Workshop Notes—Physical Modeling Sound Synthesis using Finite Difference Schemes

Stefan Bilbao
University of Edinburgh
May, 2008

1 Introduction

Welcome to this physical modeling workshop, which deals with a special type of method for simulating musical systems—finite difference methods. These are not the same as digital waveguides, and are not like modal synthesis methods either. Instead, the total behaviour of a musical object, such as a string, bar, plate, or acoustic tube, is simulated directly, using approximate values over a grid.

Some of the basics of difference schemes for synthesis will be covered in a slide presentation, so do take notes. There will be three types of systems covered here: strings/bars, acoustic tubes, and finally plates. For each type of system, there is an accompanying graphical user interface, written in the Matlab programming language. Additionally, if there is time, we'll look at some very basic issues in programming these types of methods directly in Matlab.

2 Getting Started With Matlab

A parallel goal of this workshop is to obtain a familiarity with Matlab, a very useful prototyping tool for audio DSP engineers. Most research scientists doing any kind of numerical work rely almost exclusively on matlab. You can do anything you can do with any other language, but the syntax is much easier to handle than, say, that of C++. The price you pay, of course, is that the code you write won't be very fast. But matlab isn't meant for this; as mentioned above, it's more of a prototyping tool.

Launching Matlab

Simply double click on the matlab icon. Several windows will open; the one called "Command Window" is the one that interests us for the moment. The ">>" symbol is the prompt, and indicates matlab is ready to accept a command.

Help

Matlab has built in help and documentation for all features. To get a list of all help topics, type "help" at the prompt, followed by the return key. This is your first command. You'll see a list of all the help topics in matlab. To read about any particular topic, simply type "help ***" where *** is one of the items on the list you just printed out.

Directory Structure in Matlab

One of the things about matlab that you need to know about is its directory structure, which is similar to that of a UNIX system. You can move about the whole file hierarchy of your computer within matlab. **For instance, type "pwd."** This is short for "present working directory," and refers to the place where you are currently within the hierarchy. **Now type "ls."** This is a list of everything in the current directory, not just files, but other subdirectories as well.

Another important command is "cd," meaning "change directory." Typing "cd **" moves down the hierarchy to the directory **, and typing "cd .." moves up the hierarchy.

You should move around the hierarchy until you are in the directory called "PM workshop." (I don't know where this is yet!).

There is, of course, much more to say about Matlab, and hopefully, if there is time, you will be able to do some physical modeling coding of your own a bit later...

3 Strings and Bars

Some components of musical instruments are conveniently described in "1D," meaning that one dimension of the object is significantly longer than the others. Among the best known are strings, used in keyboard instruments such as the piano, and in members of the string instrument family, and in guitars, and bars, used typically in percussion instruments such as xylophones, marimbas, chimes, etc. The most basic difference between a string-like object and a bar is in how it responds to a "push," or deformation. A string possesses no restoring force of its own; it must be under tension in order to resist such a force, and thus vibrate. A stiff object, like a bar, does not require that such tension be applied. From a perceptual standpoint, the resulting difference in timbre is rather striking—string-like objects tend to produce harmonic tones, and stiff objects, in general do not. For this reason, it is more difficult to associate a pitch with the sound of a bar.

The basics of string and bar physics will be described in lecture. Once you have a basic working understanding of these objects, you may begin with the graphical user interface, in order to explore some of the sonic properties of these objects, as well as some interesting virtual extensions. By following the tutorial examples below, you move from very simple configurations (which sound synthetic and terrible) to more complex settings where there are a lot of parameters to look after—it's a good idea to take them in one at a time!

Starting the GUI

Once you are in the correct directory in Matlab (where the tutorial files are located), you can start the GUI by typing `ppgui`. You may adjust paramaters as you like; in following the tutorial below, go to the panel at the top right, and press the corresponding button. In order to generate sound output, you then press "Compute and Play Sound." If you want to hear the same sound again, you don't need to recompute, just press "Play again."

3.1 Example 1: An Ideal String

An ideal string is one of infinitesinal thickness, no loss, and vibrating at low amplitudes; furthermore, every point on the string is assumed to move up and down only; there is no longitudinal motion. Such a string is described by three parameters:

- length L (m)
- tension T (N)
- mass density ρ (kg/m)

In real musical instruments, all three of these vary quite a bit from one instrument to the next.

Pitch

In fact, though, it is only a combination of these parameters which has perceptual significance. For instance, if the string is fixed at both ends, the number f_0, defined as

$$f_0 = \sqrt{T/\rho}/(2L)$$

is completely sufficient to describe the sound produced by the string. In other words, doubling the string length leads to the same result as lowering the tension by a factor of four. Try running Example 1 now.

In fact, f_0 is the fundamental frequency (pitch) of the string. It relates to pitch in the following way. Given that A above middle C is at 440 Hz, you may calculate f_0 as

$$f_0 = (440)2^{d/12}$$

where d is the number of semitones of the pitch above A440.

Try generating tones at the following pitches:

- an A an octave above A440
- an A two octaves below A440
- middle C

You should be able to calulate the frequency f_0 for such pitches using Matlab at the command line.

Sample Rate

You may choose, also, the sample rate at which the simulation is computed (in Hz). Note that, for simplicity, you are constrained to choose values of the sample rate which are typical of audio applications, such as 44100 Hz, 22050 Hz, etc. This is a choice made at the level of the GUI only—such a constraint is not inherent to physical modeling synthesis methods, but is a wise choice!

Duration of Simulation

You must also specify the duration of the simulation (in seconds).

Listening

The output is a steady tone—very synthetic sounding! (Why?) Experiment with various settings for the pitch, duration and sample rate.

Computing Time

The "timer on" box allows you to display the computing time, in seconds, for a given synthesis run. How does synthesis time depend on f_0, the fundamental frequency, for a given sample rate? On the sample rate for a given fundamental frequency?

Saving Soundfiles

In order to save sound output, as a .wav file, press "Save Soundfile," and enter a prefix. One slight bug here is that you need to make sure that there is no soundfile open with this name at the instant you save!

3.2 Example 2: Adding Loss

Loss is crucial, in that it leads to decay in string tones. There are many sources of such loss; all are related to various mechanisms, such as sound radiation, and internal damping. As such, the way in which sounds decay is rather complex.

A key parameter is the 60 dB decay time, sometimes written as T_{60}. It refers to the amount of time necessary for the amplitude of a tone to decay to 1/1000000 of its initial value. Typical values in real instruments range from about 1 s to upwards of 10 or 20 seconds (for, say, a grand piano). In most real world instruments, T_{60} is not a constant, but depends on the frequency; generally, the higher frequency content decays more rapidly than the lower frequency content. (Note that, in the present case of string

tones, although a fundamental frequency or pitch has been specified, this does not mean that such tones possess a single frequency component!).

A simple two-parameter loss model allows you to set the pairs

$$f_1, T_{60,1} \qquad \text{and} \qquad f_2, T_{60,2}$$

at two frequencies f_1 and f_2, in Hz. In the Example, the frequencies are chosen as 100 Hz, and 1000 Hz, and the decay times as 5 s and 2.5 s. Generally, you must choose $f_2 > f_1$, and $T_{60,2} < T_{60,1}$!

This effect is easier to hear if you use a low fundamental frequency, such as $f_0 = 50$ Hz, a relatively long decay time, such as 10 s, for $T_{60,1}$, and then vary $T_{60.2}$. try experimenting with this.

As you might expect, at frequencies intermediate between these two, the decay time is intermediate as well.

Does loss have an effect on computing time? Experiment with this, using the timer.

3.3 Example 3: Adding Stiffness

Real strings are stiff; that is, they are not like rubber, or shoelaces, which require tension to be applied in order to support vibration. Stiffness has a rather important effect on timbre, as you will see here.

In addition to the length L, tension T, and mass density ρ, other parameters describe such a stiff system: Young's modulus E, and the cross-sectional area A. The complete effect, however, requires only a single new parameter, "stiffness."

Actually, the addition of stiffness allows a whole range of objects to be simulated, with the limiting cases the string without any stiffness, to a bar without tension, such as that used in a xylophone, or marimba. The relative effect of stiffness is controlled by the "stiffness" parameter mentioned above—for musical applications, it takes on values ranging from about 0.01 up to 200 or more. When it is small relative to f_0, the resulting sound will be pitched, but slightly inharmonic, or detuned.

Inharmonicity

Try taking a fixed value for f_0, and raising the stiffness, from a value of 0.01, up to approximately 100, in steps, and listen to the change in timbre from string like to bar-like. Notice in particular that f_0 no longer corresponds to a pitch—indeed, highly inharmonic sounds (like bells) may not have a definite pitch at all!

Sound Examples

By playing with the frequency f_0, stiffness, and the loss parameters, try to generate sounds similar to

- a large bell
- a small bell
- a xylophone or marimba

Computing Time

How does computing time depend on the stiffness? Work though different values of stiffness again, and pay attention to the timer.

High-passing

You may find that, for high values of the stiffness, the sound is a little dull. In order to brighten it, you might wish to use a simple high-pass filter, which you can turn on from the window at the top left. Try listening to the difference in sound quality. Note: high-passing is not physical modeling, just an audio trick!

3.4 Example 4: Stereo and Scanning

One of the very important benefits of time-domain models like finite differneces, is that because you are computing the behaviour of the object (such as a string) everywhere, you have immediate access to the solution everywhere. Thus you may take as many outputs as you want...this can be very useful for spatializing audio output from physical models, and it is a feature that not all methods possess; modal synthesis methods, for example, must recompute each output separately, which can take a long time! In this tutorial, we'll look just at stereo output, but you could generate sound in as many channels as you like!

In general, one may read output from anywhere one likes; in this simple GUI, when the output is monophonic, it is read from the string center, and when in stereo, from two points which subdivide the string into three equal segments (i.e., at one-third and two-thirds of the way along the string).

Scanning

In addition to specifying the number of outputs, you may also allow the output positions to vary in time; that is, you may allow the output locations to "scan" the string. There are obviously many ways of specifying how the output points may vary, but a simple one is to allow them to vary sinusoidally about the static output points, at a given frequency. If the frequency is low (less than about 1 Hz), you will hear this as a delicate phasing effect between the two outputs. As the frequency becomes higher, you will here modulation effects appear, and the output will be very synthetic-sounding (i.e., something normally avoided in physical modeling). Try, from the starting set of parameters in Example 4, to adjust the scan frequency until you begin to hear this modulation. Headphones are very useful here!

Another minor parameter is the frequency spread...in stereo, the scan frequencies at the two locations may be set to be different, with a deviation of the spread. Try, for a given scan frequency, to listen to this effect.

Computing Time

Does the computing time depend on the number of outputs? Why or why not? Do you think this is true of other synthesis methods?

3.5 Example 5: Multiple Strings

So far, you have listened to a single string or bar. In some instruments, such as the piano, for example, a single strike will involve multiple strings, which need to be modelled separately. Try running Example 5. You will notice, in the figure window, that three strings are now visible.

Number of Strings

You may set the number of strings in the "No. Strings" box. In Example 5, it is set to three, but you may raise this as high as twelve in the GUI. Try running the example with more strings (say eight). Is there a difference in the decay rate when you have more strings? If so, why do you think this is?

Detuning

In the piano, for example, when a hammer strikes a number of strings, they are not all the same! In fact, the frequencies f_0 are slightly detuned. For small pitch differences, detuning is measured in cents—100 cents make up a semitone, and 1200 make up an octave. The detuning, in cents, between two frequencies f_1 and f_2 is defined as

$$Detune = 1200 \log_2(f_2/f_1)$$

For multiple strings, the value in the "Detune" box refers to the total amount of detuning over all the strings. I.e., a value of 10 means that the three strings will be detuned by 5, 0 and -5 cents with respect to

the center pitch f_0. A small value for this parameter leads to noticeable phasing effects (you might wish to turn off the scanning in order to better hear it on its own), and changes the decay rate, due to cancellation. Listen to the difference between a single string or bar, and many very slightly detuned.

However, physical modeling is not really about trying to duplicate the sounds of existing instruments—you can detune the strings or bars as much as you like. Try using a large number of strings, and a large detuning (such as, say, 1000 or more), so that the pitches are distributed over a fairly wide range; the effect can be somewhat like playing a piano with your elbows.

3.6 Example 6: The Hammer

So far, nothing has been said about how exactly the string is being excited. In fact, because this is a physical model, one needs a model of something which strikes the strings. In this case, it is a hammer, or mallet, but one could also imagine introducing other excitations, such as the bow, or a plucking finger, as well.

The hammer has two main attributes:

- mass
- stiffness

The mass is relatively easy to understand; in the GUI, the parameter "mass ratio" refers to the ratio between the mass of the hammer, and that of the segment of string with which it is in contact. The stiffness parameter is a little harder to get a grasp on; generally, the higher it is, the harder the hammer, and the brighter the sound will be. Generally, a range of about 500—3000 is good for musical sound.

Try adjusting both of these parameters, and learn the perceptual effects. Note that if you turn up the stiffness too high, you may run into trouble with numerical stability—your algorithm may crash! This is one of the difficulties with time domain methods such as finite difference schemes!

3.7 Example 7: Multiple Events

So far, the algorithm has produced a single strike. While you could build more complex sounds by "adding" sound outputs corresponding to single strikes, a more physical approach is to simply strike the same object multiple times. This will lead to a sonically different result; on a subsequent strike, the string may still be in motion!

There are several parameters which define a single strike:

- time of strike
- velocity of strike
- position of strike along strings

These may all be set, individually, for up to three strikes in the GUI; run Example 7. It is certainly possible to have more than three events, but I couldn't find a simple way of adding this to the GUI!

The strike time parameter is conceptually easy to understand. The velocity parameter is more interesting; beyond simpling making a strike louder or softer, it also changes the brightness—in general, the harder the strike, the brighter the sound. This is an effect characteristic of all percussive instruments. Try a series of three strikes of increasing velocity, but keeping the position the same. Typical values are between 10 and about 100.

The position of the strike along the strings is the most subtle parameter. By changing the position, you get a different timbre; if you know about modes, you can think about exciting different combinations of modes at different points; generally, strikes near the end points are rather bright, and those near the middle generate a hollow tone. The position is specified by a number between 0 and 1. Try generating a sequence of strikes, at the same velocity, at different locations.

3.8 Example 8: Spring Networks

So far, the system is, with the exception of the hammer, linear; things become much more interesting when nonlinear elements are attached to the strings. In a sense, you can think about these elements in the same way as preparation os acoustic instruments like pianos. This is the real point of physical modeling; you can build your own instrument, or modify it (without destroying it).

One simple way of proceeding is to attach springs between the various strings in a collection. In the GUI, three such connections are possible at once. try running Example 8.

Each spring is described by several parameters:

- indexes of the two strings to which it is connected
- positions along the two strings at which it is fastened (0–1)
- stiffness
- loss, as T_{60}

This, when coupled with all the previous parameters, enlarges the space of possible sounds greatly!

The string indexes are set in a menu; it is possible to connect a string to itself. Try varying this, for a collection of two strings, beginning from the configuration of Example 8. Disable all but one spring, and connect it first from string 1 to itself, then from string 1 to string 2.

The positions of the string connections also make a big difference. Again, for a collection of two strings and for a single spring, try holding the position at one string constant, and then varying the other.

For a single spring connection, try varying the stiffness constant. By comparing output with the case of no spring network (you can turn it off from the left-hand panel), see if you can find the value of stiffness for which one begins to hear a difference in timbre. Use just one spring, and set the damping constant T_{60} to a very high value, such as 1000, effectively turning it off!

For a single spring connection, try varying the damping constant T_{60}.

3.9 Example 9: Rattles

Another type of element, very useful from the point of view of string preparation, is the element which is free to rattle on the strings. Each element is connected (by default) to all the strings, and is subject to gravity! Such an element is characterized by several parameters:

- position along strings (0–1)
- mass ratio, relative to string (generally less than 1)
- stiffness (between 1000 and about 3000)
- physical length (m) (less than 0.1)

Try varying one or more of the parameters, beginning with just a single rattling element. You may notice, particularly for large values of the mass ratio, and for high stiffnesses, that the sound produced is synthetic sounding, or that the algorithm becomes unstable...such are the pitfalls of time domain modelling!

4 Reed Wind Instruments

Wind instruments, though quite different in many respects from strings and bars, share the feature of being 1D...that is the tube is long in one dimension and short in the others. Now, however, the vibrating material is not a solid, but the air itself contained in the tube. One key difference here is in the excitation—unlike percussive instruments, it is applied continuously. Another difference is in the nonlinearity which appears...here, it is fundamental to the oscillation itself, whereas a string behaves somewhat as expected under even linear conditions.

Starting the GUI

Once you are in the correct directory in Matlab (where the tutorial files are located), you can start the GUI by typing `windfdgui`. You may adjust paramaters as you like; in following the tutorial below, go to the panel at the top right, and press the corresponding button. In order to generate sound output, you then press "Compute and Play Sound." If you want to hear the same sound again, you don't need to recompute, just press "Play again."

4.1 Example 1: Acoustic Tube

Several of the features which appeared in the GUI for string and bars remain unchanged here: the sample rate, duration of simulation, the timer, and soundfile generation, playing, and saving, are as before.

In the first example, a very simple tone is generated using an acoustic tube model; as yet, the specifics of the type of tube, as well as of what the exciting mechanism is are hidden. There are two key parameters which you could vary:

- wave speed c, in m/s
- tube length L, in m

In the case of the tube, as opposed to the string, these cannot (quite) be reduced to a single parameter for the pitch, due to more subtle effects which we'll get to later.

Try playing with both these parameters, to see what the effect on the resulting tone is. What is the effect of doubling L? What is the effect of doubling c? What is the effect of doubling both at once?

How does computation time depend on c and L? Use the timer.

4.2 Example 2: Bore Profile

In the example 1 above, the tube is cylindrical. One of the nice things about time domain methods is that it is very easy (almost trivial, in fact) to use any shape of tube you like. Though the tube itself is always assumed to be of coircular cross-section, the area of the cross-section may vary however you like along the length of the bore.

Here, even though you have chosen the tube length separately, it is convenient to represent the bore profile as a function between $x = 0$ and $x = 1$, as shown in the plot.

Cylinders

In Example 1, you listened to the sound of a cylindrical bore; in general, it has the "woody" tone of instruments such as the clarinet.

Cones

Other wind instruments, such as saxophones and oboes, actually have a bore profile which is closer to that of a cone. You may seles a cone profile from the popup menu. For a conical instrument, there is another parameter, which relates to the flare of the cone. Here, you may set the ratio of the area at the right end of the tube to the left end.

Begin from a value of this ratio at 1, so that you have a cone, and gradually increase it, so that the flare becomes larger. How does the sound change depending on the flare? Can you always get a sound out?

Cyliders with Bells

Most wind instruments are in fact neither cylindrical, nor conical, but have a bell like extension at the radiating end. You may select such a profile from the popup menu.

Here, there are two parameters: the ratio of the surface area of the tube at the right end to that at the left end, and also the fraction of the total bore length occupied by the bell.

From what you know about clarinets, attempt to find a bore profile which matches. For a given bell fraction, how is pitch altered as you increase the surface area ratio?

Also, how does computation time depend on your choices of bore profile, and the associated parameters?

4.3 Example 3: The Reed

So far, nothing has been said about the excitation mechanism—in fact, it is a reed, and is characterized by quite a number of parameters:

- Reed surface area (m^2)
- Reed fundamental frequency (rad./s)
- Reed equilibrium displacement (m)
- Reed mass
- Reed damping parameter (1/s)
- Channel width (m)

In Example 3, parameters roughly corresponding to a clarinet are loaded. You will notce, as you try varying these (do this for all the parameters) that in fact, the range of values for which you can get a steady tone is quite limited...so be gentle!

4.4 Example 4: Pressure Excitation

The reed instrument is driven by a control signal, supplied by the user, or player: this is a mouth pressure waveform. Generally, as you would expect, this is a very slowly varying function of time. In example 4, the bore/reed parameters are set to those of, roughly a clarinet, and you may vary the strnegth of the pressure excitation, which is set initially to 1800 Pascals. What happend as you raise this value? How far can you lower it before the clarinet model no longer oscillates?

This, by the way, is a great example of nonlinearity in musical instruments; in this wind model, the reed mechanism is strongly nonlinear!

In the GUI, you also have a very crude control over the shape of the pressure waveform: ramp factor allows the input pressure to increase linearly over the duration of the sound, by the factor specified, and the rise time and decay time are additional gestural paramaters. Try playing, in particular, with the rise time: how does the ability of the instrument to sound depend on this value?

4.5 Example 5: Overblowing

Just as in real wind instruments, if you blow an instrument hard enough, its pitch will jump upwards by a large interval. This is easiest to achieve in a conical bore reed instrument, roughly like a saxophone. In example 5, the pressure waveform is initially set to 2300 Pa. Very gradually increase this value—at what pressure does the instrument clearly sound in a higher register)nearly an octave above)?

It is very insteresting to listen to the waveforms produced in the intermediate pressure values—the instrument has trouble "locking" into a given register, and "stalls," somewhat like a car motor, shuddering. This type of behaviour is very much characteristic of sounds produced both by beginners playing real wind

instruments, as well as masters of modern techniques involving so-called multiphonics. This is a great example of a type of sound which would be nearly impossible to produce using a non-physical sound synthesis method. On the other hand, it also means that you, the user, need to learn how to play the instrument, just like a beginner...

4.6 Example 6: Reed Beating

An interesting feature of reed instruments has to do with the interaction between the vibrating reed and the mouthpiece. When playing at high amplitudes, the reed effectively collides with the mouthpiece, leading to a harsher, brighter sound. This may be controlled, as in Example 6, through the use of collision mechanism, which you may turn on or off. There is one parameter, which is rather artificial, but corresponds, roughly, to the stiffness or hardness of the mouthpiece.

You should compare the sound produced with the parameter on, and with it off; you should notice little difference except when the blowing pressure is high.

This is a rather tricky mechanism to use—under some conditions, especially when the beating parameter is high, you may hear a nasty aliasing phenomenon—try raising the beating parameter until you hear it. it is good to know this sound, so that you will know what is causing it if you hear it in your experiments...

4.7 Example 7: Variations in reed equilibrium displacement and tremolo

At the mouth, one source of control information, supplied by the user or player, is mouth pressure, a slowly varying function of time. Another, more subtle control, has to do with the amount of force being applied to the reed by the lips, changing, effectively, the equilibrium displacement of the reed—previously, this value has been set to a constant in the "Reed parameters" panel.

Variation in the equilibrium displacement makes for subtle changes in timbre or brightness; though you could vary it in any way you like, in the GUI, it is restricted to a tremolo effect, governed by two parameters, the depth A, and frequency f_T. as such, if the global equilibrium displacement is set to H_0, turning on these parameters amounts to an equilibrium displacement $H(t)$ of

$$H_0(t) = H_0\left(1 + A\sin(2\pi f_T t)\right)$$

Obviously then, A should be positive and less than 1, and the frequency f_T should be one which your mouth is capable of making (in order for the results to broadly realistic sounding), typically less than about 5 Hz.

Try experimenting with these parameters, to determine values for which the output is relatively realistic.

Actually, in a real-time environment, one would expect the reed equilibrium displacement $H(t)$ to come from an external source, such as a wind controller!

4.8 Example 8: The Tonehole

So far, the wind instrument model consists of a reed, coupled to a tube of a given length; there is, as yet, no mechanism for changing pitches. The necessary component is the tonehole.

A single tonehole in a wind instrument can be characterized by three parameters:

- Position of tonehole along bore (0–1)
- Radius of hole (in m)
- Depth of penetration of hole into bore (in m)

All three parameters have a rather large effect on the resulting sound. In example 8, a tonehole is positioned at a distance 0.8 of the way along the tube. Try listening to the sound produced, and compare it to the sound produced when the tonehole is turned off.

As a first step, try moving the tonehole closer to the bell, and away from the bell. What do you notice?

Then, reload Example 8, and try varying the tone hole size, and the depth; make sure that your parameters stay within a realistic range! (I.e., make sure that the radius of the hole is not bigger than the radius of the bore!

4.9 Example 9: Squeaks

When toneholes are in place, it becomes much easier to generate characteristic sounds of wind instruments such as, e.g., squeaks. Try Example 9, which makes use of a conical bore, and a (perhaps artificially deep) tonehole. Try varying the depth of the tonehole, to see for what characteristic depths the squeaks may be heard...

4.10 Example 10: Multiple Toneholes

One can, of course, add as many of these toneholes as one likes; in the GUI, one is limited to three, with position, radius, and depth specified for each. Try Example 10, and try varying the configuration to hear the sound when all possible combinations of the holes are on or off.

4.11 Example 11: Events

A third stream of control data is supplied not by the mouth but, obviously, by the fingers, in order to control events: openings or closings of the various holes. One way to characterize such events is as follows:

- Time of event (s)
- Hole number of event
- Final state of hole (0–1) where 0 is closed, and 1 is open,
- Duration of event (s)

One interesting feature here is that holes may be partially open—a value of 0.5 corresponds to a half-open hole. For the GUI, the initial state of any hole is set to be open (or 1)!

Example 11 yields a very simple sequence of notes; try varying the various parameters, as well as perhaps the positions of the holes.

5 Plates

So far the instruments that have been modelled, i.e., strings, bars, and tubes, have been essentially "1D." Some instruments, such as various members of the percussion family, as well as various components of instruments such as soundboards, and also reverberation devices, must be modelled as 2D objects with some stiffness—plates. The user interface plategui.m explores some features of plate synthesis, especially in a percussion framework.

Some of the features are common to the other algorithms, such as the sample rate, and duration. There is, as before, a timer, which you may use in order to benchmark your machine, or to gauge the effect of various parameters on computation time.

5.1 Example 1: The Ideal Plate

A thin plate, of the type found in musical applications, is characterized by the following parameters:

- density ρ

- Young's modulus E, which is a measure of stiffness
- thickness H
- Poisson's ratio ν, which is a special parameter appearing only in 2D, and which is usually equal to about 0.3.

Such constants describe the pointwise behaviour of the medium. In addition, there is now, in 2D, the geometry of the plate to contend with. For this simple application, only rectangular geometries will be considered, but one could design a more general synthesis routine without much difficulty. As such, two additional parameters are

- L_x, side length in x direction
- L_y, side length in y direction

As before, it turns out that this collection of six parameters is actually redundant in describing a plate; one may reduce these to the following two, assuming that $\nu = 0.3$, which is very reasonable!

- stiffness $\kappa = \sqrt{\frac{EH^2}{12\rho(1-\nu^2)L_x^4}}$
- aspect ratio $\alpha = L_x/L_y$

Beginning with Example 1, try experimenting with both these parameters, and try anwering the following questions:

- what is the perceptual effect of increased stiffness?
- why (do you think) that the algorithm returns an error as the stiffness is made very high?
- what is the perceptual effect of a change in the aspect ratio? To investigate this, try listening to the effect as you move from a square configuration ($\alpha = 1$) to a more oblong plate. Again, why does the algorithm return an error if the aspect ratio is made very high?
- how does computation time depend on stiffness?
- how does computation time depend on the sample rate?
- what is the effect of a reduced sample rate, perceptually? How can you explain this?

5.2 Example 2: Loss

As in the case of strings and bars, loss plays a very important role, especially in the present context of percussive sounds. As before, there are two loss parameters:

- 60 dB dcay time T_{60}
- high frequency loss parameter b

In Example 2, you may vary these parameters. The settings for T_{60} should be obvious; b, however, should be set to be rather small, say $b < 0.1$. b generally controls the decay rate for high frequencies.

Begin by setting $T_{60} = 8$, say, and listen to the effect on the sound as b is increased, slowly, from zero. You will note that the stimbre goes from being rather metallic, to more "wood"-like.

5.3 Example 3: Boundary Conditions

A feature which was not explored in the previous case of strings/bars, was the influence of boundary conditions on the resulting sound. The reason is that, at least in the case of strings, the ends are usually held fixed, though this is not true in the case of bars!

For plates, there is a great variety in the type of boundary conditions which may be applied; here are three simple ones:

- clamped
- simply supported (pivoting)
- free

The third condition is the most realistic, but is the most difficult to deal with, in that the plate itself is capable of "drifting away." For this reason, output under free boundary conditions undergoes, here, an extra high-passing operation.

Try playing with Example 3, for the three types of conditions, and listening to the effects on the resulting sound.

5.4 Example 4: Output

As before, for a time domain method, such as a finite difference scheme, it is easy to take as many outputs as one likes, by reading from distinct points on the computational grid. In Example 4, You may choose between mono and stereo output. You also have a (somewhat limited here) choice of where you read outputs; output is always taken from a point on the line of symmetry of the plate, and a value of radius between 0 and 1.

What is the effect on the choice of the listening points on the resulting sound? In particular, what is the difference in timbre as the listening points approach the oouter edge of the plate?

5.5 Example 5: Scanning

Also as before, it is useful, from a perceptual standpoint, to allow the output locations to very (slowly!); this is, in fact, an example of so-called "scanned synthesis." Though in general, one could allow the outputs to move in any desired manner, for the GUI, they are restricted to move along circular trajectories, at a given frequency. In addition, the frequencies may be "spread" by the ammount specified, to allow a further phasing effect.

Try listening to stereo output, and compare the difference in timbres generated when the scanning frequency is zero, and non-zero. In effect, such phasing effects are very much characteristic of instruments such as gongs, which may rotate slowly, relative to the listener.

5.6 Example 6: Events

There are various parameters which characterize a single perceussion event:

- Time of event
- location of event (x, y), with $0 \leq x, y \leq 1$
- Velocity

In Example 6, a sequence of three such events is indicated.

Try turning off all but the first event, and adjusting the location parameters, to listen to the difference in timbre. What happens if you strike near an edge? Also, what is the effect of increased velocity for a single strike?

An additional parameter is the mallet radius, also between 0 and 1. Adjust this, and listen to the resulting change in timbre.

5.7 Example 7: Traps

Just as in the case of the prepared string, it is possible to add additional elements to a plate synthesis algorithm. Consider, for example, spring-like elements, or traps, attached at various points on the plate surface. In addition, one might allow the springs to behave nonlinearly—so that they are stuiffer at high amplitudes than at low amplitudes.

One of the difficulties of using nonlinear elements, in a finite difference synthesis framework, is the danger of numerical instability—sometimes, the computed solution can diverge, and the algorithm will crash. In general, the danger increases with increased stiffness of systems. For Example 7, if the algorithm quits, start it again, after reducing stiffness.

For simplicity, in the GUI, the positions of the traps are randomly generated, but you may choose the number of such elements. You may also choose the stiffness. There are two predominant effects you may notice:

- changes in pitch
- a dramatic incrase in high frequency energy (a crash)

Both such effects are characteristic of instruments such as gongs, though in this case, it should be noted that the mechanism is a wholly fanciful one, and not a rigorous model of the nonlinear (high-amplitude) dynamics of a plate. (This can be done rigorously!)

First, try reducing the stiffness of the traps; you will note that the strength of the pitch shift is lessened. Then, for the sake of comparison, turn the traps off altogether (you may set the number of traps to 0), and listen to the difference on timbre compared to the linear plate.

Pitch shifts are strongest under fixed boundary conditions—the situation is somewhat like that of steel drum panels, or a gong with a heavy rim. In order to generate crash-like sounds, you need to use a plate with a free boundary (just like a cymbal). Try listening to the difference in timbre when you make this change.

5.8 Example 8: Rattles

Rattling elements are also a possibility. Again, these may be chosen at random locations over the plate, with the following parameters:

- number of rattles
- stiffness
- mass relative to plate
- length

Again, you will note that under some conditions, the algorithm will become unstable—if this happens, reduce the stiffness (between 500 and 3000 are good values), or the mass ratio (generally should be less than 1). Try experimenting with these parameters.

You may, of course, combine the rattles with the traps—try generating some more complex sounds, if you like.

6 MATLAB tutorial

Matlab as Calculator

Matlab is, at its most basic level, a calculator. For instance, if you type

>> 3+3

at the prompt, and press return, matlab tells you

ans =
6
>>
and is ready to accept another command. The operators

```
*, /, ^
```

refer to multiplication, division and raising to an exponent, respectively. Try a few basic calculations using these operators. Parentheses () are used with the same implications as in normal algebra. I.e.,

```
2^(3+1) and 2^3 +1
```

give different answers. When in doubt, use parentheses. You can type "help ops" for a list of all the operators in matlab.

, and ;

It is possible to perform two calculations on the same line, by separating commands by a comma ",". **Try finding the sum of 3 and 4, and the product of 2 and 3 using a pair of commands separated by a comma.**

The semicolon ";" is used to suppress the output of matlab; matlab will perform the command you ask silently, without telling you. **Try finding the sum of 3 and 4, and suppressing the output.** This output suppression feature is extremely important, especially when you are working with large arrays (such as audio signals). If you don't suppress output, you may be facing several minutes worth of numbers flying by on the screen.

Built-in values

Certain well-known constants are built in to matlab. For instance, if you type
>> pi
matlab will say
ans =
3.1416

Other important constants are also built in. For instance, **try typing "i" or "j" at the prompt.**

Basic Functions

Advanced calculators possess a variety of functions. In matlab, for example, we have the functions sqrt(), sin(), cos(), etc. which behave as expected. Parentheses are essential here, to distinguish what is "inside" the function to what is "outside." **Try finding the sine of π.** Functions may also be embedded within one another: **Try finding the square root of the cosine of $\pi/4$.** Type "help elfun" for a list of matlab functions.

Variables

Matlab begins to diverge from your typical dumb calculator when you begin to make use of *variables*. You can name these at will, with the sole constraint that you don't pick a name which matlab already uses for something else (such as "sin") for example. For instance, if you type
>> a = 3
and then press return, matlab will say
a =

3
so the variable a now has the value 3. If, at any subsequent time, you type "a" at the command line, you will get the same response. The values of variables may be reassigned at any time; if, for instance, you now type

>> a = 2

and then press return, matlab will say

a =

2

Variables may be treated just like other numbers. **Try typing "a*2" at the command line.** They may also be combined to evaluate more involved algebraic expressions. **Try assigning the value of** $pi/2$ **to a variable a, and the value** $\sin(pi/2)$ **to the variable b. Then find the sum of a and b.**

Until you clear them, these variables remain in matlab's memory. You can list all the variables currently active by typing "who." (**Try this. You should see the variables "a" and "b" which you have been using**.) You can get rid of these variable easily, by typing "clear" and then the variable name. **Clear the variable a, and type who.** You can clear all the variables by typing "clear all." **Do this, and type who.**

Vectors

A very important part of matlab centers around its ability to handle sets of numbers as units. The simplest such unit is the vector, which is simply a list of numbers. These vectors are important for us, as they will represent audio waveforms, and thus sounds. To create such a unit, simply type an open square bracket "[" followed by a list of numbers separated by commas, and then a closing bracket "]". For instance, you may create a vector containing the three numbers 5, 6 and 7 by typing

>> a = [5,6,7]

at the command line. (Notice we have also assigned to it the variable name a.) **Create this vector, and then type "a" at the command line, and look at what matlab gives you.** Individual elements of the vector are accessed by the syntax "a(n)" which n refers to the nth element of the vector a. **Type a(2) at the command line. Find the sum of the squares of all the values in the vector a, and then take the square root of the sum. (This is known as a norm, and we will treat this later in the course).**

Vector Operations

Vectors may be handled just like numbers; operations applied to a vector will sometimes (in this module) be applied to each of the elements of the vector simultaneously. **Create a vector containing the values 1,2, and 3, and name it "b." Now perform the operation a+b, using the vector a created above.** The same thing applies to multiplication, division and exponents, but the syntax is slightly different; we need to use the operators

`.*  ./  .^`

The dot "." means that the operation is performed on an element-by element basis. So, in order for a multiplication or division of two vectors to make sense, they must have the same number of elements. **Try finding first the product, then the quotient of the values in the vectors a and b.**. Functions such as sin(), cos(), sqrt() etc, may be applied to the vectors and again operate on each element of the vector separately, giving another vector as an output. **Try finding the sine of a.**

Creating Vectors

One way of creating a vector is, as we showed above, by simply typing the values directly on the command line. If the vector has a simple structure, there is an easier way of proceeding. Suppose, for example, we

wish to create a vector which contains the integers from 1 to 10. We could type >> a = [1:10]
Try this, and check the output. The colon ":" in this case is what produces this list. It is possible to go further; suppose we wish to create a vector which contains all integer and half-integer values between 1 and 10. Then we could type >> a = [1:0.5:10]
Try this. The 0.5 controls the step size between adjacent elements. **Create a vector, named t, which contains all values between 0 and 1, separated by steps of 1/44100. This could serve as a list of time instants between 0 and 1 second for a digital system sampled at 44.1 kHz.**

A pair of special vectors are those which are made up of all zeros or all ones. A vector of zeros of length n is created by typing "zeros(1,n)" and a vector of ones of length n is created by typing "ones(1,n)". **Create a vector containing five zeros, and name it z.**

Sinusoids

The command

```
>> t = [0:1/44100:1];
```

creates of vector of numbers between 0 and 1, with intervals of 1/44100 between adjacent numbers. **Create this vector; How many values should be contained in this vector? (I.e., what is the vector size?)** For audio purposes, you can think of this vector t as a list of time instants between 0 and 1 seconds. A sine wave, say at 4 Hz and of amplitude 2, is easily created from this by typing

```
>> s1 = 2*sin(2*pi*4*t);
```

We've named this new vector s1. **Create this vector, and also create two more sinusoids, one of amplitude 1 and frequency 2 Hz, (called s2) and another of amplitude 4, and frequency 1 Hz (called s3)**.

Plotting

Suppose now that we would like to take a look at these vectors. This is easily done, by typing

```
>> plot(s1)
```

Do this. A new window, entitled "Figure No. 1" will open, with a plot of the values in s1 versus the index numbers of the vector (i.e., 1 to 44 101). This is a little unsatisfying, as we'd really like a plot of amplitude versus time. This can be done by typing

```
>> plot(t, s1)
```

This new plot replaces the old one, and now the horizontal axis runs from 0 to 1. **Do this.**

Listening to Waveforms

It is quite straightforward to hear waveforms in matlab. For instance, suppose you have created one second worth of a sinusoid, amplitude 100, frequency 800 Hz, at a sample rate of 44 100 Hz. **Do this, and name it s1.** Then type, simply,

```
>> sound(s1, 44100)
```

Here, the function sound simply plays the vector s1 at the sample rate you tell it to play it at. **Do this.** Note, however, the harshness of the resulting sound; the problem is that the sinusoid s1 has an amplitude of 100, but audio waveforms must take values between -1 and 1, otherwise they will be clipped. Luckily, there is another function, called soundsc, which automatically scales any waveform to lie within this range. **Play the same vector s1 using soundsc. Also, try playing the sound back at a different sample rate, say 22 050 Hz.**

Real Waveforms

matlab has a few waveforms stored in memory, just as examples. One is a sample of the Hallelujah chorus. **Type "load handel", after having cleared the workspace. Now type "whos," to show a list of**

all variables in memory. There are two elements now in matlab's memory, one called "y," and the other called "Fs." **To see what these are, type first "y," then "Fs."** y is simply the waveform itself. **To take a look at it, plot it.** Fs is the sample rate it was recorded at. **Play the sound back using "soundsc," at the correct sample rate. Then play it back at 44 100 Hz.**

.wav files

suppose you create a sound in matlab, and would like to output it as .wav file. There is a function for this purpose, called "wavwrite." **Look up the help on this function.** Wavwrite is a function which takes four arguments: first the file you wish to write (i.e., some list of numbers), the sample rate, the number of bits on which it is to be encoded, and the name of the output file, to be enclosed in single quotation marks. **Create one second worth of a sinusoid, of frequency 440 Hz, amplitude 1, at a sample rate of 44 100 Hz. Write it as a .wav file, called test.wav, at 16 bits.** The file has been created in the present working directory. **To see it, type ls**.

Now clear the workspace. The sinusoid is gone from matlab's present memory, but the .wav file is in the working directory. In order to read it into matlab, you may use the command "wavread." **Look up the help on this function.** wavread takes, in its simplest form, a single argument, just the name of the file you wish to read in, in single quotes. It produces an output vector of numbers. **Type y = wavread('test.wav'), and then type whos.** y is now in matlab's memory as a list of numbers, and can be played as a soundfile. **Do this.**

Scripts

So far, everything we have done has been at the "command line" in matlab, i.e., at the >> prompt. This is ok for simple calculations, but ideally, for most applications including sound processing, we will have to string together a large number of separate commands. matlab, in addition to being a glorified calculator, also functions like a computer language such as C, C++, Basic, etc. It is possible, of course, to write full programs (called scripts), and the nice thing is that the language has a very high-level structure (the downside of this is, of course, that programs don't run very fast in matlab, which is the main reason why matlab is used as a prototyping language, and not for real applications).

Setting up a script is very simple. Go to the "File" menu of the command window, and select New¿Mfile. A blank window will open up, which you can fill with matlab commands, just as though you were typing them into the command window. Lines are separated, generally, by semicolons, unless you want output to be printed. These scripts are also called "m" files, since the suffix for such a file will always be ".m". In order to save an m file, simply go to the file menu for your new script, and choose "save as." You may then name your file (and be sure to put it in an easily accessible directory, as matlab can only "see" files that are in its pwd.

You can then run the script by typing the name of the script at the command line (without the .m suffix).

Loops

Try, if at all possible, to avoid using standard loop constructions in your matlab code; they are highly inefficient. If you really must use them (and sometimes they are inevitable), it is possible to use for and while loops, as well as if statements. For loops are pretty much the same as in C. For instance, the series of commands

```
for n=1:N
x(n) = sin(2*pi*f*n/Fs);
end
```

fills a vector x (you need to define it first) with N values of a sine function, at frequency f, and at sample rate Fs.

See the matlab help for the commands.

7 Programming a Simple Harmonic Oscillator

The simple harmonic oscillator, defined by

$$\frac{d^2x}{dt^2} = -\omega_0^2 x$$

where $\omega_0 = 2\pi f_0$, has been discussed in lecture; a simple finite difference scheme (recursion) is as follows:

$$x_n = (2 - \omega_0^2 T^2)x_{n-1} - x_{n-2}$$

where $T = 1/SR$ is the time step. You will program this to produce sound output.

Setting up an .m file

In your pwd, create an m.file called "sho.m."

Global simulation parameters

In a synthesis algorithm, there are various parameters which must be specified before run time. In this case, they are:

- SR (sample rate, in Hertz)
- $f0$ (oscillator frequency, in Hz)
- Tf (simulation duration, in seconds)
- $x0$, initial position of oscillator, in m
- $v0$, initial velocity of oscillator, in m

In the preamble of your code, create Matlab variables corresponding to these variables, and set: $SR = 44100$, $f0 = 200$, $Tf = 3$, $x0 = 1$, $v0 = 1$.

Derived Parameters

From the global set of parameters, you may, for the sake of programming similcity, like to define some subsifdiary parameters. here are a few:

- $T = 1/SR$ (time step, in seconds)
- Nf (duration of simulation, in samples...figure out how to find this number, and make sure it is an integer; learn about the command "floor" for this purpose).
- $x1 = x0 + Tv0$, the position of the oscillator at the second time step.
- $w0 = 2\pi f0$ (angular frequency of oscillator

Add these parameters to your code after the global set.

Oscillator State

Now, you need to create a vector x, of length Nf samples, which will ultimately be the solution to the oscillator, and your sound output. It should be initialized to values of zero, except for the first value, which will be x0, and the second, which will be x1. Create this vector.

Recursion: setup

Set up a "for" loop, running over the index n, for n=3 up to Nf. This will be used to calculate the as yet unknown values of the vector x (notice that its values at n=1 and n=2 are already set by the initial conditions!).

Implementing the recursion

Program the recursive step, so that x_n is calulated from x_{n-1} and x_{n-2}.

Plotting the output waveform

Plot the vector x against a vector of time instants t, which should also be of length Nf.

Listening to Output

Listen to the resulting waveform using the command soundsc; make sure you have used the correct sample rate!

Features: Instability

For a given sample rate SR, you cannot generate a solution for any value of $f0$, the oscillator frequency. What you will find is that your solution will become numerically unstable...this will be easily visible in your plots. Try raising the frequency $f0$ until this occurs, and see if you can find, for a given sample rate, exactly the oscillator frequency $f0$ at which this occurs. Try this for a variety of sample rates, such as $SR = 44100$, $SR = 32000$, $SR = 22050$, etc. Can you find a rule giverning this behaviour?

Features: Numerical Dispersion and Detuning

One of the nice things about the SHO is that you can generate an exact solution, for comparison! (This is almost never true in more complex settings, though, which is the whole point of using finite difference schemes!). In fact, the exact solution is:

$$x(t) = A\cos(2\pi f_0 t) + B\sin(2\pi f_0 t)$$

where $A = x_0$, and $B = v_0/\omega_0$.

In your code, create, first, a vector t, containing the values of time at the instants $t = 0$, $t = T$, $t = 2T$, etc. up to $t = (Nf - 1)T$. There will be Nf of these. Use this vector in order to calculate an exact solution vector.

Then, run the code, and listen to both your computed finite difference solution, and the exact solution. Try working at a low sample rate. such as $SR = 8000$, and compare outputs, perceptually (i.e., using your ears) as the frquency f_0 is increased.

The detuning effect is fundamental to finite difference methods; especially at high frequencies, they lose accuracy...and this effect can be audible!

Features: Loss

The SHO is a canonical model of all lossless systems; finite difference schemes for oscillating systems can always be computed using an algorithm similar to the above.

One feature which is not modelled, hoever, is loss. You mat add this to the differential equation as follows:

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - 2\sigma\frac{dx}{dt}$$

Here, σ, the loss parameter, is related to T_{60}, the 60 dB decay time, as

$$T_{60} = \frac{6\ln(10)}{\sigma} \tag{1}$$

Using the information given in the lecture on finite difference schemes, try to develop a finite difference scheme for this lossy oscillator. Listen to the result, and plot it.