

Discrete event simulation of continuous systems

James Nutaro
Oak Ridge National Laboratory
nutarojj@ornl.gov

1 Introduction

Computer simulation of a system described by differential equations requires that some element of the system be approximated by discrete quantities. There are two system aspects that can be made discrete; time and state. When time is discrete, the differential equation is approximated by a difference equation (i.e., a discrete time system), and the solution is calculated at fixed points in time. When the state is discrete, the differential equation is approximated by a discrete event system. Events correspond to jumps through the discrete state space of the approximation.

The essential feature of a discrete time approximation is that the resulting difference equations map a discrete time set to a continuous state set. The time discretization need not be regular. It may even be revised in the course of a calculation. None the less, the elementary features of a discrete time base and continuous state space remain.

The basic feature of a discrete event approximation is opposite that of a discrete time approximation. The approximating discrete event system is a function from a continuous time set to a discrete state set. The state discretization need not be uniform, and it may even be revised as the computation progresses.

These two different types of discretizations can be visualized by considering how the function $x(t)$, shown in figure 1a, might be reduced to discrete points. In a discrete time approximation, the value of the function is observed at regular intervals in time. This kind of discretization is shown in figure 1b. In a discrete event approximation, the function is sampled when it takes on regularly spaced values. This type of discretization is shown in figure 1c.

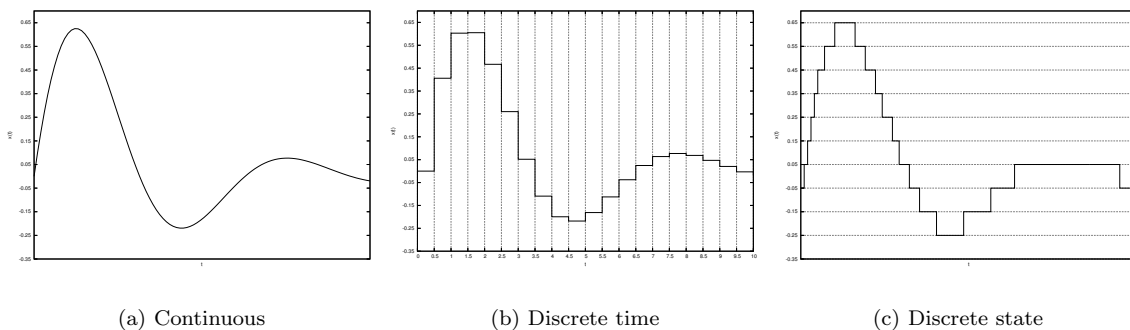


Figure 1: Time and state discretizations of a system.

From an algorithmic point of view, these two types of discretizations are widely divergent. The first approach emphasizes the simulation of coupled difference equations. Some distinguishing features of a difference equation simulator are nested "for" loops (used to compute function values at each time step), SIMD type parallel computing (using, e.g., vector processors or automated "for" loop parallelization), and good locality of reference.

The second approach emphasizes the simulation of discrete event systems. The main computational features of a discrete event simulation are very different from a discrete time simulation. Foremost among

them are event scheduling, poor locality of reference, and MIMD type asynchronous parallel algorithms. The essential data structures are different, too. Where difference equation solvers exploit a matrix representation of the system coupling, discrete event simulations often require different, but structurally equivalent, data structures (e.g., influence graphs).

Mathematically, however, they share several features. The approximation of functions via interpolation and extrapolation are central to both. Careful study of error bounds, stability regimes, conservation properties, and other elements of the approximating machinery is essential. It is not surprising that theoretical aspects of differential operators, and their discrete approximations, have a prominent place in the study of both discrete time and discrete event numerical methods.

This confluence of applied mathematics, mathematical systems theory, and computer science makes the study of discrete event numerical methods particularly challenging. This paper presents some basic results, and it avoids more advanced topics. My goal is to present essential concepts clearly, and so portions of this material will, no doubt, seem underdeveloped to a specialist. Pointers into the appropriate literature are provided for those who want a more in depth treatment.

2 Simulating of a single ordinary differential equation

Consider an ordinary differential equation that can be written in the form of

$$\dot{x}(t) = f(x(t)). \tag{1}$$

A discrete event approximation of this system can be obtained in, at least, two different ways. To begin, consider the Taylor series expansion

$$\dot{x}(t+h) = x(t) + h\dot{x}(t) + \sum_{n=2}^{\infty} \frac{h^n}{n!} x^{(n)}(t). \tag{2}$$

If we fix the quantity $D = |x(t+h) - x(t)|$, then the time required for a change of size D to occur in $x(t)$ is approximately

$$h = \begin{cases} \frac{D}{|\dot{x}(t)|} & \text{if } \dot{x}(t) \neq 0 \\ \infty & \text{otherwise} \end{cases}. \tag{3}$$

This approximation drops the summation term in equation 2 and rearranges what is left to obtain h. Algorithm 1 uses this approximation to simulate a system described by 1. The procedure computes successive approximations to $x(t)$ on a grid in the phase space of the system. The resolution of the phase space grid is D, and h approximates the time at which the solution jumps from one phase space grid point to the next.

The *sgn* function at line 14 in algorithm 1, is defined to be

$$sgn(q) = \begin{cases} -1 & \text{if } q < 0 \\ 0 & \text{if } q = 0 \\ 1 & \text{if } q > 0 \end{cases}.$$

The expression $Dsgn(f(x))$ on line 14 could, in this instance, be replaced by $hf(x)$ because

$$hf(x) = \frac{D}{|f(x)|} f(x) = Dsgn(f(x)).$$

However, the expression $Dsgn(f(x))$ highlights the fact that the state space, and not the time domain, is discrete. Notice, in particular, that the computed values of x are restricted to $x(0) + kD$, where k is an integer and D is the phase space grid resolution. In contrast to this, the computed values of t can take any value.

The procedure can be demonstrated with a simulation of the system $\dot{x}(t) = -x(t)$, with $x(0) = 1$ and $D = 0.15$. Each step of the simulation is shown in table 1. Figure 2 shows the computed $x(t)$ as a function of t.

Algorithm 1 Simulating a single ordinary differential equation.

```

t ← 0
x ← x(0)
while terminating condition not met do
  print t , x
  if f(x) = 0 then
    h ← ∞
  else
    h ←  $\frac{D}{|f(x)|}$ 
  end if
  if h = ∞ then
    stop simulation
  else
    t ← t + h
    x ← x + D sgn(f(x))
  end if
end while

```

t	x	f(x)	h
0.0	1.0	-1.0	0.15
0.15	0.85	-0.85	0.1765
0.3265	0.7	-0.7	0.2143
0.5408	0.55	-0.55	0.2727
0.8135	0.4	-0.4	0.3750
1.189	0.25	-0.25	0.6
1.789	0.1	-0.1	1.5
3.289	-0.05	0.05	3.0
6.289	0.1	-0.1	1.5
7.789	-0.05	0.05	3.0

Table 1: Simulation of $\dot{x}(t) = -x(t)$, $x(0) = 1$, using algorithm 1 with $D = 0.15$.

The approximation given by equation 3 can be obtained in a second way. Consider the integral

$$\left| \int_{t_0}^{t_0+h} f(x(t)) dt \right| = D. \quad (4)$$

As before, D is the resolution of the phase space grid and h is the time required to move from one point in the phase space grid to the next. For time to move forward, it is required that $h > 0$. In the interval $[t_0, t_0 + h]$, the function $f(x(t))$ can be approximated by $f(x(t_0))$. Substituting this approximation into 4 and solving for h gives

$$h = \begin{cases} \frac{D}{|f(x(t_0))|} & \text{if } f(x(t_0)) \neq 0 \\ \infty & \text{otherwise} \end{cases}.$$

This approach to obtaining h gives the same result as before.

There are two important questions that need answering before this can be considered a viable simulation procedure. First, can the discretization parameter D be used to bound the error in the simulation? Second, under what conditions is the simulation procedure stable? That is, under what circumstances can the error at the end of an arbitrarily long simulation run be bounded? Several authors (see, e.g., [25], [8], and [13]) have addressed these questions in a rigorous way. Happily, the answer to the first question is a yes! The second question, while answered satisfactorily for linear systems, remains (not surprisingly) largely unresolved for non-linear systems.

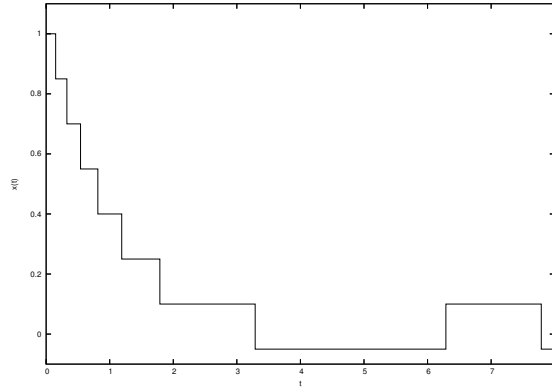
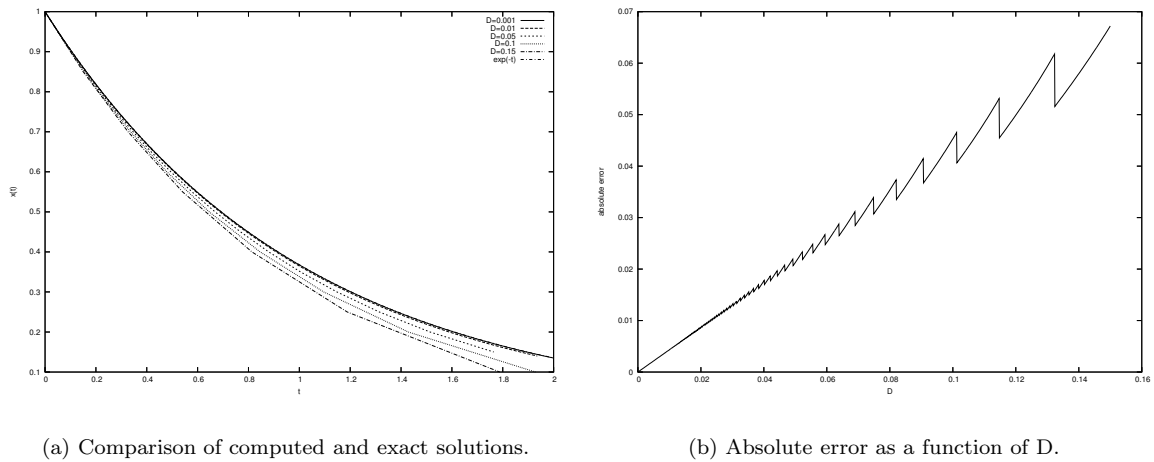


Figure 2: Computed solution of $\dot{x}(t) = -x(t)$, $x(0) = 1$ with $D = 0.15$.

The first question can be answered as follows: If $\dot{x}(t) = f(x(t))$ describes a stable and time invariant system (see [19], or most any other introductory systems textbook), then the error at any point in a simulation run is proportional to D . The constant of proportionality is determined by the system under consideration. The time invariant caveat is needed to avoid a situation in which the first derivative can change independently of $x(t)$ (i.e., the derivative is described by a function $f(x(t), t)$, rather than $f(x(t))$). In practice, this problem can often be overcome by treating the time varying element of $f(x(t), t)$ as a quantized input to the integrator (see, e.g., [12]).

The linear dependence of the simulation error on D is demonstrated for two different systems in figures 3 and 4. In these examples, $x(t)$ is computed until the time of next event exceeds a preset threshold. The error is determined at the last event time by taking the difference of the computed and known solutions. This linear dependency is strongly related to the fact that the scheme is exact when $x(t)$ is a line, or, equivalently, when the system is described by $\dot{x}(t) = k$, where k is a constant.



(a) Comparison of computed and exact solutions.

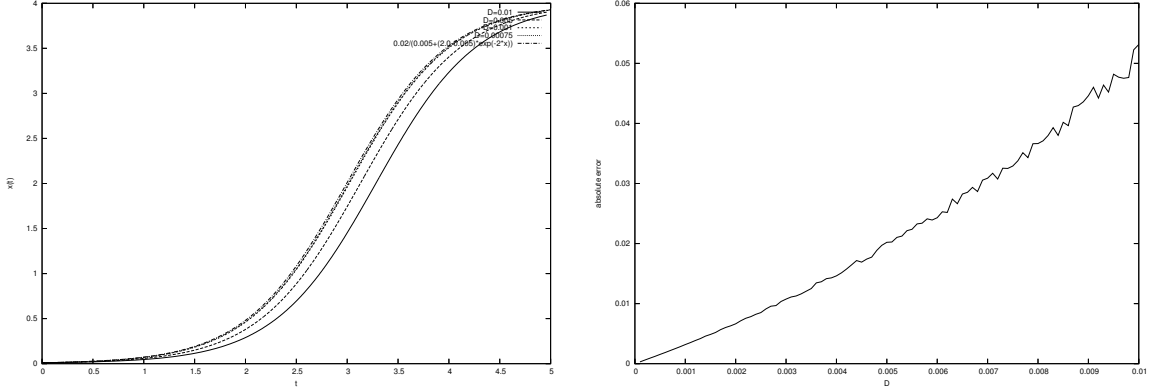
(b) Absolute error as a function of D .

Figure 3: Error in the computed solution of $\dot{x}(t) = -x(t)$, $x(0) = 1$.

3 Simulation of coupled ordinary differential equations

Algorithm 1 can be readily extended to sets of coupled ordinary differential equations. Consider a system described by equations in the form

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



(a) Comparison of computed and exact solutions.

(b) Absolute error as a function of D.

Figure 4: Error in the computed solution of $\dot{x}(t) = (2 - 0.5x(t))x(t)$, $x(0) = 0.01$.

where \bar{x} is the vector

$$[x_1(t), x_2(t), \dots, x_m(t)]$$

and $\bar{f}(\bar{x})$ is a function vector

$$[f_1(\bar{x}(t)), f_2(\bar{x}(t)), \dots, f_m(\bar{x}(t))].$$

As before, we construct a grid in the m dimensional state space. The grid points are regular spaced by a distance D along the state space axes. To simulate this system, four variables are needed for each x_i , and so $4m$ variables in total. These variables are

x_i , the position of state variable i on its phase space axis,

tN_i , the time until x_i reaches its next discrete point on the i th phase space axis,

y_i , the last grid point occupied by the variable x_i , and

tL_i , the last time at which the variable x_i was modified.

The x_i and y_i are necessary because the function $f_i(\cdot)$ is computed only at grid points in the discrete phase space. Because of this, the motion of the variable x_i along its phase space axis is described by a piecewise constant velocity. This velocity is computed using the differential function $f_i(\cdot)$ and the vector $\bar{y} = [y_1, \dots, y_m]$. The value of y_i is updated when x_i reaches a phase space grid point. The time required for the variable x_i to reach its next grid point is computed as

$$h = \begin{cases} \frac{D - |x_i - y_i|}{|f_i(\bar{y})|} & \text{if } f_i(\bar{y}) \neq 0 \\ \infty & \text{otherwise} \end{cases} \quad (6)$$

The quantity D is the distance separating grid points along the axis of motion, $|x_i - y_i|$ is the distance already traveled along the axis, and $f_i(\bar{y})$ is the velocity on the i th phase space axis.

With equation 6, and an extra variable t to keep track of the simulation time, the behavior of a system described by equation 5 can be computed with algorithm 2.

To illustrate the algorithm, consider the coupled linear system

$$\begin{aligned} \dot{x}_1(t) &= -x_1(t) + 0.5x_2(t) \\ \dot{x}_2(t) &= -0.1x_2(t) \end{aligned} \quad (7)$$

with $x_1(0) = x_2(0) = 1$ and $D = 0.1$. Table 2 gives a step by step account of the first eight iterations of algorithm 2 applied to this system. The output values computed by the procedure are plotted in figure 3

Algorithm 2 Simulating a system of coupled ordinary differential equations.

```

t ← 0
for all i ∈ [0, m] do
  tLi ← 0
  yi ← xi(0)
  xi ← xi(0).
end for
while terminating condition not met do
  print t, y1, ..., ym
  for all i ∈ [0, m] do
    tNi ← tLi + hi, where hi is given by equation 6.
  end for
  t ← min{tN1, tN2, ..., tNm}
  Copy  $\bar{y}$  to a temporary vector  $\bar{y}_{tmp}$ 
  for all i ∈ [0, m] such that tNi = t do
    yi ← yi + D sgn(fi( $\bar{y}_{tmp}$ ))
    xi ← yi
    tLi ← t
  end for
  for all j ∈ [0, m] such that a changed yi alters the value of fj( $\bar{y}$ ) and tNj ≠ t do
    xj ← xj + (t - tLj)fj( $\bar{y}_{tmp}$ )
    tLj ← t
  end for
end while

```

t	x ₁	\dot{x}_1	y ₁	tL ₁	h ₁	x ₂	\dot{x}_2	y ₂	tL ₂	h ₂
0	1	-0.5	1	0	0.2	1	-0.1	1	0	1
0.2	0.9	-0.4	0.9	0.2	0.25					
0.45	0.8	-0.3	0.8	0.45	0.3333					
0.7833	0.7	-0.2	0.7	0.7833	0.5					
1	0.6567	-0.25		1.0	0.5733	0.9	-0.09	0.9	1	1.111
1.573	0.6	-0.15	0.6	1.573	0.6667					
2.111	0.5193	-0.2		2.111	0.9033	0.8	-0.08	0.8	2.111	1.25
3.014	0.5	-0.1	0.5	3.014	1					

Table 2: Simulation of two coupled ordinary differential equations on a discrete phase space grid.

(note that the figure shows results beyond the eight iterations listed in the table). Each row in the table shows the computed values at the end of an iteration (i.e., just prior to repeating the while loop). Blank entries indicated that the variable value did not change in that iteration. The blank entries, and the irregular time intervals that separate iterations, highlight the fact that this is a discrete event simulation. An event is the arrival of a state variable at its next grid point in the discrete phase space. Clearly, not every variable arrives at its next phase space point at the same time, and so event scheduling provides a natural way to think about the evolution of the system.

Stability and error properties in the case of coupled equations are more difficult to reason about, but they generally reflect the one dimensional case. In particular, the simulation procedure is stable, in the sense that the error can be bounded at the end of an arbitrarily long run, when it is applied to a stable and time invariant linear system (see [8] and [25]). The final error resulting from the procedure is proportional to the phase space grid resolution D (see [8] and [25]).

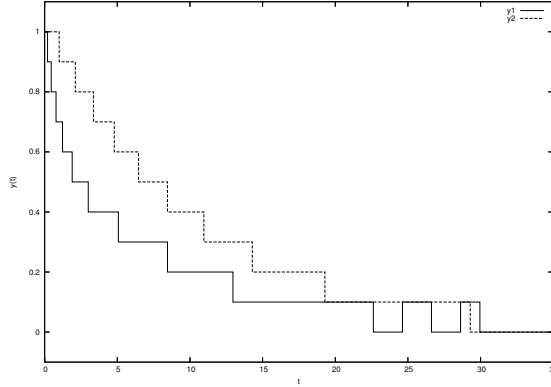


Figure 5: Plot of $y(t)$ for the calculation shown in table 2.

4 DEVS representation of discrete event integrators

It is useful to have a compact representation of the integration scheme that is readily implemented on a computer, can be extended to produce new schemes, and provides an immediate support for parallel computing. The Discrete Event System Specification (DEVS) satisfies this need. A detailed treatment of DEVS can be found in [24]. Several simulation environments for DEVS are readily available online (e.g., PowerDEVS [7], adevs [10], DEVSJAVA [26], CD++ [22], and JDEVS [3] to name just a few).

DEVS uses two types of structures to describe a discrete event system. Atomic models describe the behavior of elementary components. Here, an atomic model will be used to represent individual integrators and differential functions. Coupled models describe collections of interacting components, where components can be atomic and coupled models. In this application, a coupled model describes a system of equations as interacting integrators and function blocks.

An atomic model is described by a set of inputs, set of outputs, and set of states, a state transition function decomposed into three parts, an output function, and a time advance function. Formally, the structure is

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

where

X is a set of inputs,

Y is a set of outputs,

S is a set of states,

$\delta_{int} : S \rightarrow S$ is the internal state transition function,

$\delta_{ext} : Q \times X^b \rightarrow S$ is the external state transition function

with $Q = \{(s, e) \mid s \in S \& 0 \leq e \leq ta(s)\}$

and X^b is a bag of values appearing in X ,

$\delta_{con} : S \times X^b \rightarrow S$ is the confluent state transition function,

$\lambda : S \rightarrow Y$ is the output function, and

$ta : S \rightarrow \mathfrak{R}$ is the time advance function.

The external transition function describes how the system changes state in response to input. When input is applied to the system, it is said that an external event has occurred. The internal transition function describes the autonomous behavior of the system. When the system changes state autonomously, an internal event is said to have occurred. The confluent transition function determines the next state of the system when an internal event and external event coincide. The output function generates output values at times that coincide with internal events. The output values are determined by the state of the system

just prior to the internal event. The time advance function determines the amount of time that must elapse before the next internal event will occur, assuming that no input arrives in the interim.

Coupled models are described by a set of components and a set of component output to input mappings. For our purpose, we can restrict the coupled model description to a flat structure (i.e., a structure composed entirely of atomic models) without external input or output coupling (i.e., the component models can not be affected by elements outside of the network). With these restrictions, a coupled model is described by the structure

$$N = \langle \{M_k\}, \{z_{ij}\} \rangle$$

where

$\{M_k\}$ is a set of atomic models, and

$\{z_{ij}\}$ is a set of output to input maps $z_{ij} : Y_i \rightarrow X_j \cup \Phi$

where the i and j indices correspond to M_i and M_j in $\{M_k\}$ and Φ is the non-event.

The output to input maps describe which atomic models can affect one another. The output to input map is, in this application, somewhat over generalized and could be replaced with more conventional descriptions of computational stencils and block diagrams. The non-event is used, in this instance, to represent components that are not connected. That is, if component i does not influence component j, then $z_{ij}(x_i) = \Phi$, where $x_i \in X_i$.

These structures describe what a model can do. A canonical simulation algorithm is used to generate dynamic behavior from the description. In fact, algorithms 1 and 2 are special cases of the DEVS simulation procedure. The generalized procedure is given as algorithm 3. Its description uses the same variables as algorithm 2 wherever this is possible. Algorithm 3 assumes a coupled model N, with a component set $\{M_1, M_2, \dots, M_n\}$, and a suitable set of output to input maps. For every component model M_i , there is a time of last event and time of next event variable tL_i and tN_i , respectively. There are also state, input, and output variables s_i , x_i , and y_i , in addition to the basic structural elements (i.e., state transition functions, output function, and time advance function). The variables x_i and y_i are bags, with elements taken from the input and output sets X_i and Y_i , respectively. The simulation time is kept in variable t.

To map algorithm 2 into a DEVS model, each of the x variables is associated with an atomic model called an integrator. The input to the integrator is the value of the differential function, and the output of the integrator is the appropriate y variable. The integrator has four state variables

- q_l , the last output value of the integrator,
- q , the current value of the integral,
- \dot{q} , the last known value of the derivative, and
- σ , the time until the next output event.

The integrator's input and output events are real numbers. The value of an input event is the derivative at the time of the event. An output event gives the value of the integral at the time of the output.

The integrator generates an output event when the integral of the input changes by D. More generally, if Δq is the desired change, $[t_0, T]$ is the interval over which the change occurs, and $f(x(t))$ is the first derivative of the system, then

$$\int_0^T f(x(t_0 + t)) dt = F(T) = \Delta q. \quad (8)$$

The function $F(T)$ gives the exact change in $x(t)$ over the interval $[t_0, T]$. Equation 8 is used in two ways. If $F(T)$ and Δq are known, then the time advance of the discrete event integrator is found by solving for T . If $F(T)$ and T are known, then the next state of the integrator is given by $q + F(T)$, where T is equal to the elapsed time (for an external event) or time advance (for an internal event).

Algorithm 3 DEVS simulation algorithm.

```
 $t \leftarrow 0$  {Initialize the models}
for all  $i \in [1, n]$  do
   $tL_i \leftarrow 0$ 
  sets  $s_i$  to the initial state of  $M_i$ 
end for
while terminating condition not met do
  for all  $i \in [1, n]$  do
     $tN_i \leftarrow tL_i + ta(s_i)$ 
    Empty the bags  $x_i$  and  $y_i$ 
  end for
   $t \leftarrow \min\{tN_i\}$ 
  for all  $i \in [1, n]$  do
    if  $tN_i = t$  then
       $y_i \leftarrow \lambda_i(s_i)$ 
      for all  $j \in [1, n]$  &  $j \neq i$  &  $z_{ij}(y_i) \neq \Phi$  do
        Add  $z_{ij}(y_i)$  to the bag  $x_j$ 
      end for
    end if
  end for
  for all  $i \in [1, n]$  do
    if  $tN_i = t$  &  $x_i$  is empty then
       $s_i \leftarrow \delta_{int,i}(s_i)$ 
       $tL_i \leftarrow t$ 
    else if  $tN_i = t$  &  $x_i$  is not empty then
       $s_i \leftarrow \delta_{con,i}(s_i, x_i)$ 
       $tL_i \leftarrow t$ 
    else if  $tN_i \neq t$  &  $x_i$  is not empty then
       $s_i \leftarrow \delta_{ext,i}(s_i, t - tL_i, x_i)$ 
       $tL_i \leftarrow t$ 
    end if
  end for
end while
```

The integration scheme used by algorithms 1 and 2 approximates $f(x(t))$ with a piecewise constant function. At any particular time, the value of the approximation is given by the state variable \dot{q} . Using \dot{q} in place of $f(x(t_0 + T))$ in equation 8 gives

$$\int_0^T \dot{q} dt = \dot{q}T.$$

When \dot{q} and T are known, then the function

$$\hat{F}(T, \dot{q}) = \dot{q}T \quad (9)$$

approximates $F(T)$. Because T must be positive (i.e., we are simulating forward in time), the inverse of equation 9 can not be used to compute the time advance. However, the absolute value of the inverse,

$$\hat{F}^{-1}(\Delta q, \dot{q}) = \begin{cases} \frac{\Delta q}{|\dot{q}|} & \text{if } \dot{q} \neq 0 \\ \infty & \text{otherwise} \end{cases} \quad (10)$$

is suitable.

The state transition, output, and time advance functions of the integrator can be defined in terms of equations 9 and 10. This gives

$$\begin{aligned} \delta_{int}((q_l, q, \dot{q}, \sigma)) &= \\ & (q + \hat{F}(\sigma, \dot{q}), q + \hat{F}(\sigma, \dot{q}), \dot{q}, \hat{F}^{-1}(D, \dot{q})), \\ \delta_{ext}((q_l, q, \dot{q}, \sigma), e, x) &= \\ & (q_l, q + \hat{F}(e, \dot{q}), x, \hat{F}^{-1}(D - |q + \hat{F}(e, \dot{q}) - q_l|, x)), \\ \delta_{con}((q_l, q, \dot{q}, \sigma), x) &= \\ & (q + \hat{F}(\sigma, \dot{q}), q + \hat{F}(\sigma, \dot{q}), x, \hat{F}^{-1}(D, x)), \\ \lambda((q_l, q, \dot{q}, \sigma)) &= q + \hat{F}(\sigma, \dot{q}), \quad \text{and} \\ ta((q_l, q, \dot{q}, \sigma)) &= \sigma. \end{aligned}$$

In this definition, \hat{F} computes the next value of the integral using the previous value, the approximation of $f(x(t))$ (i.e., \dot{q}), and the time elapsed since the last state transition. The time that will be needed for the integral to change by an amount D is computed using \hat{F}^{-1} . The arguments to \hat{F}^{-1} are the distance remaining (i.e., D minus the distance already traveled) and the speed with which the distance is being covered (i.e., the approximation of $f(x(t))$).

An implementation of this definition is shown in figure 6. This implementation is for the adevs simulation library. The implementation is simplified by taking advantage of two facts. First, the output values can be stored in a shared array that is accessed directly, rather than via messages. Second, the derivative value, represented as an input in the formal expression, can be calculated directly from the shared array of output values whenever a transition function is executed.

The integrator class is derived from the atomic model class, which is part of the adevs simulation library. The atomic model class has virtual methods corresponding with the output and state transition functions of the DEVS atomic structure. The time advance function for an adevs model is defined as $ta(\sigma)$, where σ is a state variable of the atomic model, and its value is set with the *hold*(\cdot) method. The integrator class adds a new virtual method, $f(\cdot)$, that is specialized to compute the derivative function using the output value vector \bar{y} .

A DEVS simulation of a system of ordinary differential equations, using algorithm 3, gives the same result as algorithm 2. This is demonstrated by a simulation of the two equation system 7. The code used to execute the simulation is shown in figure 7. The state transitions and output values computed in the course of the simulation is shown in table 3. A comparison of this table with table 2 confirms that they are identical.

```

class Integrator: public atomic {
public:
    /* Arguments are the initial variable value, variable index,
    integration quantum, and an array for storing output values. */
    Integrator(double q0, int index, double D, double* x):
    atomic(),index(index),q(q0),D(D),x(x) { x[index] = q; }
    /* Initialize the state prior to start of the simulation. */
    void init() {
        dq = f(index,x); compute_sigma();
    }
    /* DEVS state transition functions. */
    void delta_int() {
        q = x[index]; dq = f(index,x); compute_sigma();
    }
    void delta_ext(double e, const adevs_bag<PortValue>& xb) {
        q += e*dq; dq = f(index,x); compute_sigma();
    }
    void delta_conf(const adevs_bag<PortValue>& xb) {
        q = x[index]; dq = f(index,x); compute_sigma();
    }
    /* DEVS output function. */
    void output_func(adevs_bag<PortValue>& yb) {
        x[index] += D*sgn(dq);
        output(cell_interface::out,NULL,yb); // Notify influencees of change.
    }
    /* Event garbage collection function. */
    void gc_output(adevs_bag<PortValue>& g){}
    /* Virtual derivative function. */
    virtual double f(int index, const double* x) = 0;
private:
    /* Index of the variable associated with this integrator. */
    int index;
    /* Value of the variable, its derivative, and the integration quantum. */
    double q, dq, D;
    /* Shared output variable vector. */
    double* x;
    /* Sign function. */
    static double sgn(double z) {
        if (z > 0.0) return 1.0; if (z < 0.0) return -1.0; return 0.0;
    }
    /* Set the value of the time advance function. */
    void compute_sigma() {
        if (fabs(dq) < ADEVS_EPSILON) hold(ADEVS_INFINITY);
        else hold(fabs((D-fabs((q-x[index])))/dq));
    }
};

```

Figure 6: Code listing for the Integrator class.

```

/* Integrator for the two variable system. */
class TwoVarInteg: public Integrator {
public:
    TwoVarInteg(double q0, int index, double D, double* x):
    Integrator(q0,index,D,x){}
    /* Derivative function. */
    double f(int index, const double* x) {
        if (index == 0) return -x[0]+0.5*x[1];
        else return -0.1*x[1];
    }
};

int main() {
    double x[2];
    TwoVarInteg* intg[2];
    // Integrator for variable x1
    intg[0] = new TwoVarInteg(1.0,0,0.1,x);
    // Integrator for variable x2
    intg[1] = new TwoVarInteg(1.0,1,0.1,x);
    // Connect the output of x2 to the input of x1
    staticDigraph g;
    g.couple(intg[1],1,intg[0],0);
    // Run the simulation for 3.361 units of time
    devssim sim(&g);
    while (sim.timeNext() <= 3.4) {
        cout << "t = " << sim.timeLast() << endl;
        for (int i = 0 ; i < 2; i++) {
            intg[i]→printState();
        }
        sim.execNextEvent();
    }
    // Done
    return 0;
}

```

Figure 7: Main simulation code for the two equation simulator.

t	q_1	\dot{q}_1	y_1	ta	event type	q_2	\dot{q}_2	y_2	ta_2
0	1	-0.5	1	0.2	init	1	-0.1	1	1
0.2	0.9	-0.4	0.9	0.25	internal				
0.45	0.8	-0.3	0.8	0.3333	internal				
0.7833	0.7	-0.2	0.7	0.5	internal				
1	0.6567	-0.25		0.5733	external	0.9	-0.09	0.9	1.111
1.573	0.6	-0.15	0.6	0.6667	internal				
2.111	0.5193	-0.2		0.9033	external	0.8	-0.08	0.8	1.25
3.014	0.5	-0.1	0.5	1	internal				

Table 3: DEVS simulation of two coupled ordinary differential equations.

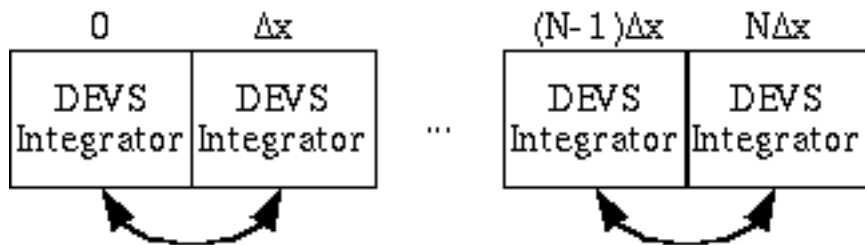


Figure 8: A cellspace view of the system described by equation 13.

5 The heat equation

In many instances, discrete approximations of partial differential equations can be obtained by a two step process. In the first step, a discrete approximation of the spatial derivatives is constructed. This results in a large set of coupled ordinary differential equations. The second step approximates the remaining time derivatives. This step can be accomplished with the discrete event integration scheme.

To illustrate this process, consider the heat (or diffusion) equation

$$\frac{\partial u(t, x)}{\partial t} = -\frac{\partial^2 u(t, x)}{\partial x^2}. \quad (11)$$

The function $u(t, x)$ represents the quantity that becomes diffuse (temperature if this is the heat equation). The spatial derivative can be approximated with a center difference, this giving

$$\frac{\partial^2 u(t, k\Delta x)}{\partial x^2} \approx \frac{u(t, (k+1)\Delta x) - 2u(t, k\Delta x) + u(t, (k-1)\Delta x)}{\Delta x^2}, \quad (12)$$

where Δx is the resolution of the spatial approximation, and the k are indices on the discrete spatial grid. Substituting 12 into 11 gives a set of coupled ordinary differential equations

$$\frac{du(t, k\Delta x)}{dt} = -\frac{u(t, (k+1)\Delta x) - 2u(t, k\Delta x) + u(t, (k-1)\Delta x)}{\Delta x^2} \quad (13)$$

that can be simulated using the DEVS integration scheme. This difference equation describes a grid of N integrators, and each integrator is connected to its two neighbors. The integrators at the end can be given fixed left and right values (i.e., fixing $u(t, -1\Delta x)$ and $u(t, (N+1)\Delta x)$) equal to a constant), or some other suitable boundary condition can be used. For the sake of illustration, let $u(t, -1\Delta x) = u(t, (N+1)\Delta x) = 0$. With these boundary conditions, two equivalent views of the system can be constructed. The first view, show in equation 14, utilizes a matrix to describe the coupling of the differential equations in 13.

$$\frac{d}{dt} \begin{bmatrix} u(t, 0) \\ u(t, \Delta x) \\ u(t, 2\Delta x) \\ \dots \\ u(t, (N-1)\Delta x) \\ u(t, N\Delta x) \end{bmatrix} = \frac{1}{\Delta x} \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & \dots & \dots & \dots & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u(t, 0) \\ u(t, \Delta x) \\ u(t, 2\Delta x) \\ \dots \\ u(t, (N-1)\Delta x) \\ u(t, N\Delta x) \end{bmatrix} \quad (14)$$

Because the k th equation is directly influenced only by the $(k+1)$ st and $(k-1)$ st equations, it is also possible to represent equations 13 as a cell space in which each cell is influenced by its left and right neighbors. The discrete event model favors this representation. The discrete event cellspace, which is illustrated in figure 8, has an integrator at each cell, and the integrator receives input from its left and right neighbors. Figure 9 shows the adevs simulation code for equation 13. The cellspace view of the equation coupling is implemented using the adevs Cellspace class.

The discrete event approximation to equation 13 has two potential advantages over a similar discrete time approximation. The discrete time approximation is obtained from the same approximation to the spatial

```

class DiffInteg: public Integrator, public cell_interface {
public:
    DiffInteg(double q0, int index, double D, double* x, double dx):
        Integrator(q0,index,D,x),cell_interface(){ dx2=dx*dx; }
    double f(int index, const double* x) {
        return (x[index-1]-2.0*x[index]+x[index+1])/dx2;
    }
private:
    static double dx2;
};
double DiffInteg::dx2 = 0.0;

void print(const double* x, double dx, int dim, double t) {
    for (int i = 0; i < dim; i++) {
        double soln = 100.0*sin(M_PI*i*dx/80.0)*exp(-t*M_PI*M_PI/6400.0);
        cout << i*dx << " " << x[i] << " " << fabs(x[i]-soln) << endl;
    }
}

int main() {
    // Build the solution array and assign boundary and initial values
    double len = 80.0;
    double dx = 0.1;
    int dim = len/dx;
    double* x = new double[dim+2];
    // Half sine intial conditions with zero at boundaries
    for (int i = 0; i <= dim; i++) {
        x[i] = 100.0*sin(M_PI*i*dx/80.0);
    }
    x[0] = x[dim+1] = 0.0;
    // Create the DEVs model
    double D = 10.0;
    cellSpace cs(cellSpace::SIX_POINT,dim);
    for (int i = 1; i <= dim; i++) {
        cs.add(new DiffInteg(x[i],i,D,x,dx),i-1);
    }
    // Run the model
    devssim sim(&cs);
    sim.run(300.0);
    print(x,dx,dim+2,sim.timeLast());
    // Done
    delete [ ] x;
    return 0;
}

```

Figure 9: Code listing for the heat equation solver.

derivatives, but using the explicit Euler integration scheme to approximate the time derivatives (see, e.g., [18]). Doing this gives a set of coupled difference equations

$$u(t + \Delta t, k\Delta x) = u(t, k\Delta x) + \Delta t \left(\frac{u(t, (k + 1)\Delta x) - 2u(t, k\Delta x) + u(t, (k - 1)\Delta x)}{\Delta x^2} \right).$$

This discrete time integration scheme has an error term that is proportional to the time step Δt . In this respect, it is similar to the discrete event scheme whose approximation to the time derivative is proportional to the quantum size D . However, there is an extra constraint in the discrete time formulation that is not present in the discrete event approximation. This extra constraint is a stability condition on the set of difference equations (not the differential equations, which are inherently stable). For a stable simulation (i.e., for the state variables to decay rather than explode), it is necessary that

$$\Delta t \leq \frac{\Delta x^2}{2}.$$

Freedom from the stability constraint is a significant advantage that the discrete event scheme has over the discrete time scheme. For discrete time systems, this stability constraint can only be removed by employing implicit approximations to the time derivative. Unfortunately, this introduces a significant new computational overhead because a system of equations in the form $Ax = b$ must be solved at each integration step (see, e.g., [18]).

The unconditional stability of the discrete event scheme can be demonstrated with a calculation. Consider a heat conducting bar with length 80. The ends are fixed at a temperature of 0. The initial temperature of the bar is given by $u(0, x) = 100\sin(\pi x/80)$. Figures 10a and 10b show the computed solution at $t = 300$ using $\Delta x = 0.1$ and different values of D . Even with large values of D , it can be seen that the computed solution remains bounded. Figure 10c shows the error in the computed solution for the more reasonable choices of D . From the figure, the correspondence between a reduction in D and a reduction in the computational error is readily apparent.

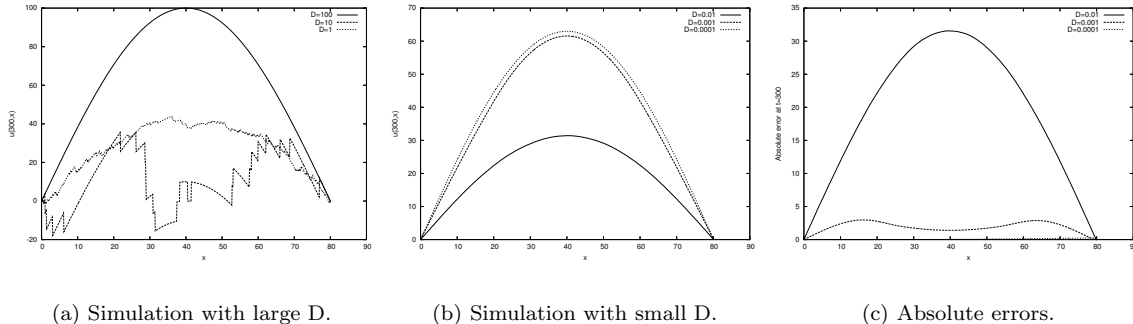


Figure 10: DEVS simulation of the heat equation with various quantum sizes.

In many instances, the discrete event approximation enjoys a computational advantage as well. An in depth study of the relative advantage of a DEVS approximation to the heat equation over a discrete time approximation is described in [5] and [23]. This advantage is realized in the forest fire simulation described by [12], where a diffusive process is the spatially explicit piece of the forest fire model. In that report, the DEVS approximation is roughly four times faster than explicit discrete time simulation giving the same errors with respect to experimental data.

The reason for the performance advantage can be understood intuitively in two related ways. The first is to observe that the time advance function determines the frequency with which state updates are calculated at a cell. The time advance at each cell is inversely proportional to the magnitude of the derivative, and so cells that are changing slowly will have large time advances relative to cells that are changing quickly. This causes the simulation algorithm to focus effort on the changing portion of the solution, with significantly less work being devoted to portions that are changing slowly. This is demonstrated in figure 11. The state

transition frequency at a point is given by the inverse of the time advance function following an internal event (i.e., $\dot{u}(i\Delta x, t)/D$, where i is the grid point index). Figure 11a shows the state transition frequency at the beginning and end of the simulation. Figure 11b shows the total number of state changes the are computed at each grid point over the course of the calculation. It can be seen that the computational effort is focused on the center of the bar, where the state transition functions are evaluated most frequently.

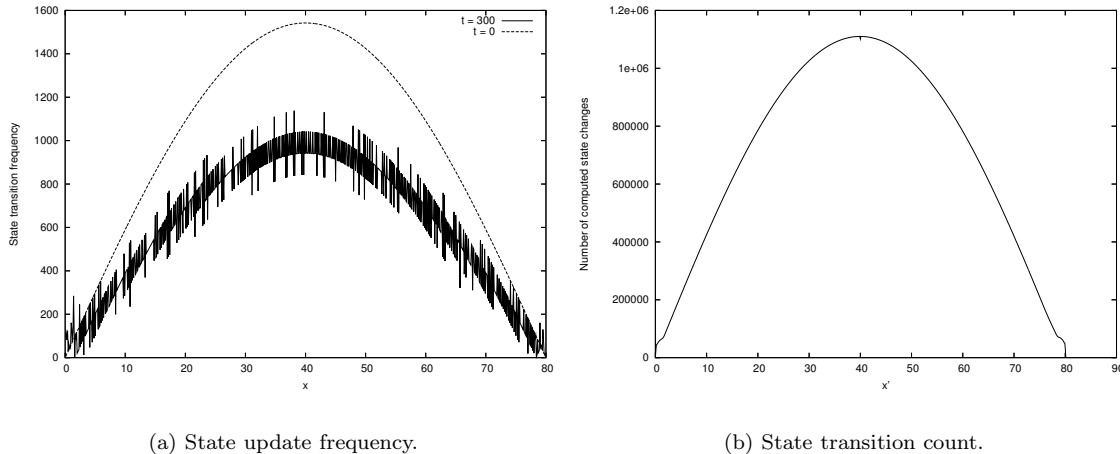


Figure 11: Activity tracking in the DEVS diffusion simulation using $D = 0.0001$.

A second explanation can be had by observing that the number of quantum crossings required for the solution at a grid point to move from its initial to final state is, approximately, equal to the distance between those two states divided by the quantum size. This gives a lower bound on the number of state transitions that are required to move from one state to another. It can be shown that, in many instances, the number of state transitions required by the DEVS model will closely approximate this ideal number (see [5]).

6 Conservation laws

Conservation laws are an important application area where DEVS approximations of the time derivatives can be usefully applied. A DEVS simulation of Euler’s fluid equations is presented in [16]. In that report, a significant performance advantage was obtained, relative to a similar time stepping method, via the activity tracking property described above. In this section, the application of DEVS to conservation laws is demonstrated for a simpler problem, where it is easier to focus on the derivation of the discrete event model.

A conservation law in one special dimension is described by a partial differential equation

$$\frac{\partial u(t, x)}{\partial t} + \frac{\partial F(u(t, x))}{\partial x} = 0.$$

The flux function $F(u(t, x))$ describes the rate of change in the amount of u (whatever u might represent) at each point x (see, e.g., [18]). To be concrete, consider the conservation law

$$\frac{\partial u(t, x)}{\partial t} + u(t, x) \frac{\partial u(t, x)}{\partial x} = 0. \tag{15}$$

Equation 15 describes a material with quantity $u(t, x)$ that moves with velocity $u(t, x)$. In this equation, the flux function is $u(t, x)^2/2$. Equation 15 is obtained by taking the partial with respect to x of this flux function.

As before, the first step is to construct a set of coupled ordinary differential equations that approximates the partial differential equation. There are numerous schemes for approximating the derivative of the flux

```

class ClawInteg: public Integrator, public cell_interface {
public:
    ClawInteg(double q0, int index, double D, double* x, double dx):
        Integrator(q0,index,D,x),cell_interface(){ ClawInteg::dx=dx; }
    double f(int index, const double* x) {
        return 0.5*(x[index-1]*x[index-1]-x[index]*x[index])/dx;
    }
private:
    static double dx;
};
double ClawInteg::dx = 0.0;

```

Figure 12: Integrator for the conservation law solver.

function with respect to x (see, e.g., [9]). One of the simplest is an upwinding scheme on a spatial grid with resolution Δx . Applying an upwinding scheme to 15 gives

$$u(t, k\Delta x) \frac{\partial u(t, k\Delta x)}{\partial x} \approx -\frac{1}{2\Delta x} (u(t, (k-1)\Delta x)^2 - u(t, k\Delta x)^2). \quad (16)$$

Substituting 16 into 15 gives the set of coupled ordinary differential equations

$$\frac{du(t, k\Delta x)}{dt} = \frac{1}{2\Delta x} (u(t, (k-1)\Delta x)^2 - u(t, k\Delta x)^2). \quad (17)$$

It is common to approximate the time derivatives in equation 17 with the explicit Euler integration scheme using a time step Δt . This gives the set of difference equations

$$u(t + \Delta t, k\Delta x) = u(t, k\Delta x) + \frac{\Delta t}{2\Delta x} (u(t, (k-1)\Delta x)^2 - u(t, k\Delta x)^2)$$

that approximates the set of differential equations. The difference equations are stable provided that the condition

$$\frac{\Delta t}{\Delta x} \max |u(i\Delta t, j\Delta x)| \leq 1$$

is satisfied at every time point i and every spatial point j . Because equations 17 are nonlinear, it is not necessarily true that a discrete event approximation will be stable regardless of the size of the integration quantum. However, it is possible to find a sufficiently small quantum for which the scheme works (see [13]). This remains an open area of research, but we will move recklessly ahead and try generating solutions with several different quantum sizes and observe the effect on the solution.

For this example, a space 10 units in length is assigned the initial conditions

$$u(0, x) = \begin{cases} \sin(\pi x/4) & \text{if } 0 \leq x \leq 4 \\ 0 & \text{otherwise} \end{cases}$$

and the boundary conditions $u(t, 0) = u(t, 10) = 0$. The integrator implementation for this model is shown in figure 12. The simulation main routine is identical to the one for the heat equation (except where DiffInteg is replaced by ClawInteg; see figure 9). Figure 13 shows snapshots of the solution computed with $\Delta x = 0.1$ and three different quantum sizes; 0.1, 0.01, and 0.001. The computed solutions maintain important features of the real solution, included the shock formation and shock velocity (see [18]).

While the advantage of the discrete event scheme with respect to stability remains unresolved (but look promising!), a potential computational advantage can be seen. From the figure, it is apparent that the larger derivatives follow the shock, with the area in front of the shock having zero derivatives and the area behind the shock having diminishing derivatives. The DEVS simulation apportions computational effort appropriately. This is shown in figure 14 for the simulation with $D = 0.001$. Figure 14a shows several

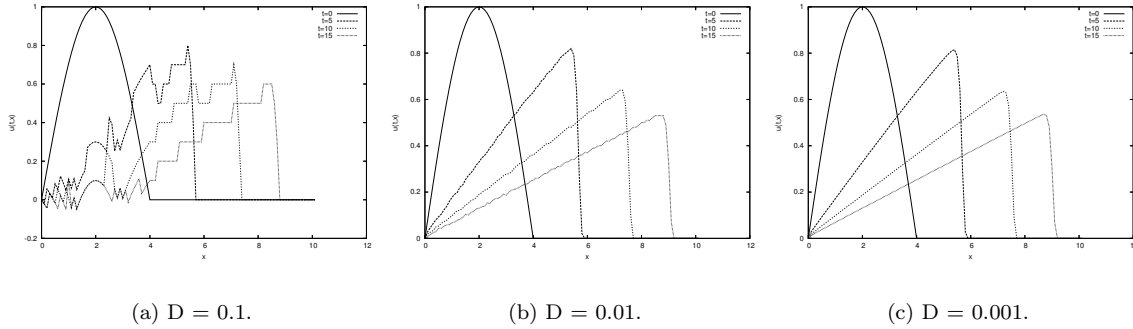


Figure 13: Simulation of equation 17 with various quantum sizes.

snapshots of the cell update frequency (i.e., $\dot{u}(i\Delta x, t)/D$ following an internal event, where i is the grid point index) at times corresponding to the solution snapshots shown in figure 13c. Figure 14b shows the total number of state transitions computed at each cell at those times. The effect of this front tracking behavior can be significant. In [16] it is responsible for a speedup of 35 relative to a discrete time solution for Euler's equations in one spatial dimension.

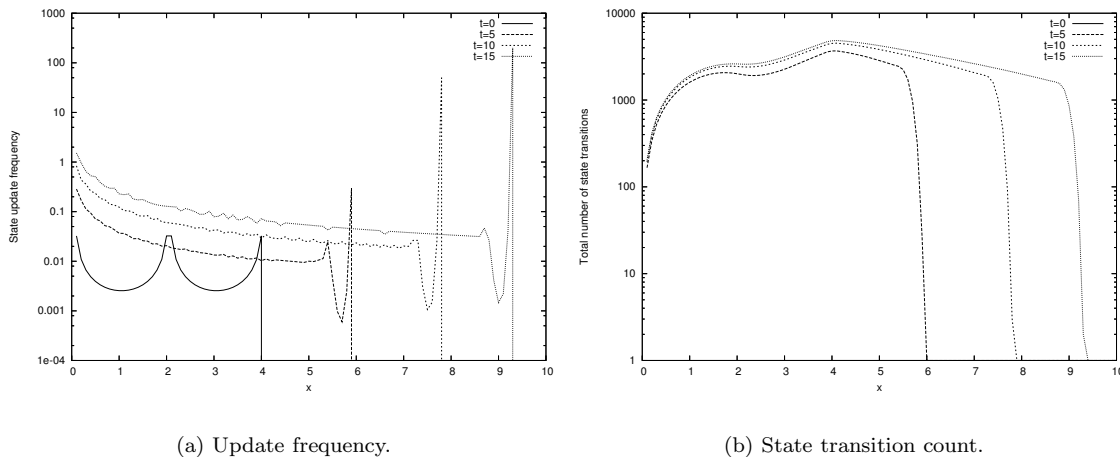


Figure 14: Front tracking in the DEVS simulation of equation 17 with $D = 0.001$.

7 Two point integration schemes

The integration scheme discussed to this point is a single point scheme. It relies on a single past value of the function, and it is exact for the linear equation $\dot{x}(t) = k$, where k is a constant. Recall that the single point scheme for simulating a system described as $\dot{x}(t) = f(x(t))$ can be derived from the expression

$$\left| \int_{t_0}^{t_0+h} f(x(t)) dt \right| = D \quad (18)$$

by approximating $f(x(t))$ with the value $f(x(t_0))$.

If the function $f(x(t))$ in equation 18 is approximated using the previous two values of the derivative, then the resulting method is called a two point scheme. A DEVS model of a two point scheme requires the state variables

- q , the current approximation to $x(t)$,
- q_l , the last grid point occupied by q ,
- σ , the time required to move from q to the next grid point,
- \dot{q}_1 and \dot{q}_0 , the last two computed values of the derivative, and, possibly,
- h , the time interval between \dot{q}_1 and \dot{q}_0 .

At least two different two point methods have been described (see [8] and [14]). The first method approximates $f(x(t))$ in equation 18 with the line connecting points \dot{q}_1 and \dot{q}_0 . The distance moved by $x(t)$ in the interval $[h, h + T]$ can be approximated by

$$\int_h^{h+T} \frac{\dot{q}_1 - \dot{q}_0}{h} + \dot{q}_0 dt = \frac{\dot{q}_1 - \dot{q}_0}{2h} T^2 + \dot{q}_1 T = \Delta q.$$

The functions

$$\hat{F}_1(T, \dot{q}_1, \dot{q}_0, h) = \frac{\dot{q}_1 - \dot{q}_0}{2h} T^2 + \dot{q}_1 T, \quad (19)$$

and

$$\hat{F}_1^{-1}(\Delta q, \dot{q}_1, \dot{q}_0, h) = \Delta T, \quad (20)$$

where ΔT is the smallest positive root of

$$\left| \frac{\dot{q}_1 - \dot{q}_0}{2h} T^2 + \dot{q}_1 T \right| = \Delta q$$

and ∞ if such a root does not exist, can be used to define the state transition, output, and time advance functions (which will be done in a moment). Equations 19 and 20 are exact when $x(t)$ is a quadratic.

The second method approximates $f(x(t))$ with the piecewise constant function

$$a\dot{q}_1 + b\dot{q}_0, \quad a + b = 1. \quad (21)$$

If $x(t)$ is the line $mt + b$, then $f(x(t)) = m$, $(am + bm) = (a + b)m = m$, and so this approximation is exact. Integrating equation 21 over the interval $[0, T]$ gives the approximating functions

$$\hat{F}_2(T, \dot{q}_1, \dot{q}_0) = (a\dot{q}_1 + b\dot{q}_0)T, \text{ and} \quad (22)$$

$$\hat{F}_2^{-1}(\Delta q, \dot{q}_1, \dot{q}_0) = \frac{\Delta q}{|a\dot{q}_1 + b\dot{q}_0|}. \quad (23)$$

This approximation does not require the state variable h .

For brevity, let \bar{q} denote the state of the integrator, and let $\bar{d}q$ denote the variables \dot{q}_1, \dot{q}_0 or \dot{q}_1, \dot{q}_0, h as needed. Which is intended will be clear from the context in which it is used. The time advance function for a two point scheme is given by

$$ta(\bar{q}) = \sigma,$$

and the output function is defined by

$$\lambda(\bar{q}) = \hat{F}(\sigma, \bar{d}q).$$

If equations 19 and 20 are used to define the integration scheme, then the resulting state transition functions are

$$\begin{aligned} \delta_{int}(\bar{q}) &= (q + \hat{F}_1(\sigma, \bar{d}q), q + \hat{F}_1(\sigma, \bar{d}q), q_1, q_1, \sigma, \\ &\quad \hat{F}_1^{-1}(D, \dot{q}_1, \dot{q}_1, \sigma)), \\ \delta_{ext}(\bar{q}, e, x) &= (q_l, q + \hat{F}_1(e, \bar{d}q), x, q_1, e, \\ &\quad \hat{F}_1^{-1}(D - |q + \hat{F}_1(e, \bar{d}q) - q_l|, x, \dot{q}_1, e)), \text{ and} \\ \delta_{con}(\bar{q}, x) &= (q + \hat{F}_1(\sigma, \bar{d}q), q + \hat{F}_1(\sigma, \bar{d}q), x, q_1, \sigma, \\ &\quad \hat{F}_1^{-1}(D, x, \dot{q}_1, \sigma)). \end{aligned}$$

When equations 22 and 23 are used to define the integrator, then the state transition functions are

$$\begin{aligned}\delta_{int}(\bar{q}) &= (q + \hat{F}_2(\sigma, \bar{d}q), q + \hat{F}_2(\sigma, \bar{d}q), q_1, q_1, \\ &\quad \hat{F}_2^{-1}(D, \dot{q}_1, \dot{q}_1)), \\ \delta_{ext}(\bar{q}, e, x) &= (q_l, q + \hat{F}_2(e, \bar{d}q), x, q_1, \\ &\quad \hat{F}_2^{-1}(|q + \hat{F}_2(e, \bar{d}q) - q_l| - D, x, \dot{q}_1)), \text{ and} \\ \delta_{con}(\bar{q}, x) &= (q + \hat{F}_2(\sigma, \bar{d}q), q + \hat{F}_2(\sigma, \bar{d}q), x, q_1, \\ &\quad \hat{F}_2^{-1}(D, x, \dot{q}_1)).\end{aligned}$$

The scheme that is constructed using equations 19 and 20 is similar to the QSS2 method in [8], except that the input and output trajectories used here are piecewise constant rather than piecewise linear.

The scheme constructed from equations 22 and 23 is nearly second order accurate when a and b are chosen correctly. If $a = \frac{3}{2}$ and $b = -\frac{1}{2}$, then the error in the integral of 21 is

$$E = (f(x_1) - \frac{3f(x_1)}{2} + \frac{f(x_0)}{2})T + \frac{1}{2}T^2 \frac{d}{dt}f(x_1) + \sum_{n=3}^{\infty} \frac{1}{n!} \frac{d^{(n+1)}}{dt} f(x_1)T^n. \quad (24)$$

For this scheme to be nearly second order accurate, the terms that depend on T and T² need to be as small as possible. Let h be the time separating x_1 and x_0 (i.e., $x_1 = x(t_1)$ and $x_0 = x(t_0)$ and $h = t_1 - t_0$), and let $\alpha = \frac{T}{h}$, the ratio of the current time advance to the previous time advance. It follows that $T = \alpha h$. The function $\frac{d}{dt}f(x_1)$ can be approximated by

$$\frac{d}{dt}f(x_1) \approx \frac{f(x_1) - f(x_0)}{h}. \quad (25)$$

Substituting equation 25 into equation 24 and dropping the high order error terms gives

$$E \approx \alpha h \left(\frac{f(x_1) - f(x_0)}{2} + \alpha \frac{f(x_0) - f(x_1)}{2} \right). \quad (26)$$

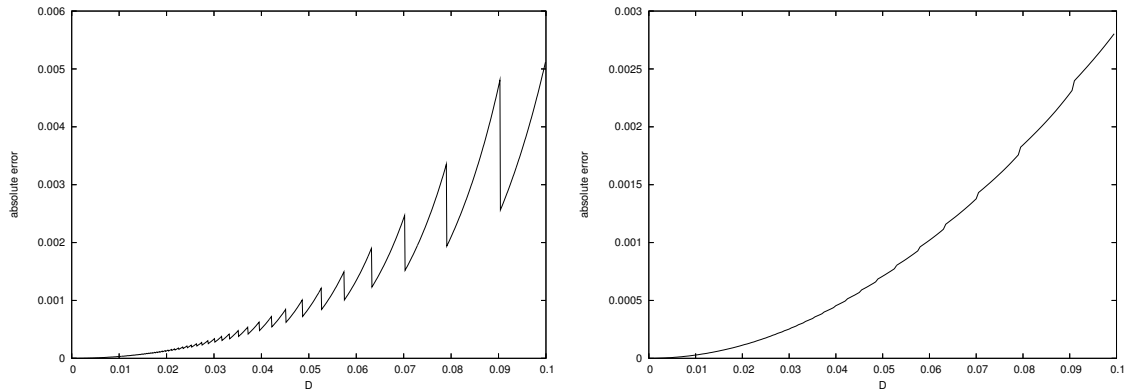
Equation 26 approaches zero as α approaches 1. It seems reasonable to assume T and h become increasingly similar as D is made smaller. From this assumption, it follows that the low order error terms in equation 24 vanish as D shrinks.

Figures 15a and 15b show the absolute error in the computed solution of $\dot{x}(t) = -x(t)$, $x(0) = 1$, as a function of D for these two integration schemes. The simulation is ended at $t = 1.0$, and α and the absolute error are recorded at that time. In both cases, it can be observed that the absolute error is proportional to D².

These two schemes use additional information to reduce the approximation error with respect to the single point scheme. Fortunately, these two schemes share the unconditional linear stability of the single point scheme (see [8] and [13]), and so they represent a trade off between storage, execution time, and accuracy. When dealing with very large systems, the single point scheme has the advantage of needing less computer memory because it has fewer state variables per integrator. However, it will, in general, be less accurate than a two point scheme for a given quantum size. Moreover, if the quantum size is selected to obtain a given error, then the two point scheme will generally use a larger quantum than the one point scheme, and so the simulation will finish more quickly using the two point scheme.

8 Conclusions

This chapter introduced some essential techniques for constructing discrete event approximations to continuous systems. Discrete event simulation of continuous systems is an active area of research, and the breadth of the field can not be adequately covered in this short space. So, in the conclusion, some recent results are summarized and references given for the interested reader.



(a) Simulation error using equations 19 and 20.

(b) Simulation error using equations 22 and 23.

Figure 15: Simulation error as a function of D for the system $\dot{x}(t) = -x(t)$ with $x(0) = 1$.

In [1], an adaptive quantum scheme is introduced. This scheme allows the integration quantum to be varied during the course of the calculation in order to maintain an upper bound on the global error. An application of adaptive quantization to a fire spreading model is discussed in [11].

A methodology for approximating general time functions as DEVS models is discussed in [4]. The approximations introduced in that paper associate events with changes in the coefficients of an interpolating polynomial. An application of this methodology to partial differential equations is shown in [21].

Applying DEVS models to finite element method for equilibrium problems is discussed in [2] and [17]. A steady state heat transfer problem is used to demonstrate the method.

Simulation of partial differential equations leads naturally to parallel computing. Parallel discrete event simulation for the numerical methods presented in this paper are discussed in [13] and [15]. Specific issues that emerge when simulating DEVS models using logical-process based algorithms are described in [15]. Parallel discrete event simulation applied to particle in cell methods is discussed in [20] and [6].

References

- [1] Jean-Sébastien Bolduc and Hans Vangheluwe. Mapping odes to devs: Adaptive quantization. In *Proceedings of the 2003 Summer Simulation MultiConference (SCSC'03)*, pages 401–407, Montréal, Canada, July 2003.
- [2] M. D’Abreu and G. Wainer. Improving finite elements method models using cell-devs. In *Proceedings of the 2003 Summer Computer Simulation Conference*, Montreal, QC, Canada, 2003.
- [3] Jean-Baptiste Filippi and Paul Bisgambiglia. Jdevs: an implementation of a devs based formal framework for environmental modelling. *Environmental Modelling & Software*, 19(3):261–274, March 2004.
- [4] Norbert Giambiasi, Bruno Escude, and Sumit Ghosh. Gdevs: A generalized discrete event specification for accurate modeling of dynamic systems. *Trans. Soc. Comput. Simul. Int.*, 17(3):120–134, 2000.
- [5] R. Jammalamadaka. Activity characterization of spatial models: Application to the discrete event solution of partial differential equations. Master’s thesis, University of Arizona, Tucson, Arizona, USA, 2003.
- [6] H. Karimabadi, J. Driscoll, Y.A. Omelchenko, and N. Omid. A new asynchronous methodology for modeling of physical systems: breaking the curse of courant condition. *Journal of Computational Physics*, 205(2):755–775, May 2005.

- [7] E. Kofman, M. Lapadula, and E. Pagliero. Powerdevs: A devs-based environment for hybrid system modeling and simulation. Technical Report LSD0306, School of Electronic Engineering, Universidad Nacional de Rosario, Rosario, Argentina, 2003.
- [8] Ernesto Kofman. Discrete event simulation of hybrid systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.
- [9] Dietmar Kroner. *Numerical Schemes for Conservation Laws*. Wiley, Chichester, New York, 1997.
- [10] A. Muzy and J. Nutaro. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. In *1st Open International Conference on Modeling & Simulation*, pages 401–407, ISIMA / Blaise Pascal University, France, June 2005.
- [11] Alexandre Muzy, Eric Innocenti, Antoine Aiello, Jean-Francois Santucci, and Gabriel Wainer. Cell-devs quantization techniques in a fire spreading application. In *Proceedings of the 2002 Winter Simulation Conference*, 2002.
- [12] Alexandre Muzy, Paul-Antoine Santoni, Bernard P. Zeigler, James J. Nutaro, and Rajanikanth Jammalamadaka. Discrete event simulation of large-scale spatial continuous systems. In *Simulation Multi-conference*, 2005.
- [13] James Nutaro. *Parallel Discrete Event Simulation with Application to Continuous Systems*. PhD thesis, University of Arizona, Tuscon, Arizona, 2003.
- [14] James Nutaro. Constructing multi-point discrete event integration schemes. In *Proceedings of the 2005 Winter Simulation Conference*, 2005.
- [15] James Nutaro and Hessam Sarjoughian. Design of distributed simulation environments: A unified system-theoretic and logical processes approach. *SIMULATION*, 80(11):577–589, 2004.
- [16] James J. Nutaro, Bernard P. Zeigler, Rajanikanth Jammalamadaka, and Salil R. Akerkar. Discrete event solution of gas dynamics within the devs framework. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, *International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 319–328. Springer, 2003.
- [17] H. Saadawi and G. Wainer. Modeling complex physical systems using 2d finite elemental cell-devs. In *Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04)*, Arlington, VA, U.S.A., 2004.
- [18] Gilber Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, Massachusetts, 1986.
- [19] Ferenc Szidarovszky and A. Terry Bahill. *Linear Systems Theory, Second Edition*. CRC Press LLC, Boca Raton, Florida, 1998.
- [20] Yarong Tang, Kalyan Perumalla, Richard Fujimoto, Homa Karimabadi, Jonathan Driscoll, and Yuri Omelchenko. Parallel discrete event simulations of physical systems using reverse computation. In *ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Monterey, CA, June 2005.
- [21] Gabriel A. Wainer and Norbert Giambiasi. Cell-devs/gdevs for complex continuous systems. *SIMULATION*, 81(2):137–151, February 2005.
- [22] Gabriel Wainer. Cd++: a toolkit to develop devs models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.
- [23] Bernard P. Zeigler. Continuity and change (activity) are fundamentally related in devs simulation of continuous systems. In *Keynote Talk at AI, Simulation, and Planning 2004 (AIS'04)*, October 2004.

- [24] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation, 2nd Edition*. Academic Press, 2000.
- [25] Bernard P. Zeigler, Hessam Sarjoughian, and Herbert Praehofer. Theory of quantized systems: Devs simulation of perceiving agents. *Cybernetics and Systems*, 31(6):611–647, September 2000.
- [26] Bernard P. Zeigler and Hessam S. Sarjoughian. Introduction to devs modeling and simulation with java: Developing component-based simulation models. Unpublished manuscript, 2005.