Computer Simulation of Stringed Instruments

Charles S. Peskin
Courant Institute of Mathematical Sciences, New York University

1 Pitch, Intervals, and Frequency

Musicians describe a note by its *pitch*, and say that two notes of different pitch are separated by a certain *interval*. The names for intervals are somewhat confusing, but they can be made systematic by counting the number of *half-steps* to get from one note to the other. A half-step is the interval between adjacent keys on a piano, including both the black and the white keys. An *octave*, for example, is equal to 12 half-steps, and when two notes are separated by an octave, they are given the same name. The reason for this is that when the two notes are played together they sound almost like a single note.

The physical attribute of a sound wave that determines its pitch is its *frequency*, i.e., the number of cycles per unit time. The relationship between pitch and frequency is surprising, however. It turns out that a given musical interval corresponds to a particular *ratio* of frequencies. An octave, for example, corresponds to a ratio of 2:1 in frequency. Since there are 12 half-steps in an octave, and all of them are equal as musical intervals, it must be the case that a musical interval called a half-step corresponds to a ratio of frequencies of $2^{1/12} : 1$.

The number $2^{1/12}$ is not a rational number, that is, it cannot be expressed as a ratio of integers. In fact, all of the numbers $2^{m/12}$, for any integer m that is not a multiple of 12, is irrational. (You might enjoy trying to prove these facts. To get started, recall the proof that $\sqrt{2}$ is irrational, which you may have learned in high school. This is a special case of the foregoing, with $m = 6$.) Thus, all standard musical intervals (other than intervals between notes that have the same name) involve irrational frequency ratios.

This is a very strange state of affairs, since music is supposed to be based on harmony, and harmony is supposed to be defined by rational frequency ratios. To see what is going on here, consider the interval from C to G, which musicians call a “fifth”. This name comes from the fact that C and G are the first and last notes of the 5-note sequence CDEFG of the major scale. From a mathematical point of view, this is not a sensible way to name intervals, for two reasons: The sequence involves 4 steps, not 5, and they are not equal steps anyway, since EF is a half-step, and the others are whole steps. A more sensible way to describe this interval would be that it involves 7 half-steps, and this implies a frequency ratio of $2^{7/12} = 1.4983$, which is *very* close to the rational number 3/2. Next to the octave, this is the interval that sounds most "simple" when the two notes are played together. The next simplest-sounding interval is the "third" which is from C to E. This involve 4 half-steps and a frequency ratio of $2^{4/12} = 1.2599$, which is close to the rational number 5/4. These examples suggest that division of the octave into 12 equal intervals (meaning equal frequency ratios) works because it produces some intervals, at least, with frequency ratios that are *almost* equal to ratios of small integers.

What we have described above is the *well-tempered* scale, associated especially with Bach, in which every half-step has the same frequency ratio, and therefore intervals other than the octave are slightly out-of-tune from a strictly harmonic point of view. The benefit of the well-tempered approach is that an instrument can play in any key.

2 The Equation of a Vibrating String

We consider a string of length L with fixed ends at $x = 0$ and $x = L$. The string is under a constant tension T, and its mass per unit length is M. We imagine the string vibrating in a vertical plane. At rest, the string lies on the x axis, and we describe its vertical displacement from rest by the function $H(x,t)$. The equation of motion of the string, for small dispacements, is the wave equation:

$$M\frac{\partial^2 H}{\partial t^2} = T\frac{\partial^2 H}{\partial x^2}. \tag{1}$$

On the left-hand side of this equation we have mass per unit length $\times$ acceleration, i.e., force per unit length, and on the right-hand side we have tension $\times$ curvature. Since tension is force, and curvature has units of 1/length, the units check.

Before discussing the numerical solution of this equation, we describe two kinds of solutions that provide important insight concerning the behavior of such a string. First, we ignore the boundary conditions, and consider an infinite string that obeys equation (1). We look for a solution in the form of a traveling wave:

$$H(x,t) = h(x - ct). \tag{2}$$

Note that h is a function of one variable instead of two, so this is a restriction on the form of $H(x,t)$. Such a solution is called a *traveling wave* because the spatial profile of the wave is the same at every instant of time, except for a shift in the x direction by the amount ct. An exercise for the reader is to substitute this assumed form of H into equation (1) and to show that the form of the function h does not matter, but that the wave velocity c must be given by

$$c = \pm\sqrt{\frac{T}{M}}. \tag{3}$$

In summary, traveling wave solutions of the wave equation can be any shape, and they can travel in either direction, but always at the speed $\sqrt{\frac{T}{M}}$. (Check the units. They might seem wrong if you think that M has units of mass. It actually has units of mass/length.)

The second type of solution that we consider takes into account that the ends of the string are fixed at $x = 0$ and $x = L$, where we require

$$H(0, t) = H(L, t) = 0. \tag{4}$$

To find such solutions, we try the form

$$H(x, t) = A_k(t)\sin(k\pi x/L), \quad k = 1, 2, \ldots . \tag{5}$$

If this works, we will have infinitely many solutions, one for each positive integer k. (The case $k = 0$ is not interesting, since it only gives $H = 0$, and negative integers do not need to be considered since the case $-k$ and the case k are essentially the same.) Solutions of the form (4) are called *standing waves*, since their spatial shape remains fixed over time, and the only thing that changes with time is the amplitude, $A_k(t)$, which we have not yet determined.

When $H(x, t)$ is given by equation (5), it automatically satisfies the boundary conditions (4), but we do not yet know whether this form of $H(x, t)$ can satisfy the wave equation. We leave it as an exercise for the reader to substitute equation (5) into equation (1) and to show that a solution of (1) is obtained if and only if $A_k(t)$ satisfies the ordinary differential equation

$$\frac{d^2A_k}{dt^2} = -\frac{T}{M}\left(\frac{k\pi}{L}\right)^2 A_k. \tag{6}$$

This differential equation is solved by

$$A_k(t) = A\sin\left(\sqrt{\frac{T}{M}}\frac{k\pi}{L}t\right), \tag{7}$$

in which A is an arbitrary constant. Note that we could equally well have used the cosine function here, instead of the sine, or more generally any linear combination of the two.

The coefficient of t in the above expression is called the angular frequency. It is the number of radians per unit time of the oscillation, and is often denoted by the symbol ω. In music, however, it is more common to describe the frequency of a sound in cycles (instead of radians) per unit time. To see what the frequency in this sense is, we multiply and divide by 2 and factor out π inside the argument of the sine function in equation (7). This gives

$$\begin{aligned} A_k(t) &= A\sin\left(2\pi\sqrt{\frac{T}{M}}\frac{k}{2L}t\right) \\ &= A\sin\left(2\pi f_k t\right), \end{aligned} \tag{8}$$

where

$$f_k = \sqrt{\frac{T}{M}}\frac{k}{2L} \tag{9}$$

is the frequency in cycles per unit time. To check this interpretation, note that $A_k(t)$ comes back to the same value every time that t advances by $1/f_k$. Thus $1/f_k$ is the duration of one cycle, and f_k is the number of cycles per unit time. The unit of cycles per second is called the Hertz, and is abbreviated Hz. Thus, if time is measured in seconds, f_k is the frequency in Hz.

Equation (9) is a formula for infinitely many different frequecies, all of which are possible frequencies at which the same string can vibrate. All of these frequencies are integer multiples of the *fundamental frequency* f_1, which is given by

$$f_1 = \sqrt{\frac{T}{M}}\frac{1}{2L} \tag{10}$$

The frequencies with $k > 1$ are called *harmonics*.

The period associated with the fundamental frequency is given by

$$\frac{1}{f_1} = \frac{2L}{\sqrt{T/M}} \tag{11}$$

This is equal to the time it takes to traverse a distance $2L$ while traveling at the speed $\sqrt{T/M}$, which is the wave speed that was found previously. The factor 2 may seem a bit mysterious. One way to think about it is to recall that the spatial form of the wave at the fundamental frequency involves only half a sine wave, i.e., the wavelength of this wave is $2L$ rather than L. Another way to think about it is that the right-hand side of equation (11) is the time required for the wave to make a round trip from one end of the string to other and back to the starting point.

If the vibrating string can support infinitely many standing waves, each with a different spatial profile and a different frequency of vibration, which one do we get when we pluck the string (as in a guitar) or hit it with a hammer (as in a piano)? The answer is: all of them!

This is because of two very fundamental concepts. The first of these is the *linearity* of the wave equation together with its fixed-end boundary conditions. If we are given any set of solutions to equation (1) that satisfies the boundary conditions (4), then we can multiply each of these solutions by an arbitrary constant and add the results, and this procedure will produce a function of x and t that is again a solution of the wave equation with the same boundary conditions. Thus, the standing waves described above can be combined in this manner to produce more general solutions. The second fundamental concept is Fourier analysis, from which we learn that essentially *any* function satisfying the boundary conditions (4) can be written as a sum of terms, each of which is of the form $c_k \sin(k\pi x/L)$ for some positive integer k. This is truly amazing, since the function in question may not look at all like a sine wave, and yet such a representation in terms of sine waves is possible (although an infinite number of terms may be required). Thus, no matter how we initiate the vibration of the string, what we will get can be expressed as a combination of the standing waves described above (but the amplitudes $A_k(t)$ may include cosine terms as well as sine terms). Indeed, this is one way to solve the wave equation, but we will not follow that route here. Instead, we will use finite difference methods.

3 A Numerical Method for the Wave Equation

We rewrite the wave equation as the following first order system:

$$\frac{\partial V}{\partial t} = \frac{T}{M}\frac{\partial^2 H}{\partial x^2}, \tag{12}$$

$$\frac{\partial H}{\partial t} = V. \tag{13}$$

Here, $V(x,t)$ is the velocity of the point of the string at location x and at time t. The first equation of the above pair is just equation (1) with the acceleration rewritten in terms of V instead of H, and the second equation is the definition of V.

To formulate a finite difference scheme for the above equations, we introduce a timestep Δt and a spatial step Δx, and we approximate equations (12-13) by the following:

$$\begin{aligned} \frac{V(x,t+\Delta t)-V(x,t)}{\Delta t} &= \frac{T}{M}\frac{H(x+\Delta x,t)-2H(x,t)+H(x-\Delta x,t)}{(\Delta x)^2}, \quad (14) \\ \frac{H(x,t+\Delta t)-H(x,t)}{\Delta t} &= V(x,t+\Delta t). \quad (15) \end{aligned}$$

The time discretization here is a slight modification of Euler's method, with the modification being the use of $V(x,t+\Delta t)$ on the right-hand side of equation (15), instead of $V(x,t)$.

To understand the approximation to $\partial^2 H/\partial x^2$ that is used on the right hand side of equation (14), note that it can be written so that it looks like the derivative of the derivative:

$$\frac{\dfrac{H(x+\Delta x)-H(x)}{\Delta x}-\dfrac{H(x)-H(x-\Delta x)}{\Delta x}}{\Delta x} \quad (16)$$

Here we have not written the time argument, since all of the interest is in the spatial variable.

Although we do not prove it here (or even define stability), this numerical method is stable if and only if the distance traveled in one timestep moving at the wave speed $\sqrt{T/M}$ is less than or equal to one spatial step. This gives the very important restriction that

$$\Delta t \leq \frac{\Delta x}{\sqrt{T/M}}. \quad (17)$$

The inequality (17) is called the Courant-Friedrichs-Lewy or CFL condition. Note in particular that a finer spatial mesh requires a smaller time step. If you want to see what numerical instability looks like, try violating the CFL condition, and see what happens!

4 Matlab Implementation

Let the variable x have discrete values

$$x_j = (j-1)\Delta x, \;\; j = 1\ldots J, \quad (18)$$

with

$$\Delta x = \frac{L}{J-1}. \tag{19}$$

Then $x_1 = 0$ and $x_J = L$, so the points with indices 1 and J are the fixed endpoints of the string, and the points with indices $j = 2 \ldots (J-1)$ are the interior points.

In the Matlab code that follows, the variables H and V at any given time will be stored in one-dimensional arrays with J elements, with the first and last element always equal to zero. Although those zeros are never updated, it is convenient to have them there, since they will be referenced when the neighboring points are updated, and of course they are also useful for plotting the configuration of the string.

The main loop of the vibrating string program now reads as follows:

```
j=2:(J-1); % make a list of the indices of interior points
for clock=1:clockmax
  t=clock*dt;
  V(j)=V(j)+(dt/dx^2)*(T/M)*(H(j+1)-2*H(j)+H(j-1));
  H(j)=H(j)+dt*V(j);
end
```

Note the use of the Matlab *subarray* notation in the forgoing. An index into an array can be any list of indices, and Matlab interprets this as a new array constructed from the original by pulling out the elements with indices in the list. (The list can even include repeats, although we do not use that feature here.) Thus, since `j` has been defined to be `2:J-1`, `H(j+1)` denotes the arrary with elements `H(3)...H(J)`, and `H(j-1)` denotes the array with elements `H(1)...H(J-2)`.

Up to now, we have been describing a vibrating string with no damping of any kind, so the vibration will last forever, and this will produced a sustained sound (rather like the sound of an organ) rather than the sound of a guitar or piano string. Various mechanisms of damping are possible, and we will not try to be too realistic here, but just provide one mechanism that works well in terms of the sound that it produces. This is a force per unit length on the string equal to $R\partial^2 V/\partial x^2$, where R is a coefficient that has to be chosen by trial and error to produce a good sound. (The value of R that works well may depend on the other parameters of the string, so in a multistring instrument, one should allow each string to have a different value of R.) The modification of the above code that includes this form of damping is as follows.

```
j=2:(J-1); % make a list of the indices of interior points
for clock=1:clockmax
  t=clock*dt;
  V(j)=V(j)+(dt/dx^2)*(T/M)*(H(j+1)-2*H(j)+H(j-1)) ...
           +(dt/dx^2)*(R/M)*(V(j+1)-2*V(j)+V(j-1));
  H(j)=H(j)+dt*V(j);
end
```

The stability restriction on Δt is now more complicated:

$$\frac{(\Delta t)^2}{(\Delta x)^2}\frac{T}{M}+2\frac{\Delta t}{(\Delta x)^2}\frac{R}{M}\leq 1 \tag{20}$$

This is a beautiful result. When $R=0$ it reduces to the stability condition stated above for the wave equation without damping. When $T=0$, it reduces to a well known stability condition for the solution of the diffusion equation by Euler's method. What is not so obvious is how these two stability condions will combine in our case, and the answer is given by the above inequality.

Since the left-hand side of (20) is an increasing function of Δt for positive Δt, we can solve for the largest value of Δt that satisfies the inequality by considering the equality case, which can be regarded as a quadratic equation for Δt. We call the positive solution of this equation $(\Delta t)_{\text{max}}$. It is given by

$$(\Delta t)_{\text{max}}=-\frac{R}{T}+\sqrt{\left(\frac{R}{T}\right)^2+\frac{(\Delta x)^2}{(T/M)}}. \tag{21}$$

Next, we consider how to hear the sound that the program produces. We need to extract from the state of the string at each time step a number that we can interpret as being the sound level at that instant of time. One might think of using the average of H or the averge of V, but this would eliminate all of the even harmonics, so it is probably not the best way to listen to the string. Physically, the string communicates the sound to air not so much directly as through the body of the instrument, to which the string is attached at its ends. This motivates listening to what is happening at one end of the string, but the end itself is not moving, so the position of the first interior point might be a reasonable choice. Thus, in the code that follows, the output from the string will be taken to be `H(2)`.

It will not be practical to hear the sound while the program is running, so the samples of the sound need to be stored in an array `S`, and we may not want to send every timestep to the speaker, so we provide a parameter `nskip` and record only those results for which `clock` is divisible by `nskip`. The choice of `nskip` will be discussed later. Code that implments this, and also plays the sound at the end, is as follows:

```
count=0;
S=zeros(1,ceil(clockmax/nskip));
j=2:(J-1); % make a list of the indices of interior points
for clock=1:clockmax
  t=clock*dt;
  V(j)=V(j)+(dt/dx^2)*(T/M)*(H(j+1)-2*H(j)+H(j-1)) ...
           +(dt/dx^2)*(R/M)*(V(j+1)-2*V(j)+V(j-1));
  H(j)=H(j)+dt*V(j);
  if(mod(clock,nskip)==0)
    count=count+1;
    S(count)=H(2); %sample the sound level at the present time
  end
end
soundsc(S(1:count)) %play the recorded samples of the sound
```

In this code, `count` keeps track of where to put the recorded sound in the array `S`. It is good (although not strictly necessary) to pre-allocate the array `S` by filling it with zeros. This makes the program run a lot faster, since otherwise Matlab has to keep growing the array every time a new entry is put into it. The function `ceil` converts a non-integer real number to the integer just above it, and it leaves integers unchanged. Within the loop over time steps, the statement `if(mod(clock,nskip)==0)` makes sure that we record sound only when `clock` is divisible by `nskip`, and the line `count=count+1` makes sure that we put the recorded sound in the right place in the array `S`.

Finally, the function `soundsc` actually plays the recorded sound. The letters `sc` at the end of this function name refer to scaling of the sound recording so that its amplitude is always within the interval $[-1, 1]$. There is another function `sound` which does not do this scaling but instead replaces all values above 1 by 1, and all values below -1 by -1, and this truncation badly distorts the quality of the sound.

Next, we consider the setup of the string, including both the choice of parameters and the specification of initial conditions. If we want a string to have a particular fundamental frequency, there are many ways to achieve this, since the fundamental frequency is determined by the three parameters T,M, and L, see equation (10). Here, we choose $L = 1$ and $M = 1$, and then choose T to get the right frequency. The code for this is

```
f1=440 % frequency in Hz (cycles/second)
L=1;M=1;T=M*(2*L*f1)^2;
```

Another parameter that we need to set is the amount of time τ that we want the vibration of the string to last before dying away. More precisely, the amplitude will decay exponentially, and τ is the time constant of this exponential decay, i.e., the amound of time that it takes for the amplitude to decay by a factor of $1/\mathrm{e}$. Once τ has been chosen, it can be used to set the parameter R as follows:

```
tau = 1.2; %decay time (seconds)
R= (2*M*L^2)/(tau*pi^2);
```

We also need to set the numerical parameters `dx` and `dt`. The value of `dx` follows immeditately from the choices of `L` and `J`, but the choice of `dt` is complicated by the following considerations. First, we have to ensure that `dt` is smaller than `dtmax`, see equation (21). Also, we are going to send samples of the computed sound to the speakers, and this works best when the freqency of the samples (not to be confused with the frequency of the sound) is 8192 samples/second. Thus, the interval between samples should be (1 second)/8192. The time step may have to be smaller than this, however, to satisfy the stability criterion. In that case, we can make everything work out well by choosing `dt=1/(8192*nskip)`, where `nskip` is a positive integer that is chosen large enough to make the timestep small enough, and then, instead of sending the result of every time step to the speaker, we send only those time steps for which clock is divisisible by `nskip` as we have already discussed. Code that sets `dx` and then `dt` and `nskip`, along with some related parameters, is as follows:

```
J=81;dx=L/(J-1);
%max time step for numerical stability:
dtmax=-(R/T)+sqrt((R/T)^2+(dx^2/(T/M)));
%Now set dt and nskip such that
%dt<=dtmax, nskip is a positive integer, and dt*nskip = 1/8192
%Also, make nskip as small as possible, given the above criteria.
nskip = ceil(1/(8192*dtmax));
dt=1/(8192*nskip);
tmax=     %total time of the simulation in seconds
clockmax=ceil(tmax/dt);
```

The next thing to do is to specify initial conditions for the string. Here it matters how the string is set in motion. A guitar string, for example, is plucked, and a piano string is hit with a hammer.

To pluck a string is to pull on one point of the string, and then let go. At the moment of release, which we call $t = 0$, the string is not moving, so $V(x, 0) = 0$. The displacement $H(x, 0)$ is of the following form:

$$H(x,0) = H_p \frac{x}{x_p}, \quad x \le x_p, \tag{22}$$

$$H(x,0) = H_p \frac{L-x}{L-x_p}. \quad x_p \le x. \tag{23}$$

Note that this is a pair of straight lines, one through the origin and the other through the point $(L, 0)$, that meet at the point (x_p, H_p). Thus x_p is the position of the pluck, and H_p is its amplitude. The code that implements the above is as follows:

```
V=zeros(1,J);
xp=L/3;Hp=1;
for jj=1:J
  x=(jj-1)*dx;
  if(x<xp)
    H(jj)=Hp*x/xp;
  else
    H(jj)=Hp*(L-x)/(L-xp);
  end
end
```

Note that we don't have to worry about the borderline case `x=xp`. It takes care of itself because the two formulae agree at that point.

When a string is hit with a hammer, the immediate result is nonzero velocity of the part of the string that was hit. The displacement of the srtring, immediately after the hammer hits it, is zero, since it takes time for displacement to develop. The hammer is defined by an interval $(x_{\mathrm{h1}}, x_{\mathrm{h2}})$ that defines the part of the string that is hit by the hammer. Let V_{h} be the velocity that is imparted to this part of the string. The initial conditions are then given by $H(x, 0) = 0$ and

$$V(x,0) = V_h, \quad x_{\mathrm{h1}} < x < x_{\mathrm{h2}}, \tag{24}$$

$$V(x,0) = 0, \quad \text{otherwise.} \tag{25}$$

The code that implements this is as follows:

```
H = zeros(1,J);
V = zeros(1,J);
xh1=0.25*L;xh2=0.35*L;V_h=1;
%list of points hit by hammer:
jstrike=ceil(1+xh1/dx):floor(1+xh2/dx);
V(jstrike) = V_h;
```

Now we are ready to put everything together. We'll use the plucked string as an example, but it is simple enough to change the intial conditions to those of a string struck by a hammer as we have just described. The code for the plucked string is as follows:

```
%vibstring.m
clear all
f1=440 % frequency in Hz (cycles/second)
%string parameters to make frequency f1:
L=1;M=1;T=M*(2*L*f1)^2;
tau = 1.2; %decay time (seconds)
%damping constant to make decay time tau:
R= (2*M*L^2)/(tau*pi^2);
J=81;dx=L/(J-1);
%maximum time step for numerical stability:
dtmax=-(R/T)+sqrt((R/T)^2+(dx^2/(T/M)));

%Now set dt and nskip such that:
%dt<=dtmax, nskip is a positive integer, and dt*nskip = 1/8192.
%Also, make nskip as small as possible, given the above criteria.
nskip = ceil(1/(8192*dtmax));
dt=1/(8192*nskip);

tmax= 4;   %total time of the simulation in seconds
clockmax=ceil(tmax/dt);

%initial conditions for plucked string:
V=zeros(1,J);
xp=L/3;Hp=1; %position and amplitude of pluck
for jj=1:J
  x=(jj-1)*dx;
  if(x<xp)
```

```
    H(jj)=Hp*x/xp;
  else
    H(jj)=Hp*(L-x)/(L-xp);
  end
end

%Initialize graphics to see a movie of the string:
set(gcf,'double','on') %turn on double-buffering
%make initial plot:
Hhandle=plot(0:dx:L,H,'linewidth',2);
axis([-0.2*L,L+0.2*L,-Hp,Hp]) %set axis limits
axis manual %freeze axes
drawnow

count=0;
S=zeros(1,ceil(clockmax/nskip));
tsave = zeros(1,ceil(clockmax/nskip));

j=2:(J-1); % list of indices of interior points
for clock=1:clockmax
  t=clock*dt;
  V(j)=V(j)+(dt/dx^2)*(T/M)*(H(j+1)-2*H(j)+H(j-1)) ...
           +(dt/dx^2)*(R/M)*(V(j+1)-2*V(j)+V(j-1));
  H(j)=H(j)+dt*V(j);
  if(mod(clock,nskip)==0)
    count=count+1;
    S(count)=H(2); %sample the sound
    tsave(count)=t; %record sample time
  end
  set(Hhandle,'ydata',H)  %update movie frame
  drawnow                 %show latest frame
end
soundsc(S(1:count)) %play the recorded soundwave

%plot the soundwave as a function of time:
figure(2)
plot(tsave(1:count),S(1:count))
```

The above code will generate a movie showing the string as it vibrates. As written, every timestep will be plotted. Alternatively the two lines

```
set(Hhandle,'ydata',H)  %update movie frame
drawnow                 %show latest frame
```

could be moved up to within the `if` statement, and then then there would be one frame shown every `nskip` timesteps, instead of every timestep. Yet another possibility would be to put those two lines within their own `if` statement with a separate parameter like `nskip` (but with a different name, of course) to control the frequency with which frames of the movie are generated.

5 Piano Program

In this section we move up from a single string to the simulation of an array of strings. The goal is to model a piano. We start by introducing a simple low-level language for specifying the piece of music that the piano will play. This language will allow for the playing of chords. In fact, any number of notes can be played at the same time. Each note that is to be played has four attributes: the time at which the note starts (in seconds, measured from the time at which the music starts); the duration of the note (in seconds); the relative amplitude or loudness of the note (a dimensionless number); and an integer index designating the string on which the note is to be played.

These attributes are specified in four one-dimensional arrays:

```
tnote, dnote, anote, inote,
```

all of the same length, `nmax`, where `nmax` is the number of notes to be played. The notes are listed in these arrays in order of their start time, and if two or more notes start at the same time, their order does not matter. Thus the entries in `tnote` are a non-decreasing sequence of start times. The durations that are specified in `dnote` refer to the amount of time that the key on the piano keyboard is held down. A feature of the piano is a mechanism that stops the vibration of a string when its key is no longer depressed, and we simulate that mechanism here. (On a real piano, there is a pedal that interferes with this mechanism, so that a note can continue to sound regardless of whether its key is up or down, but we ignore that possibility here.) The entries in `anote` are relative amplitudes only, since the sound will eventually be played by the Matlab command `soundsc`, which scales the entire soundwave to lie within the interval $[-1, 1]$, so if all of the entries in anote are all multiplied by some constant, the resulting sound will be the same. The entries in inote are indices that identify which string is to be used to play a given note. The frequencies of the strings are specified separately. This means that we can tune the virtual piano independently of how we specify the piece of music that is to be played on it.

In setting up the strings of the piano, we will specify the lowest frequency flow and the number of strings nstrings, and then proceed as follows

```
L=1;          % length of all strings
%number of points on each string, and resulting dx:
J=81;dx=L/(J-1);
flow = 220; %frequency of string with lowest pitch (1/s)
nstrings = 25; %number of strings
for i=1:nstrings
  f(i)=flow*2^((i-1)/12);  % frequency (1/s)
  tau(i)=1.2*(440/f(i));   % decay time (s)
  M(i)=1;                  % mass/length
  T(i)=M(i)*(2*L*f(i))^2;  % tension
  R(i)=(2*M(i)*L^2)/(tau(i)*pi^2); % damping constant
  %Find the largest stable timestep for string i:
  dtmax(i) = - R(i)/T(i) + sqrt((R(i)/T(i))^2 + dx^2/(T(i)/M(i)));
end
%Timestep that will be stable for all strings:
dtmaxmin = min(dtmax);
%Now set dt and nskip such that:
%dt<=dtmaxmin, nskip is a positive integer, and dt*nskip = 1/8192.
%Also, make nskip as small as possible, given the above criteria.
nskip = ceil(1/(8192*dtmaxmin));
dt=1/(8192*nskip);
```

The above code creates an array of strings with pitches separated by a half-step (frequency ratio of $2^{1/12}$), as in a real piano. A decay time is assigned to each string, with a decay time of 1.2 seconds for a string with a frequency of 440 Hz, and with the decay time inversely proportional to the frequency of the string. (This is just a guess, both with regard to the 1.2 seconds and also with regard to the dependence on frequency, and adjustment may be required to get a realistic sound.) The maximum stable time step is calculated separately for each string, but in the end we have to choose one time step that will be stable for all of them, so we choose the minimum of the entries in `dtmax` and proceed from there as we did with the single string.

To make the code run as fast as possible, we should not try to cover a larger range of frequencies than is needed to play the music that we have in mind. This is especially true at the high-frequency end, since the timestep is typically limited by the highest-frequency string. Following this thought further, we could make the virtual piano contain *only* the strings that will actually be played, but this would complicate the specification of the music, since there would then no longer be any simple rule that translates between string number and frequency.

With the above code, we do have such a rule. Recall that each entry in `inote` is the index of the string on which the note is to be played. If the entry is `i`, this means that the note is `i-1` half-steps above the note with frequency `flow`. This means that we can easily change key, merely by changing the parameter `flow`, without changing any of the entries in `inote`.

Another change that can easily be made concerns the tempo with which the music is to be played. This can be changed merely by multiplying both of the arrays `tnote` and `dnote` by the same positive constant. To slow the tempo, choose a constant greater than one, and to speed it up, choose a constant smaller than one. Unlike running a tape or a record (to mention some old technologies) at a faster or slower speed than normal, these changes will have no effect on the pitches of the notes.

Since we have multiple strings, the variables `H` and `V` are now two-dimensional arrays, with the first index `i` being the string number, and the second index `j` specifying the location within the string. The strings are initialized at rest, and an array `jstrike` is created to specify the set of points that will be hit by the hammer when the string is played:

```
H=zeros(nstrings,J);
V=zeros(nstrings,J);
%(xh1,xh2) = interval on string hit by hammer:
xh1=0.25*L;xh2=0.35*L;
%initialize list of points hit by hammer:
jstrike=ceil(1+xh1/dx):floor(1+xh2/dx);
```

The most interesting part of the piano program is a `while` loop, *executed during each time step*, that starts all of the notes playing that were not already started and whose start time has been reached or exceeded. This loop reads as follows (with explanation below):

```
  t=clock*dt;
  while((n<=nmax) && tnote(n)<=t)
    V(inote(n),jstrike)=anote(n); %strike string inote(n)
                                  %with amplitude anote(n)
    tstop(inote(n))=t+dnote(n);   %record future stop time
    n=n+1                         %consider next note
  end
```

This loop progresses through the notes of the music, which are indexed by `n` in order of their start time, to see whether any notes that have not already been started need to be started now. If so, the relevant string is struck, within the range of points specified by the array `jstrike`, and the future stop time for the note that is being started is stored in the array `tstop`, for which the index is the string number, so that the string in question will know when to stop vibrating. (The stop time is here written as `t+dnote(n)`, but it could also have been written as `tnote(n)+dnote(n)`, since these are within `dt` of each other.) The `while` loop continues until it encounters a note whose start time is still in the future, or until there are no more notes to be played.

An important detail concerns the logical operator `&&` that appears in the condition of the `while` loop. A statement of the form `A && B` evaluates to TRUE if and only if `A` and `B` are both true, and moreover A is evaluated first, and if it turns out to be false, B is not evaluated. This avoids what would otherwise be a nasty bug in the code. If `n > nmax`, then `tnote(n)` is undefined, and the attempted evaluation of `tnote(n)` will stop the program with an error message. The case `n > nmax` inevitably happens in our program once the last note has been started and before it has finished playing, and it would be particularly frustrating for the program to fail in such a way at the very end, especially if it has been running for hours.

We now have most of the pieces and can put everything together:

```
%piano.m
clear all
%specify the music to be played:
nmax=     ; %number of notes
tnote = []; %array of onset times of the notes (s)
dnote = []; %array of durations of the notes (s)
anote = []; %array of relative amplitudes of the notes
inote = []; %array of string indices of the notes

%initialize string parameters:
L=1;          % length of all strings
J=81;dx=L/(J-1); %number of points/string, space step
flow = 220; %frequency of string with lowest pitch (1/s)
nstrings = 25; %number of strings
for i=1:nstrings
  f(i)=flow*2^((i-1)/12);  % frequency (1/s)
  tau(i)=1.2*(440/f(i));   % decay time (s)
  M(i)=1;                  % mass/length
  T(i)=M(i)*(2*L*f(i))^2;  % tension
  R(i)=(2*M(i)*L^2)/(tau(i)*pi^2); % damping constant
  %Find the largest stable timestep for string i:
  dtmax(i) = - R(i)/T(i) + sqrt((R(i)/T(i))^2 + dx^2/(T(i)/M(i)));
end
%The timestep of the computation has to be stable for all strings:
dtmaxmin = min(dtmax);
%Now set dt and nskip such that:
%dt<=dtmaxmin, nskip is a positive integer, and dt*nskip = 1/8192.
%Also, make nskip as small as possible, given the above criteria.
nskip = ceil(1/(8192*dtmaxmin));
dt=1/(8192*nskip);
tmax=tnote(nmax)+dnote(nmax); clockmax=ceil(tmax/dt);
%initialize an array that will be used to tell a string
%when to stop vibrating:
tstop=zeros(nstrings,1);
%initialize arrays to store state of the strings:
H=zeros(nstrings,J);
V=zeros(nstrings,J);
```

```
%(xh1,xh2)=part of string hit by hammer:
xh1=0.25*L;xh2=0.35*L;
%list of points hit by hammer:
jstrike=ceil(1+xh1/dx):floor(1+xh2/dx);
j=2:(J-1); %list of interior points
%initialize array to store soundwave:
count=0; %initialize count of samples of recorded soundwave
S=zeros(1,ceil(clockmax/nskip)); %array to record soundwave
tsave = zeros(1,ceil(clockmax/nskip)); %array for times of samples

n=1 ; %initialize note counter
for clock=1:clockmax
  t=clock*dt;
  while((n<=nmax) && tnote(n)<=t)
    V(inote(n),jstrike)=anote(n); %strike string inote(n)
                                  %with amplitude anote(n)
    tstop(inote(n))=t+dnote(n);   %record future stop time
    n=n+1;                        %increment note counter
  end
  for i=1:nstrings
    if(t > tstop(i))
      H(i,:)=zeros(1,J);
      V(i,:)=zeros(1,J);
    else
      V(i,j)=V(i,j) ...
            +(dt/dx^2)*(T(i)/M(i))*(H(i,j+1)-2*H(i,j)+H(i,j-1)) ...
            +(dt/dx^2)*(R(i)/M(i))*(V(i,j+1)-2*V(i,j)+V(i,j-1));
      H(i,j)=H(i,j)+dt*V(i,j);
    end
  end
  if(mod(clock,nskip)==0)
    count=count+1;
    S(count)=sum(H(:,2)); %sample the sound at the present time
    tsave(count)=t;       %record the time of the sample
  end
end
soundsc(S(1:count)) %listen to the sound
plot(tsave(1:count),S(1:count)) %plot the soundwave
```

The above code does not generate an animation of the strings, since the code will have to run for a long time to produce any significant piece of music, and animation would slow it down substantially, but of course animation could be added. In such an animation, it would be good to separate the strings by adding a constant proportional to the string number to the displacement of each string (for plotting purposes only). Another possibility would be to make a 3D plot in which the strings appear side by side, as in a real piano.

Also, since the program will take a long time to run, it would be a good idea to save the array `S` by writing it to a file, so that it can be heard on demand, without having to run the program again to re-generate it. The Matlab commands `save` and `load` can be used for this purpose.

6 Guitar Program – A Challenge for the Reader

It would be an interesting project to adapt the foregoing program to the case of a guitar. The most important difference between the guitar and the piano is that each string of the guitar has, in effect, a variable length based on where the player holds down the string with a finger. The possible lengths are discrete, however, because the guitar has frets, which are ridges that run perpendicular to the strings. A finger placed on a string between two frets holds the string against these frets, and the fret that is nearer to the body of the instrument becomes, in effect, a fixed end of the part of the string that is vibrating. The frets are placed so that shortening the string by one fret raises the pitch of the sound by a half-step. It is an interesting mathematical fact that this can be accomplished simultaneously for all strings by frets that run across the neck of the guitar, so that the same set of lengths works for strings with different tensions and possibly also different masses per unit length. This can be undestood from the theory of the vibrating string discussed earlier, and that theory can also be used to derive the correct positions of the frets, which are not equally spaced. We leave details to the interested reader.

The variable effective length of a guitar string can be simulated in different ways. One is to change the parameter L. Although this can be made to work, it has the disadvantage that Δx then has to change, and possibly also Δt to ensure numerical stability. Changing L is also un-physical. When a fret is used, it is not really the length of the string that changes, but rather that only part of the string is allowed to vibrate. My suggestion, then, is to keep L, J, and Δx all fixed, but if a fret at at a given value of x is in use for a given string, to set and hold H and V equal to zero at all grid points for that string on the side of the fret that is farther from the body of the instrument, for as long as the fret is held down.

Another difference is the manner in which the vibration is initiated, which is by a pluck in the case of a guitar. This has been discussed above, but note that the pluck is also influenced by the fret that is being held down when the pluck occurs. This requires a minor but important change in the code for the initial condition of a plucked string.

The language for specifying the music that is to be played also has to change. The array `dnote` is no longer needed (and therefore `tstop` is no longer needed as well), since there is nothing on the guitar corresponding to the mechanism on a piano that stops the note when the finger of the pianist comes off of the key. Another change is that in the case of a guitar a note needs to be specified not only by the string number but also by the fret number. Thus, instead of only `inote`, we now need two arrays, e.g., `inote` and `fnote`, where `inote(n)` is the string number and `fnote(n)` is the fret number.