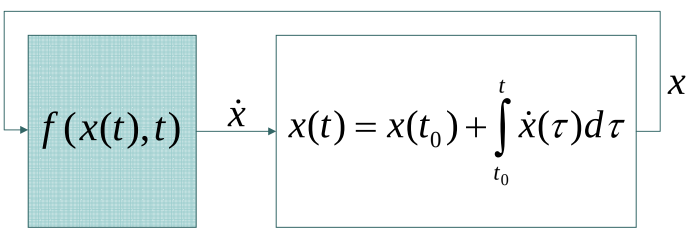


Such events must be detected and treated similarly as requiring a smaller step size.

Bouncing Ball



Continuous Time Model of Computation (MoC)

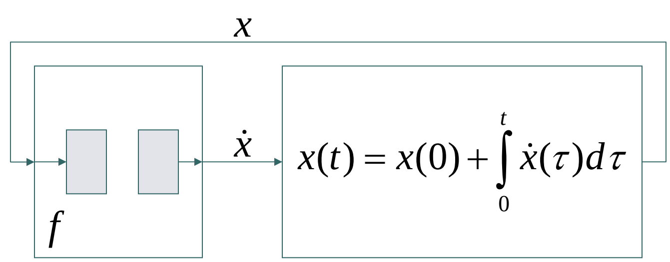


At each discrete time t_n, given a time increment $t_{n+1} = t_n + h$, we can estimate $x(t_{n+1})$ by repeatedly evaluating f with different values for the arguments. We may then decide that h is too large and reduce it and redo the process.

How General Is This MoC?

Does it handle:

- Systems without feedback? yes
- External inputs? yes
- State machines?

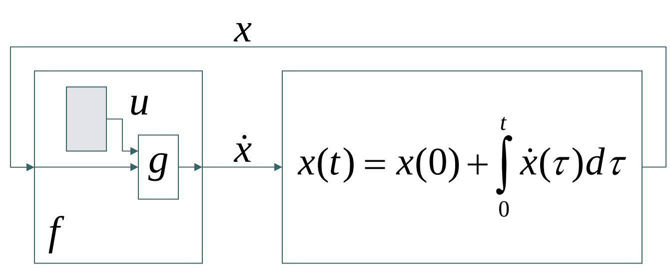


$$\dot{x}(t) = f(x(t), t)$$

How General Is This MoC?

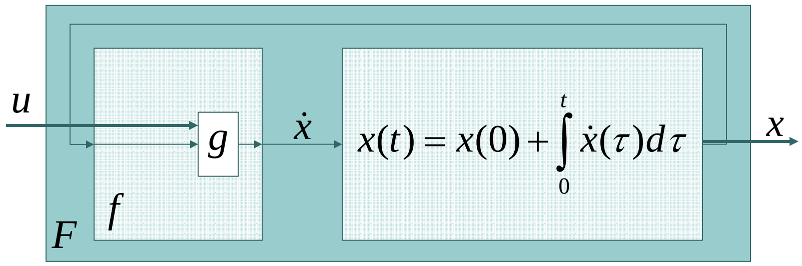
Does it handle:

- Systems without feedback?
- External inputs? yes
- State machines?



$$\dot{x}(t) = f(x(t), t) = g(u(t), x(t), t)$$

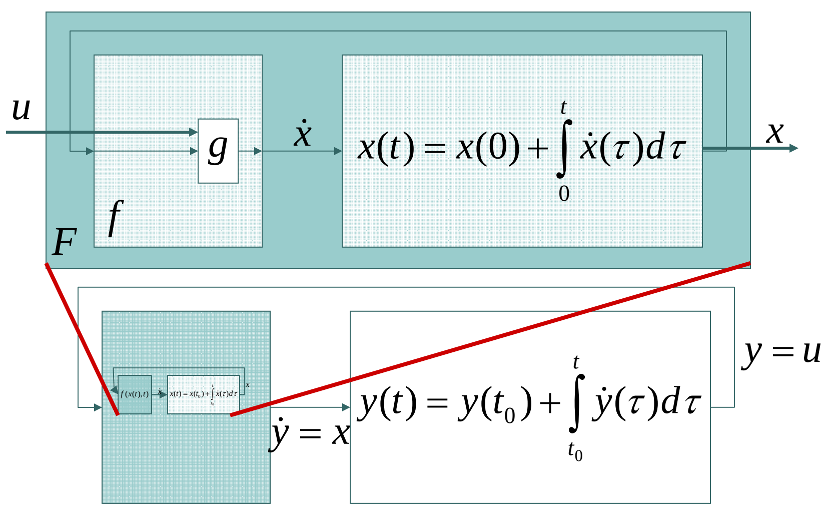
The Model Itself as a Function



Note that the model function has the form:

$$F : (T \to R^m) \to (T \to R^m)$$

Is the MoC Compositional?



For a model of computation to be *compositional*, it must be possible to turn a model into a component in another model.

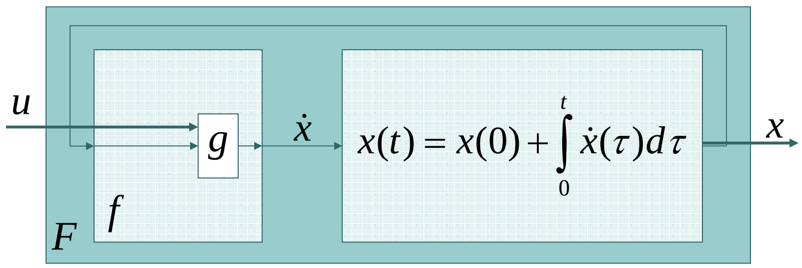
The Model Itself as a Function

Note that the model function has the form:

$$F : (T \to R^m) \to (T \to R^m)$$

Which does not match the form:

$$f : R^m \times T \to R^m$$

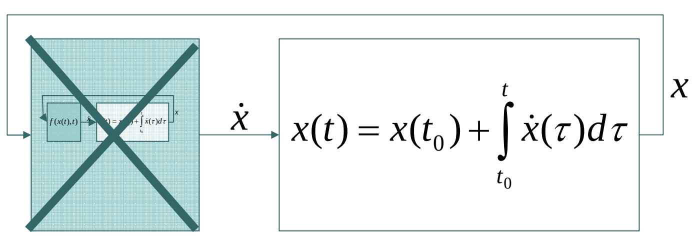


Given the model, we don't actually know the function *f*.

Consequently, the MoC is Not Compositional!

In general, the behavior of the inside dynamical system cannot be given by a function of form:

$$f : R^m \times T \to R^m$$

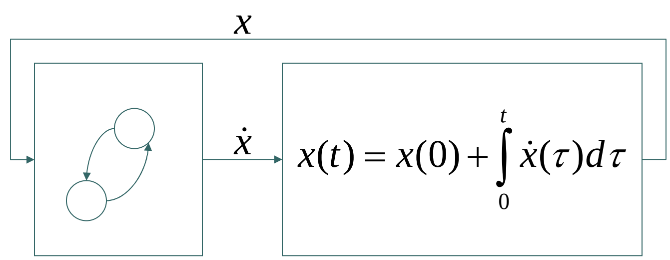


To see this, just note that the output must depend only on the current value of the input and the time to conform with this form.

So How General Is This MoC?

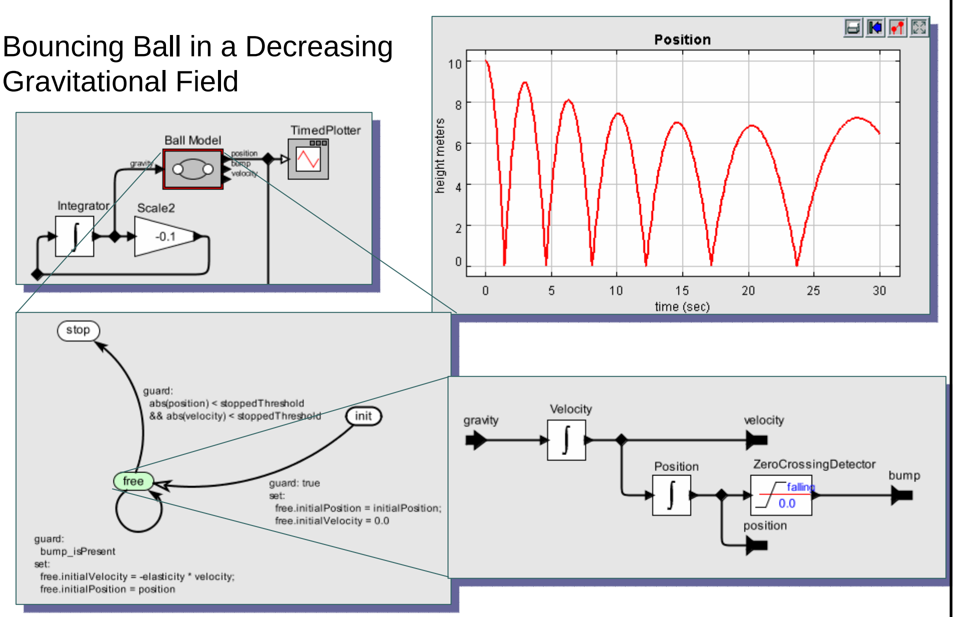
Does it handle:

- External inputs?
- Systems without feedback?
- State machines? No… The model needs work…



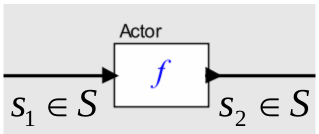
Since this model is itself a state machine, the inability to put a state machine in the left box explains the lack of compositionality.

Bouncing Ball in a Decreasing Gravitational Field



What Makes This Possible

Simple actor

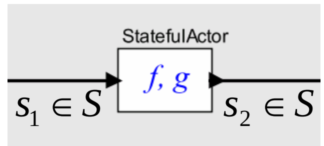


$S = (T \to R)$

$f : R^m \times T \to R^m$

$\forall t \in T, \quad s_2(t) = f(s_1(t), t)$

Actor with State



$S = (T \times N \to R)$

$f : \Sigma \times R^m \times T \to R^m$

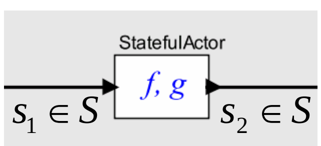
$g : \Sigma \times R^m \times T \to \Sigma$

state space

$\forall (t, n) \in T \times N, \quad s_2(t, n) = ?$

The new function f gives outputs in terms of inputs and the current state. The function g updates the state at the specified time.

Actors With State



$S = [T \times N \to R]$

$f : \Sigma \times R^m \times T \to R^m$

$g : \Sigma \times R^m \times T \to \Sigma$

At each $t \in T$ the output is a *sequence* of one or more values where given the current state $\sigma(t) \in \Sigma$ and the input $s_1(t)$ we evaluate the procedure

$$
\begin{aligned}
s_2(t,0) &= f(\sigma(t), s_1(t,0), t) \\
\sigma_1(t) &= g(\sigma(t), s_1(t,0), t) \\
s_2(t,1) &= f(\sigma_1(t), s_1(t,1), t) \\
\sigma_2(t) &= g(\sigma_1(t), s_1(t,1), t) \\
&\ldots
\end{aligned}
$$

until the state no longer changes. We use the final state on any evaluation at later times.