The Artisynth Modeling Framework

John E. Lloyd

June 18, 2005

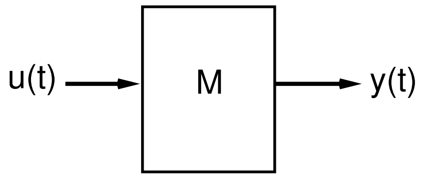
1 Overview

Artisynth is a system for simulating the physical and acoustical behavior of the human vocal tract and surrounding structures. It provides a general purpose environment which allows researchers to interconnect new or existing models of different vocal tract components and drive them from a single application.

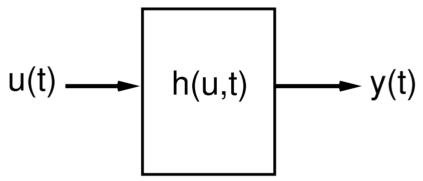
The Artisynth modeling framework supports this functionality with a set of Java classes and interfaces that defines what models are, how they can interact, and how they can be controlled, with the aim of allowing a collection of interacting models to be simulated as coherent whole.

2 Models

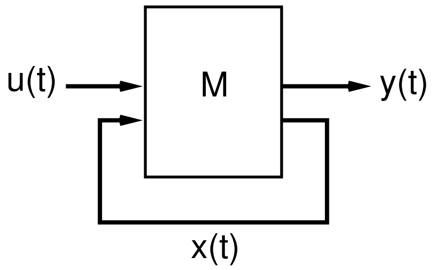
A *model* can be thought of as a process that maps inputs $u(t)$ to outputs $y(t)$ over time:



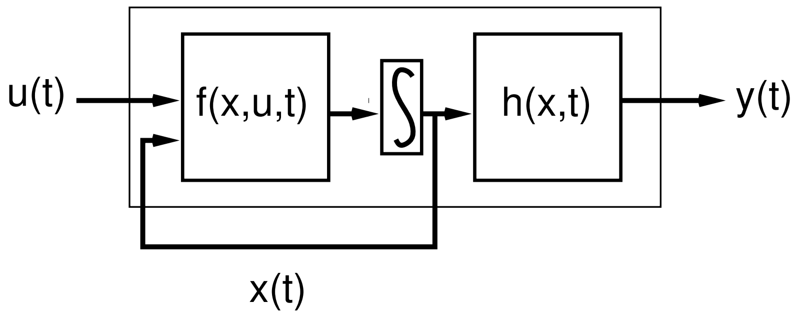
A *parametric* model is one in which the outputs are determined directly by a function $h(u, t)$ of the inputs and time:



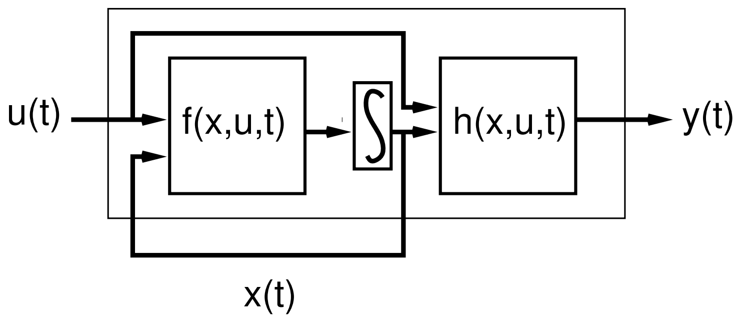
A *dynamic* model is one in which outputs are determined from a combination of time, inputs, and model state $x(t)$:



One particular form of a dynamic model is a *differential* model, in which the outputs are a function $h(x,t)$ of the state and time, and the state is given by the integral of a function $f(x,u,t)$:



A dynamic model could also be a hybrid combination of differential and parametric models:



2.1 Model Advancement and State

One of the main functions of Artisynth is to control and coordinate the advancement of a collection of models over time. Models are implemented by Java objects which implement the `Model` interface. To facilitate time advancement, models are expected to supply the method

```
void advance (long t0, long t1);
```

which advances themselves from time t_0 to t_1. Time is specified as a long integer in nano-seconds. Models are also expected to supply a method

```
void prepareForAdvance (long t1);
```

which performs any necessary preliminary computation prior to advancement. In particular, this could include calculating state-dependent information which is used by constraints for modifying the model input (as described below). The simulator first calls `prepareForAdvance()`, then adjusts the inputs to values appropriate for time t_1, and finally calls `advance()`:

```
model.prepareForAdvance (t1);
... set inputs to u(t1) ...
model.advance (t0, t1);
```

For parametric models, only time t_1 should be required to perform the advance, and it should also be possible to "advance" them to any t_1 in the past or future. For dynamic models, t_0 will generally be needed for advancement (e.g., a differential model needs the time increment $t_1 - t_0$), and it is assumed that they can only advance to future times (i.e., $t_1 \geq t_0$).

To move a dynamic model *back* in time, the system must explicitly set its state. To facilitate this, models should supply the method

```
void setState (ModelState state);
```

where `ModelState` is a generic model state interface. An empty state object for a particular model can be created using the factory method

```
ModelState createModelState();
```

and a model's state can be copied using

```
void getState (ModelState state);
```

If the model is stateless, then `createModelState` should return `null`. Whether or not a model has state can also be determined using the method

```
        boolean isDynamic();
```

Unlike parametric models, the advance of a dynamic model depends, in principle, on the input values $u(t)$ over the interval $t \in [t_0, t_1]$. However, because inputs are preset to $u(t_1)$, this detailed information is lost and will result in an error which increases with the size of the time step $t_1 - t_0$. Therefore, models are allowed to specify a maximum step size beyond which they will never be asked to advance, using the method

```
        long getMaxStepSize();
```

In general, when the simulator makes successive calls to `advance(t0, t1)`, t_0 will be equal to the previous value of t_1. However, this will not always be the case: the simulator may move around in time, as long as state information is set appropriately for dynamic models.

3 Constraints

A *constraint* is a relationship among a set of models that mutually affects their outputs. A constraint between two models that only affects the output of one is called a *dependency*, with the affected model being dependent on the unaffected model.

Constraints are enforced by objects which implement the `Constraint` interface. Each constraint object knows internally which models it affects, and enforcement is achieved by modifying the inputs or outputs of the affected models, using the methods

```
        void modifyInputs(long t1);

        void modifyOutput(long t1);
```

These methods are invoked before and after the models' advance methods, as illustrated by the following code fragment:

```
        ... update inputs to time t1 ...
        constraint.modifyInputs(t1);
        model1.advance (t0, t1);
        model2.advance (t0, t1);
        constraint.modifyOutputs(t1);
```

Given the definition of a constraint, the utility of the `modifyOutputs()` method is clear. What is perhaps less obvious is the need for `modifyInputs()`. This is primarily intended for constraints involving dynamic models, where it can be used to add constraining forces or input controls to help maintain constraint satisfaction (as illustrated by the example of Section 3.1). However, it may also be applicable to parametric models, as discussed in Section 3.2.

3.1 Input modification for mechanical models

In this section we show how a constraint between mechanical models (which are differential dynamic models of the kind shown in Section 2), gives rise to constraining forces which can be applied to the model inputs.

Consider two mechanical models, defined by

$$M_1\ddot{x}_1 - f_1(x_1, \dot{x}_1) = u_1(t), \quad y_1 = h_1(x_1(t)),$$

$$M_2\ddot{x}_2 - f_2(x_2, \dot{x}_2) = u_2(t), \quad y_2 = h_2(x_2(t)),$$

where M_i are mass matrices, $f_i(x_i, \dot{x}_i)$ are internal forces, and the inputs $u_i(t)$ are external forces.

A bilateral constraint between these two models takes the form

$$g(y_1(t), y_2(t)) = 0. \tag{1}$$

Differentiating this relationship yields the differential constraint

$$\begin{pmatrix} \frac{\partial g}{\partial y_1} & \frac{\partial g}{\partial y_2} \end{pmatrix} \begin{pmatrix} \delta y_1 \\ \delta y_2 \end{pmatrix} \;=\; 0, \tag{2}$$

which in turn implies a constraint on the state velocities given by

$$\begin{pmatrix} G_1 & G_2 \end{pmatrix} \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} \;=\; 0, \tag{3}$$

where

$$G_1 \equiv \frac{\partial g}{\partial y_1}\frac{\partial h_1}{\partial x_1} \quad \text{and} \quad G_2 \equiv \frac{\partial g}{\partial y_2}\frac{\partial h_2}{\partial x_2}.$$

Differentiating again yields a constraint on the state accelerations:

$$\begin{pmatrix} G_1 & G_2 \end{pmatrix} \begin{pmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{pmatrix} = \gamma$$

where γ contains the terms involving $\dot{G}_1$ and $\dot{G}_2$ (and hence accounts for artifacts such as Coriolis forces).

In general, the constraint will give rise to constraint forces acting on each system. For non-dissipative constraints, these forces will take the form $G_1^T \lambda$ and $G_2^T \lambda$, where λ is a vector of Lagrange multipliers. These constraint forces can be applied (within the method `modifyInputs()`) as a modification to the force inputs $u_i(t)$.

To determine λ, it is notationally useful to collect the two systems into one combined constrained system

$$M\ddot{x} - G^T \lambda - f(x, \dot{x}) = u(t),$$

$$G\ddot{x} = \gamma,$$

where

$$M \equiv \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix}, x \equiv \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, G \equiv \begin{pmatrix} G_1 & G_2 \end{pmatrix}, f \equiv \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, u \equiv \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}.$$

Letting $k \equiv u(t) + f(x, \dot{x})$ be the sum of the internal and external forces, one can then solve directly for λ:

$$\lambda = (GM^{-1}G^T)^{-1}(\gamma - GM^{-1}k). \tag{4}$$

3.2 Input modification for parametric models

A constraint of the form (1) between two parametric models can also produce differential constraints on the model inputs, which suggests that input modification might be useful in this situation as well.

Consider a constraint of the form (1) applied to two parametric models

$$y_1(t) = h_1(u_1(t), t) \quad \text{and} \quad y_2(t) = h_2(u_2(t), t).$$

If h_1 and h_2 are themselves differentiable, then the differential constraint (2) gives rise to a differential constraint on the inputs, given by

$$\begin{pmatrix} G_1 & G_1 \end{pmatrix} \begin{pmatrix} \delta u_1 \\ \delta u_2 \end{pmatrix} = 0 \tag{5}$$

where

$$G_1 \equiv \frac{\partial g}{\partial y_1} \frac{\partial h_1}{\partial u_1} \quad \text{and} \quad G_2 \equiv \frac{\partial g}{\partial y_2} \frac{\partial h_2}{\partial u_2}$$

3.3 Limitations Regarding Coupled Constraints

It is important to note that independent application of constraints, using the `modifyInputs()` and `modifyOutputs()` methods (as illustrated in Procedure 1, below) may not work properly in the case of coupled constraints. To see this, imagine that we have *two* constraints,

$$g_a(y_1(t), y_2(t)) = 0 \quad \text{and} \quad g_b(y_1(t), y_2(t)) = 0$$

acting on the mechanical systems described in Section 3.1. In this situation, G_1 and G_2 in (3) is formed from the composition of G_{a1} and G_{a2}, associated with g_a, and G_{b1} and G_{b2}, associated with g_b:

$$G_1 \equiv \begin{pmatrix} G_{a1} \\ G_{b1} \end{pmatrix} \quad \text{and} \quad G_2 \equiv \begin{pmatrix} G_{a2} \\ G_{b2} \end{pmatrix}.$$

The remainder of the analysis is unchanged, but the computation of λ in (4), from which the constraint forces are determined, requires the inversion of $GM^{-1}G^T$, where $G \equiv \begin{pmatrix} G_1 & G_2 \end{pmatrix}$. Any coupling between g_a and g_b will therefore affect the constraint forces, and in particular, the constraint objects associated with g_a and g_b will not be able to determine these independently except in very special cases.

The only way to correctly handle such a situation is for the simulator to collect appropriate derivative information from each constraint, and then compute a global solution for the required constraint forces. Constraint objects would need to supply this derivative information, and the models would also need to provide information such as mass matrices. Interfaces to facilitate this are presently being considered.

4 Accessing Model Data

An interface is provided which allows Artisynth to set and query specific data values within models and constraints. Among other things, this provides the default method for passing input and output data between models and the application.

Models and constraints which provide accessible datums should implement the `Accessible` interface. To get the value of a particular datum, an application first calls the `Accessible` method

```
Peek getPeek (String name, int idx);
```

to obtain a `Peek` object. This peek object in turn supplies the method

```
Object get();
```

which returns an object containing the value of the datum in question. Similarly, to set a particular datum, the `Accessible` method

```
Poke getPoke (String name, int idx);
```

is used to obtain a `Poke` object, which supplies a method

```
void put (Object obj);
```

which changes the datum's value.

It is up to the implementor of `Accessible` to determine which datums are accessible and in what way. Information about the names and index values of accessible datums can be returned by the method

```
AccessibleInfo[] getAccessibleInfo ();
```

where `AccessibleInfo` provides details about each accessible item.

5 Probes

Artisynth uses the Accessible interface described above to implement input and output *probes*, which arrange for the timed movement of information to and from models. Input probes set data, using `Poke` objects. Output probes read data, using `Peek` objects. Each probe has a *start* time and *stop* time, which specifies the time interval over which it is active, and an *update interval*, which specifies intermediate time events during this active interval (see Figure 1). Each intermediate time event, along with the start and stop times, is scheduled as an event within the Artisynth simulator, and thus presents a specific time to which the simulator advances.

Each probe implements the method

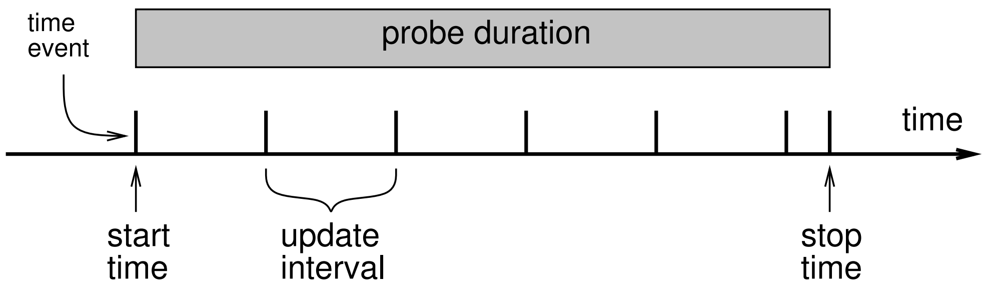


Figure 1: Time profile for a probe.

```
    void apply (long time);
```

which is used to apply the probe at a particular time. An output probe may be associated with several models and is applied whenever the simulator advances to one of its time events. An input probe is associated with only one model and is applied just before that model is advanced. Input probes therefore do not need to specify time events (using an update interval) in order to be applied. However, an input probe *will* be applied at every time event that it specifies.

6 Simulation Scheduling

The Artisynth simulator advances the models in concert through a sequence of time events, which are determined by input and output probes (note that graphical and audio display events can conceptually be considered as output probes). In other words, the simulator looks for the next probe time event and advances all the models to that time. If the time to the next event exceeds the maximum step size for a particular model (or set of models), then this advance is performed using a sequence of `advance()` calls, each with a time step not exceeding the prescribed maximum step size.

The order and maximum step size by which models are advanced is determined by the constraint relationships between them. The latter can be represented using a directed graph (Figure 2), with the model at the head of an edge being dependent on the model at the tail. Edges with two heads are equivalent to an edge in each direction, or a codependency between their respective models. To arrange the scheduling order, one first determines all

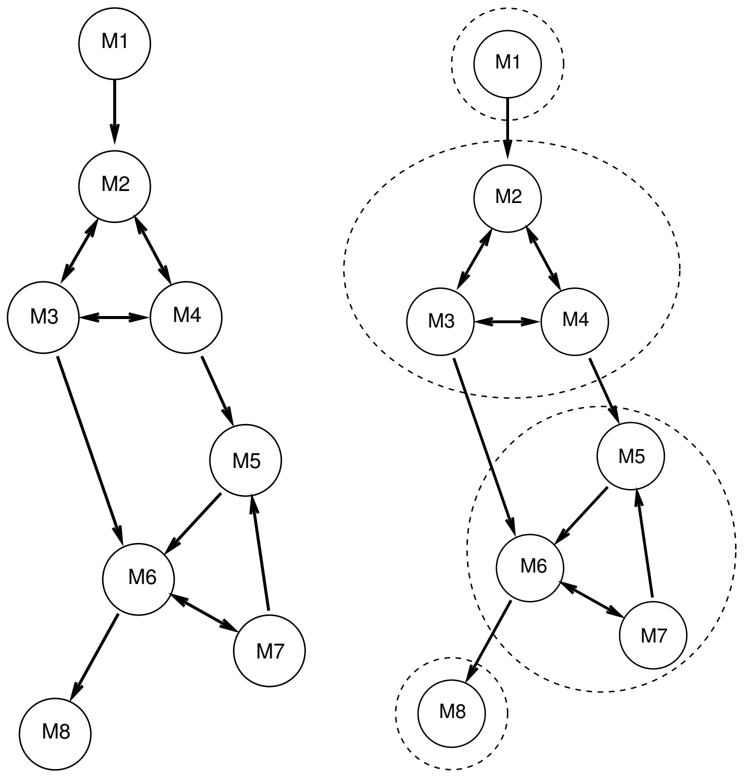


Figure 2: Directed graph showing the constraint relationships among a set of models. Dependency closures are indicated on the right by dashed enclosures.

the *dependency closures*. A dependency closure is a set of completely interdependent models; i.e., each model is directly or indirectly dependent on every other model in the set. The closures in Figure 2 are given by $\{M_1\}$, $\{M_2, M_3, M_4\}$, $\{M_5, M_6, M_7\}$, and $\{M_8\}$ (a linear time algorithm for computing such closures is given in Section 5.5 of Aho, Hopcroft, and Ullman, *The Design an Analysis of Computer Algorithms*).

All models within a closure are advanced together, with a step size given by the minimum of the maximum permitted step size for each model in the closure. This update is done according to the pseudo-code in Procedure 1. The set of dependency closures forms an *acyclic* directed graph, which can be used to determine an ordering which specifies which closures should be advanced first. Overall operation of the simulator is detailed in Procedure 2.

```
for (each model in closure)
 { model.prepareForAdvance(t1);
   for (each input probe)
    { probe.apply (t1);
    }
 }
for (each constraint on which this closure is dependent)
 { constraint.modifyInputs(t1);
 }
for (each constraint in closure)
 { constraint.modifyInputs(t1);
 }
for (each model in closure)
 { model.advance (t0, t1);
 }
for (each constraint in closure)
 { constraint.modifyOutputs(t1);
 }
```

Procedure 1: Advance all models in a closure from time t_0 to time t_1

```
t0 = 0;
while (simulating)
 { tnext = next probe event time;
   for (each closure, in dependent order)
    { h = closure.getMaxStepSize();
      while (t0 < tnext)
       { t1 = max (t0+h, tnext);
         closure.advance (t0, t1);
       }
      t0 = t1;
    }
   for (each output probe associated with this event)
    { probe.apply (t0);
    }
 }
```

Procedure 2: Overall simulator loop

7 External Integration

By default, an Artisynth model is expected to advance itself from one time to the next. For differential models, this implies that the model perform the necessary integration within the `advance()` method. However, Artisynth also supports the ability for models to be integrated externally, by the simulator itself, if they implement the `IntegrableModel` interface.

There are two reasons for providing this capability. The first is to allow the simulator to provide high-performance numerical integrators that would be difficult for model developers to implement on their own. The second is to allow the effects of constraints to be explicitly incorporated into the integration process.

Externally integrable models must have a state that can be represented as a vector and which can be obtained and set using the the methods

```
int getState (VectorNd x, int idx);

int setState (VectorNd x, int idx);
```

These methods transfer the state information to or from a vector `x` starting at an index location `idx`, and return the index value incremented by the model's state size. This allows the state for a whole set of models to be stored in a single vector and transfered using code fragments like this:

```
int idx = 0;
for (each model)
 { idx = model.setState (x, idx);
 }
```

Externally integrable models must also supply the state vector derivative at a particular time, using the method

```
int getDerivative (VectorNd dxdt, long time, int idx);
```

An accurate integrator often advances from time t_0 to t_1 by evaluating trial state and derivative information at a number of intermediate time values. Such evaluations are typically done as follows:

1. Estimate a new state x for time t_k based on previously obtained trial state and derivative information

2. Set the state at time t_k to x
3. Obtain a new trial derivative $\dot{x}$ for time t_k

Performing such trial evaluations on all the models within a dependency closure is complicated by the constraints themselves and by the fact that not all models may be externally integrable. Pseudo-code to perform such a trail evaluation is given in Procedure 3. Note that if an external integration procedure requires backtracking in time, then it will also be necessary to save and restore the states for any dynamic models which are not externally integrable.

The `IntegrableModel` interface also provides optional methods for obtaining information about a model's derivative Jacobian J, which an external integrator may require, particularly for performing implicit integration. The Jacobian is the partial derivative of the state derivative with respect to the state itself, so that if $\dot{x} = f(x)$, then

$$J \equiv \frac{\partial f}{\partial x}.$$

The method for obtaining J is

```
    boolean getJacobian (MatrixNd Jtotal, long time, int idx);
```

which copies J, evaluated at a particular time, into a (possibly larger) system Jacobian `Jtotal`, starting at the row and column indices specified by `idx`. Because J can be difficult to compute, implementation of this method is optional; if not implemented, it should return false.

```
VectorNd x;    // state estimate
VectorNd dxdt; // new derivative estimate

for (each model in closure)
 { model.prepareForAdvance(t1);
   for (each input probe)
    { probe.apply (t1);
    }
 }

x = new state estimate for all externally integrable models;

for (each constraint on which this closure is dependent)
 { constraint.modifyInputs(t1);
 }
for (each constraint in closure)
 { constraint.modifyInputs(t1);
 }
idx = 0;
for (each model in closure)
 { if (model is externally integrable)
    { idx = model.setState (x, idx);
    }
   else
    { model.advance (t0, t1);
    }
 }
for (each constraint in closure)
 { constraint.modifyOutputs(t1);
 }
idx = 0;
for (each externally integrable model)
 { idx = model.getDerivative (dxdt, idx);
 }
```

Procedure 3: Trial update from time t_0 to time t_1 within a dependency closure.