MAS.110 Fundamentals of Computational Media Design
Fall 2008

CsoundXO

A Manual for Developers of

Activities

for the

One Laptop Per Child XO

who wish to use

Audio and Musical Sounds

in their applications

Version 1.01

Barry Vercoe
Media Lab
MIT

Nov 4, 2008

PART ONE: THE PYTHON API ENVIRONMENT

Contents

1. A Beginning Tutorial

Two Cooperating Worlds

Csound on the **XO laptop** consists of two parts:

1. a User Interface written in **Python**, with real-time interaction and graphics support
2. a **Csound** Audio-processing Engine, pre-compiled and written in the language **C**.

This introduction shows how the world of a Csound Orchestra/Score performance can be imbedded in the world of Python, and how the two worlds can interact in real-time applications.

Example 1. The following **Python** program contains an *orchestra* and *score* as imbedded strings.

```
import CsoundAPI
from time import sleep

orchestra = '''                         ; begin an orchestra string within triple quotes
        sr = 32000                      ; the audio signal sampling rate
        kr = 1000                       ; the control signal rate
        ksmps = 32                      ; the number of audio samples in a control period
gisine  ftgen   0, 0, 1024, 10, 1       ; draw a sine wave in 1024 words using Gen 10

        instr   1                       ; define instrument number 1
a1      oscil   8000, 440, gisine       ; make a continuous sine wave, amp 8000, freq 440 Hz
        out     a1                      ; send this sine wave a1 to the audio output device
        endin

        instr   2                       ; define a second instrument
a1      oscil   8000, 550, gisine       ; as above, but at frequency 550 Hz
        out     a1
        endin
 '''
score = '''                             ; now begin a score string (of what to play and when)
i1  0  3                                ; play instrument 1 at time 0 for 3 seconds
i2  5  3                                ; play instrument 2 at time 5 for 3 seconds
e                                       ; end of the score
'''
address = "localhost"
port = 7000                             # define some parameters
                                        # create a connection socket, and call it csound
csound = CsoundAPI.CsoundAPI(address, port, is_gtk=False)
csound.newInstance()                    # allocate a Csound instance at the end of this socket

csound.sendOrcStr(orchestra)            # send our orchestra string to this new instance
csound.sendScoStr(score)                # also send it our score string
csound.sendPrep()                       # tell the instance to load both and get ready to play
csound.sendPlay()                       # tell the instance to start playing
sleep(10)                               # take a rest while the instance is performing
csound.close()                          # and we are done
```

Example 2. This is similar to the above, but with the addition of *control signals* in the orchestra and additional *parameter fields* in the score. This will draw attention to Orchestra Syntax.

```
import CsoundAPI
from time import sleep

orchestra = '''
        sr = 32000                      ; audio sampling rate
        kr = 1000                       ; etc …
        ksmps = 32                      ;  … all making up the Orchestra Header section
gisine  ftgen   0, 0, 1024, 10, 1

        instr   1                       ; instrument 1
a1      oscil   p4, p5, gisine          ; amp and frequency now derive from score p-fields
        out     a1
        endin

        instr   2                       ; instrument 2
k1      lfo     2, gisine, 3            ; low frequency oscillator:  freq 2 Hz after 3 secs delay
a1      oscil   p4, p5 + k1*10, gisine  ; add k1 (a control rate signal) to the score frequency
        out     a1                      ; and listen to the result
        endin
 '''
score = '''
i1  0  3  8000  440                     ; specify the amp and frequency in parameters 4 and 5
i2  5  5  8000  550
e
'''
        … the remaining Python part is the same as Example 1.
```

The Orchestra Syntax is quite simple, each line consisting of three main parts:
1. Result cell 2. Opcode (verb) defining the operation 3. Input parameters separated by commas

Result cells can be of 3 kinds: I-time cells, K-rate cells, Audio-rate cells – where each kind is identified by the first letter of its name, 'i', 'k', or 'a'. These cells are locally scoped within the instr/endin delimiters, meaning that *a1* of instrument 1 is different from *a1* of instrument 2, and *k1* of instrument 2 is unknown to instrument 1. There is however an additional qualifier 'g' that can make any cell *global* and thus visible to all instruments. We see an example of this in the orchestra Header statements, where the *isine* output name from *ftgen* becomes a global *gisine*, and is thus reachable by all instruments.

Note: We are not required to imbed orchestra and score as strings in the Python control program. They can be stored as named files within the local file system. If we were to store an orchestra and score in a subdirectory X, then the commands sendOrcStr(name) and sendScoStr(name) would be replaced by sendOrcNam("X/newname.orc") and sendScoNam("X/newname.sco") where X is the pathname of a directory of files. Although strings imbedded in Python script are nicely interactive, larger orchestras and scores are often best maintained as independent files, called up by Python whenever needed.

Example 3. This shows how Python can modify the orchestra in the middle of a performance. The vehicle here is the **alias** attribute, which serves to expose an orchestra variable to the outside world. *Result* variables (above), whether local or global, are known only *within* the orchestra

since the Prep-time language translator and loader will hide them away in internal memory. Csound's **alias** directive allows Python's **getAlias** and **setAlias** to read and write to that memory at any time.

```
import CsoundAPI
from time import sleep

orchestra = """
        sr = 32000                      ; audio signal sampling rate
        kr = 1000                       ; the control signal rate
        ksmps = 32
gisine  ftgen   0, 0, 1024, 10, 1
gkfrq1  init    440                     ; create two global orch cells, initial values 440 and 550
gkfrq2  init    550
        alias   gkfrq1, "inifreq1"      ; expose them to the outer Python program
        alias   gkfrq2, "inifreq2"

        instr   1                       ; instrument 1
k1      lfo     3, gisine, 5            ; low frequency oscil:  delay 5, then a 3 Hz vibrato
a1      oscil   p4, gkfrq1+k1*9, gisine ; initial gkfrq of 440 (+ 0) is modifiable by Python
        out     a1
        endin

        instr   2                       ; instrument 2
k1      lfo     3, gisine, 5            ; low frequency oscil:  delay 5, then a 3 Hz vibrato
a1      oscil   p4, gkfrq2+k1*8, gisine ; initial gkfrq of 500 (+ 0) is modifiable by Python
        out     a1
        endin
 """
score = """                             ; score (of what to play when)
i1  0  7  8000                          ; play instr 1 at time 0 for 7 seconds
i2  0  7  8000                          ; play instr 2 at time 0 for 7 seconds
e                                       ; end of the score
"""

address = "localhost"
port = 7000                             # define some parameters
cs = CsoundAPI.CsoundAPI(address, port, is_gtk=False) # create a Connection Socket, call it cs
cs.newInstance()                        # allocate a Csound instance at the end of  this socket
sleep(1)
cs.sendOrcStr(orchestra)                # send it our orchestra string
cs.sendScoStr(score)                    #   and our score string
cs.sendPrep()                           # load these both and get ready to play
cs.sendPlay()
sleep(2)                                # after 2 seconds of hearing 440 and 550 Hz
cs.setAlias("inifreq1", 366)            #   change the 440 Hz to 366
sleep(1)                                # after 1 more second
cs.setAlias("inifreq2", 619)            #   change the 550 Hz to 619
sleep(5)                                # listen for the vibrato to begin after 2 more secs
cs.close()
```

Since it would be very tiresome to score all music using only frequencies, there are several tools that use note names, pitch classes, and MIDI codes, as we'll see later. Meanwhile we will next look more closely at the wide range of Python commands that drive a Csound Orchestra.

2. The Python - Csound API

Csound is controlled and managed via a set of Python Bindings which have entry into a running Csound environment. When Csound is first launched it passes control to the **Csound Server**, which will open a socket on a fixed port and then wait for a Python program to do the same:

Csound server awaiting Python connection …

If Python is then launched with the same port number, the two will establish a formal connection.

This is a connection to a base *instance* of Csound called the *Csound Server,* which resides on every machine that has a valid Csound executable. A *Csound Server* is responsible for basic functions such as talking to that machine's file system and sending audio to its external devices. The Server instance does not do any sound generation itself, but delegates this to other allocated *instances* on that machine.

All allocated instances are created at the request of Python which can continue to send commands (via the Server) to each individual instance. Python can also tell these instances to communicate with each other. In this way a Python host can serve either as a simple Master, or as a Configuration Editor that defines the topology of cooperating Csound instances on the same or different machines.

Communication between Python and a *Csound Server* is via one or more *Connection Sockets*. Each socket is retained by the Server in a Client structure that remains intact while Csound instances come and go. A *close* command to a socket will remove that socket and any instance it has been supporting. A *serverShutdown* command will remove all current sockets and shutdown the Csound Server itself.

Most API commands have arguments, either a character string or a numerical value, passed to the target Server. Some API commands also request some form of *response* that is returned to a user-defined *callback* method. This is user-defined in Python, and may be a simple *print* or a more complex *plot* method. For plot and other graphic methods, enable the PyGTK graphics package by setting *is_gtk=True* in the following command.

cs = CsoundAPI.CsoundAPI(address, port, is_gtk=False)
set up a *connection socket* to the addressed Csound Server and give it a unique *name* (here "cs"). A Python host program may set up several Connection Sockets to the same or different Servers, each uniquely addressable by that *name*. Servers may be on the same or different machine as Python, their location being specified by the *address* argument.

cs.newInstance()
allocate a new Csound *instance* that can be reached through the named Connection Socket. Since a socket can address only one *instance*, the socket name can also be considered the name of the Csound *instance*. Each allocated *instance* is initially empty, and its performance will need an Orchestra plus a Score or Midifile sent via the named socket. If a *newInstance* command is sent more than once on the same socket, a fresh *instance* will be allocated and the previous one destroyed. See also **sendDestroy**.

cs.setOrcNam("…")
tell this instance it has access to an orchestra file named "…". May need a complete path name.

cs.setScoNam("...")
tell this instance it has access to a score file named "…". May need a complete path name.

cs.setMidiNam("...")
tell this instance it has access to a Midi file named "…". May need a complete path name.

cs.sendOrcStr(strname)
send the Python-defined string *strname* to serve as the orchestra for this instance. Every instance must have an orchestra, delivered either as a string or a file name.

cs.sendScoStr(strname)
send the Python-defined string *strname* to serve as the score for this instance. Having a score is optional, since orchestra instruments can be invoked by realtime LineInput events (below).

cs.setAlias(aliasname, value)
locate the orchestra variable that has been given this *aliasname*, and send it this new *value*.

cs.getAlias(aliasname, callbackname)
locate the orchestra variable with this *aliasname* and report its current value back to Python. This command requires a user-defined *callback* method that will handle the response.

cs.setVolume(value)
set the output Volume level of this instance to *value*. The default value is 1.0 (no level change).

cs.getVolume(callbackname)
report the current setting of Volume in the named instance. Requires a *callback* method.

cs.setMasVol(value)
set the Master Volume level to *value*. This affects all instances. The default value is 1.0

cs.getMasVol(callbackname)
report the current setting of the Master Volume. Requires a *callback* method.

cs.setTempo(value)
set the performance Tempo of the current score to *value*. In the absence of a Tempo statement in the score, the default tempo is 60 beats / minute. i.e. scores measure time in seconds by default. This command will override that and perform the score at this new tempo. When the score has been pre-processed by a Tempo statement, both of these time-warpings will be in effect.

cs.getTempo(callbackname)
report the current performance tempo of the current score. Requires a *callback* method.

cs.setMidiSpeed(value)
set the Midi performance speed to *value*. A Midi file has an internal tempo which is often hard to assess. This command will *modify* that tempo by the factor *value*. The default factor is 1.0

cs.getMidiSpeed(callbackname)
report the current speed factor in this Midifile performance. Requires a *callback* method.

cs.setMsgLevel(value)
set the instance msg level to *value*. All printf statements in Csound are accorded an importance factor, with Network and Error messages being highest and general info and output amplitudes being lowest. Low level amplitude messages can be useful during instrument development, but

during real-time performance of well-formed orchestras this detail will steal CPU cycle time. The Message level is set by a bit-wise OR of unique binary values (here in decimal format):

Network messages	1
Error messages	2
Warning messages	4
Performance errors	8
API messages	16
Orchestra messages	32
General information	64
Note segment amplitudes	128

The default message level is the bit-wise OR of all the above, which is 255. A lower message level of say 63 will show instrument allocations but not segment amplitudes, etc. A message level of 7 is a good one to work with. Use 0 when you are certain your orchestra is behaving perfectly.

cs.getMsgLevel(callbackname)
report the current Message level for the current instance. Requires a *callback* method.

cs.setDisplays(value)
set the bit that causes stored waveforms to be displayed on creation. The value 1 = ON, and the value 0 = OFF. The default value is 1 (displays ON).

cs.setGraphics(value)
recast graphic displays into ascii characters (which will work on all terminals). The value 1 = ON, the value 0 = OFF. The default value is 1 (ascii graphics ON).

cs.sendBeat()
send a Beat event to the current performing score. This event will cause all score times to be interpreted as Beats. Using a rhythmic succession of these commands the user can effectively "conduct" a performance. Sometimes in real conducting the "beat" may be subdivided into groups of 2 or 3, or sometimes clustered into higher groupings of 2 or 3. This input interpreter will soon figure out if you are doing that, and respond accordingly.

cs.enterBeatInputMode()
suspend normal ascii input on your terminal and interpret all keystrokes as conductor Beats (as in the above). Type 'q' to exit from this mode.

cs.sendLinevt("...")
send a single score event from the terminal. Any scoreline appropriate for a running orchestra can be input as a realtime event. For instance in Example 1 above, the input

```
csound.sendLinevt("i1 0 3")
```

would immediately duplicate the first note played by the score, meaning that an entire score could be replaced by sendLinevt commands. Note however that only a 0 in pfield 2 (start time) will launch a note immediately. The input

```
csound.sendLinevt("i1 3 5")
```

will tell the orchestra to generate this event 3 seconds from NOW (you will hear this). The duration (given by pfield 3) will be measured from whenever this event actually occurs.
Line events are stored inside Csound in a Linevent buffer, from which events are retrieved in time-sequenced order. The Linevent buffer can be cleared by the **clearLines** command (below). [Note however that Line events can also originate from *within* a Csound instance (or Equivalent) and be redirected to *another* instance or Equivalent with whom it has privileged communication (see the IIS server and IIC client sockets below).]

cs.enterLineInputMode()
suspend normal Python input on your terminal and interpret all lines as Line events (see above). This is a more efficient means of input than the above, and the same rule for pfield 2 will apply. Type 'exit' to exit from this mode.

cs.clearLines()
clear the current Linevent buffer.

cs.setUploadBeat(value, callbackname)
tell the current instance to notify the Python host each time new a Beat is encountered in the performing score or midifile. For score with simple beats, set *value* to 1; for scores with compound beats (eg midifiles sometimes encode triple rhythms in groups of 2) set *value* to 1.5 or some other suitable number. Presumes a Python callback method on what to do on receiving an uploaded Beat signal.

cs.sendSVOpen ()
tell the current instance to open an Inter-Instance Server (IIS) socket, allowing it to accept connections from the IIC sockets (see below) of one or more external instances (or equivalent) who wish to communicate privately through a privileged socket. IIS and IIC sockets are separate from Python communication sockets. They are Csound-to-Csound or Csound-to-Equivalent *privileged* communications which do not go through the Python environment, but carry out high-speed data transfers between instances and thus avoid the time-jitter that Python can induce. An IIS socket can accept more than one IIC communication link, and their interleaved arrival messages will be dispatched accordingly by Csound. The links are full duplex (two-way).

cs.sendCLOpen(IPadrs, instance)
tell the current instance to open an Inter-Instance Client (IIC) socket, and make a privileged connection to an IIS SVOpen socket at the indicated address. A single Csound instance (or its equivalent) can open several IIC sockets to different external instances, provided that each external server module activates an IIS SVOpen socket that will accept the connection. The remotely paired SVOpen and CLOpen can be activated in any order, each waiting on the other.

cs.setInsRemote(insno, IPadrs, instance)
tell this instance that any note event meant for instrument *insno* is to be sent to the Csound instance (or equivalent) at IPadrs for performance. This command automatically includes a CLOpen and IIC socket that requires the target instance to have a matching SVOpen and IIS socket (see above).

cs.setInsGlobal(insno)
tell this instance that any note event meant for instrument *insno* is to be sent to *all* Csound instances with whom the current instance has SVOpen IIS communication (see above).

cs.setMidRemote(chnum, IPadrs, instance)
tell this instance that any midi command meant for channel *chnum* is to be sent to the Csound instance (or equivalent) at IPadrs for performance. This command automatically includes a CLOpen and IIC socket that requires the target instance to have a matching SVOpen and IIS socket (see above).

cs.setMidGlobal(chnum, IPadrs, instance)
tell this instance that any midi command on channel *chnum* is to be copied to *all* Csound instances with whom the current instance has SVOpen IIS communication (see above).

cs.setAudioRemoteSI(IPadrs, instance)

tell this instance to copy its audio output to the Input Stream of the Csound instance at the IPadrs. This command automatically includes a CLOpen and IIC socket that requires the target instance to have a matching SVOpen and IIS socket (see above). The incoming audio can be accessed by the receiving orchestra by its **in** statement, as in

```
a1      in
```

cs.setAudioRemoteSO(IPadrs, instance)
tell this instance to *add* its audio output to the Output Stream of the Csound instance at the IPadrs. This command automatically includes a CLOpen and IIC socket that requires the target instance to have a matching SVOpen and IIS socket (see above). The incoming audio is not accessible by the receiving orchestra, but will be audible in its output.

cs.setXtraScoreEvents(extras)
cs.setXtraMidiNoteOns(extras)
cs.setXtraMidiNoteOffs(extras)
cs.setXtraMidiEvents(extras)
cs.setXtraMidiMessages(extras)
cs.setXtraSysexMessages(extras)
cs.setXtraAudioBufsSI(extras)
cs.setXtraAudioBufsSO(extras)
tell this instance to send *additional* copies of remote-targeted events or messages. This is to ensure that, under conflicting network conditions, the messages will arrive. Each message carries an index number so that when 2 or more actually arrive only the first is placed into execution by the receiver.

cs.setIOBufFrames(value)
set the audio I/O buffer size. Audio devices routinely buffer sent/received samples into mid-size blocks. Large is efficient, small is most interactive but there is minimum. The default is 1024 sample frames.

cs.setBufTimer(period, callbackname)
set a Csound timer to respond every *period* seconds. The period is measured in *audio buffer* periods, typically around $1/60^{th}$ of a second, and has that resolution of time. Presumes a host callback method.

cs.setP2Timer(period, callbackname)
set a Csound timer to respond every *period* seconds. The period is measured in orchestra *K-periods*, typically around $1/500^{th}$ of a second, and has that resultion of time. Presumes a host callback method.

cs.setOMaxLag(val)
set the Audio delay time in seconds that will synchronize Beat input with audio output.

cs.setInName(str)
set the audio input name. The name ‘adc’ will initiate reads from the audio device, any other name from a file by that name.

cs.setOutName(str)
set the audio output name. The name ‘dac’ will inititate writes to the audio device, any other name to a file of that name.

cs.setIntAudio()
tell the Orchestra translator to install integer audio data paths. This command must *precede* the sendPrep command (below). Csound normally does 80% of its calculations using Floating Point

arithmetic. While this works well on large expensive CPUs, it can sometimes soak the processor in the XO. Also since the FP Unit consumes more electricity than the fixed-point component, it can run down your battery faster. Fixed-point processing in Csound will run about 60% faster than floating, and will produce equally high fidelity sound for most opcodes, introducing slight distortion for just a few. So for conservation-minded computing, try inserting the setIntAudio command before you type sendPrep.

cs.sendPrep()
ask your Csound instance to translate and load the current Orchestra, to time-sort the events for any Score present, and to initialize any Midifile it has been given. These operations do extreme error checking, so that if either your orchestra or optional score cannot be found or have syntax errors, then sendPrep will return an error condition and the instance will not be able to play.

cs.sendPlay()
start the current instance performing its orchestra (presuming the sendPrep was error free), and play to the end of the entire score. This can be interrupted by Pause followed by Resume, after which it will continue on to the end.

cs.sendPlayForTime(value)
this operates as the above, but will stop (i.e. Pause) when the given time has elapsed. On an ensuing Play or Resume it will then pick up from where it left off.

cs.sendPause()
pause the current instance in mid-performance.

cs.sendResume()
resume a Paused performance from where it left off.

cs.sendPlayAll()
send a Play command to all Orchestras that are successfully Prepared. This enables multiple Orchestras to start playing in sync. The audio output of all instances are added into a single output stream, which may result in amplitudes exceeding the maximum of 32,767. If this is a problem, adjust each individual orchestra with setVolume, or adjust the sum with setMasVol. If at any time this multi-instance performance encounters a PauseAll followed by ResumeAll, it will then continue each orchestra to its end. Note that a single orchestra within the set can be individually Paused and Resumed.

cs.sendPlayAllForTime(value)
this operates as the above, but will stop (ie PauseAll) when the given time has elapsed. The rules for single or grouped Pause and Resume still apply.

cs.sendPauseAll()
Pause all instances of a multi-instance performance.

cs.sendResumeAll()
Resume a multi-instance performance, even if they were Paused in different ways.

cs.sendRewind()
stop and rewind the current score to its beginning, then Pause for a Resume or Play command.

cs.sendRewindAll()
stop and rewind all multi-instance scores to their beginning, then wait on Resume or Play.

cs.sendFastFwdForTime(value)

skip forward on a score or midifile performance and begin playing from that point. The skip will be instantaneous. Although no sound is produced during skipping, all score events or midi events are launched silently so that the score parameters are up-to-date at the new point of performance.

cs.sendFastFwdToTime(value)
skip forward or back to the requested point in a score or midifile and begin playing from that time. The skip will be instantaneous. A skip forward to a later score time will operate like FastFwdForTime (see above). A skip backwards to an earlier score time will first do a Rewind to the beginning of the score or midfile, then a Fast Forward to the time requested. Although no sound is produced during forward skipping, all score events or midi events are launched silently so that the score parameters are up-to-date at the new point of performance.

cs.waitonEnd()
Delay further Python commands to this instance until the current score has finished performing.

cs.sendDestroy()
ask the Csound Server to release the *current* instance and return all of its allocated memory. This action is included when a *current* instance is replaced by a *fresh* one via **newInstance**. Note that this action does not destroy the socket connection, only the attached Csound instance.

cs.sendDisconnect()
destroy the link between Python and the named socket connection. If a Csound instance is currently attached, this will release it first (i.e. **sendDestroy**) and then disconnect the socket. The resident server will then report on how many connection sockets remain.

cs.close()
this is currently an alias for **sendDisconnect**, but may have a different function in the future when Csound runs as a daemon.

cs.serverShutdown()
shutdown the present Csound Server. This command, sent via *any* socket, will terminate all active orchestras, destroy all instances, destroy all socket connections to the host Csound Server, and finally shut down the server itself.

3. Graphic Control using PYGTK

4. The Csound Environment

The Orchestra File

Csound runs from two basic files: an *orchestra* file and a *score* file. The orchestra file is a set of *instruments* that tell the computer how to synthesize sound; the score file tells the computer when. An instrument is a collection of modular statements which either *generate* or *modify* a signal; signals are represented by *symbols*, which can be "patched" from one module to another. For example, the following two statements will generate a 440 Hz sine tone and send it to an output channel:

```
asig    oscil   10000, 440, 1
        out     asig
```

The first line sets up an oscillator whose controlling inputs are an amplitude of 10000, a frequency of 440 Hz, and a waveform number, and whose output is the audio signal *asig*. The second line takes the signal *asig* and sends it to an (implicit) output channel. The two may be encased in another pair of statements that identify the instrument as a whole:

```
        instr   1
asig    oscil   10000, 440, 1
        out     asig
        endin
```

In general, an orchestra statement in **Csound** consists of an action symbol followed by a set of input variables and preceded by a result symbol. Its *action* is to process the inputs and deposit the result where told. The meaning of the input variables depends on the action requested. The 10000 above is interpreted as an amplitude value because it occupies the first input slot of an oscil unit; 440 signifies a frequency in Hertz because that is how an oscil unit interprets its second input argument; the waveform number is taken to point indirectly to a stored function table, and before we invoke this instrument in a score we must fill function table #1 with some waveform.

The output of **Csound** computation is not a real audio signal, but a stream of numbers which describe such a signal. If written to a file these can later be played (converted to sound) by an independent program; for now, we will think of variables such as *asig* as tangible audio signals.

Let us now add some extra features to this instrument. First, we will allow the pitch of the tone to be defined as a *parameter* in the score. Score parameters can be represented by orchestra variables which take on their different values on successive notes. These variables are named sequentially: p1, p2, p3, ... The first three have a fixed meaning (see the Score File), while the remainder are assignable by the user. Those of significance here are:

p3 - duration of the current note (always in seconds).
p5 - pitch of the current note (in units agreed upon by score and orchestra).

Thus in

```
asig    oscil   10000, p5, 1
```

the oscillator will take its pitch (presumably in cps) from score parameter 5.

If the score had forwarded pitch values in units other than cycles-per-second (Hertz), then these must first be converted. One convenient score encoding, for instance, combines *pitch class* representation (00 for C, 01 for C#, 02 for D, ... 11 for B) with *octave* representation (8. for middle C, 9. for the C above, etc.) to give pitch values such as 8.00, 9.03, 7.11. The expression

```
cpspch(8.09)
```

will convert the pitch A (above middle C) to its cps equivalent (440 Hz). Likewise, the expression

```
cpspch(p5)
```

will first read a value from p5, then convert it from octave.pitch-class units to cps. This expression could be imbedded in our orchestra statement as

```
asig    oscil   10000, cpspch(p5), 1
```

to give the score-controlled frequency we sought.

Next, suppose we want to shape the amplitude of our tone with a linear rise from 0 to 10000. This can be done with a new orchestra statement

```
amp     line    0, p3, 10000
```

Here, *amp* will take on values that move from 0 to 10000 over time p3 (the duration of the note in seconds). The instrument will then become

```
        instr   1
amp     line    0, p3, 10000
asig    oscil   amp, cpspch(p5), 1
        out     asig
        endin
```

The signal *amp* is not something we would expect to listen to directly. It is really a variable whose purpose is to control the amplitude of the audio oscillator. Although audio output requires fine resolution in time for good fidelity, a controlling signal often does not need that much resolution. We could use another kind of signal for this amplitude control

```
kamp    line    0, p3, 10000
```

in which the result is a new kind of signal. Signal names up to this point have always begun with the letter **a** (signifying an *audio* signal); this one begins with **k** (for *control*). Control signals are identical to audio signals, differing only in their resolution in time. A control signal changes its value less often than an audio signal, and is thus faster to generate. Using one of these, our instrument would then become

```
        instr   1
kamp    line    0, p3, 10000
asig    oscil   kamp, cpspch(p5), 1
        out     asig
        endin
```

This would likely be indistinguishable in sound from the first version, but would run a little faster. In general, instruments take constants and parameter values, and use calculations and signal processing to move first towards the generation of control signals, then finally audio signals. Remembering this flow will help you write efficient instruments with faster execution times.

We are now ready to create our first orchestra file. Type in the following orchestra using the system editor, and name it "intro.orc".

```
          sr = 20000                  ; audio sampling rate is 20 kHz
          kr = 500                    ; control rate is 500 Hz
          ksmps = 40                  ; number of samples in a control period (sr/kr)
          nchnls = 1                  ; number of channels of audio output

          instr   1
kctrl     line    0, p3, 10000        ; amplitude envelope
asig      oscil   kctrl, cpspch(p5), 1 ; audio oscillator
          out     asig                ; send signal to channel 1
          endin
```

It is seen that comments may follow a semi-colon, and extend to the end of a line. There can also be blank lines, or lines with just a comment. Once you have saved your orchestra file, we can next consider the score file that will drive it.

The Score File

The purpose of the score is to tell the instruments when to play and with what parameter values. The score has a different syntax from that of the orchestra, but similarly permits one statement per line with comments after a semicolon. The first character of a score statement is an **opcode**, determining an action request; the remaining data consists of numeric parameter fields (pfields) to be used by that action.

Suppose we want a sine-tone generator to play a pentatonic scale starting at C-sharp above middle-C, with notes of 1/2 second duration. We would create the following score:

```
;  a sine wave function table
f1 0 1024 10 1
;  a pentatonic scale
i1   0  .5      0.      8.01
i1   .5 .       .       8.03
i1 1.0  .       .       8.06
i1 1.5  .       .       8.08
i1 2.0  .       .       8.10
e
```

The first statement creates a stored sine table. The protocol for generating wave tables is simple but powerful. Lines with opcode **f** interpret their parameter fields as follows:

p1 - function table *number* being created
p2 - *creation time*, or time at which the table becomes readable
p3 - table *size* (number of points), which must be a power of two or one greater
p4 - *generating subroutine*, chosen from a prescribed list.

Here the value 10 in p4 indicates a request for subroutine **GEN10** to fill the table. **GEN10** mixes harmonic sinusoids in phase, with relative strengths of consecutive partials given by the succeeding parameter fields. Our score requests just a single sinusoid. An alternative statement:

```
f1 0 1024 10 1 0 3
```

would produce one cycle of a waveform with a third harmonic three times as strong as the first.

The **i** statements, or note statements, will invoke the p1 instrument at time p2, then turn it off after p3 seconds; it will pass all of its p-fields to that instrument. Individual score parameters are separated by any number of spaces or tabs; neat formatting of parameters in columns is tidy but not essential. The dots in p-fields 3 and 4 of the last four notes invoke a *carry feature*, in which values are simply copied from the immediately preceding note *of the same instrument*. A score normally ends with an **e** statement.

The unit of time in a **Csound** score is the *beat*. In the absence of a *Tempo* statement, one beat takes one second. To double the speed of the pentatonic scale in the above score, we could either modify p2 and p3 for all the notes in the score, or simply insert the line

```
t 0 120
```

to specify a tempo of 120 beats per minute from beat 0.

Two more points should be noted. First, neither the *f*-statements nor the *i*-statements need be typed in time order; **Csound** will sort the score automatically before use. Second, it is permissible to play more than one note at a time with a single instrument. To play the same five notes as a three-second pentatonic chord we would create the following:

```
;   a sine wave function
f1  0  1024   10     1
;   five notes at once
i1   0   3        0        8.01
i1   0   .        .        8.03
i1   0   .        .        8.06
i1   0   .        .        8.08
i1   0   .        .        8.10
e
```

Note also that the **f** statement need not be in the score. It is also a legal statement in an orchestra, and was used in that way in Examples 1, 2 and 3 above where its proximity made things clearer. However, by being in the orchestra header section an **f** statement always has an action time of 0. When included in the score, an **f** statement can be scheduled at any time during the performance.

Now go into an editor once more and create your own score file. Name it "intro.sco". Next go back into Python and use the orchestra *intro.orc* to play the score *intro.sco*. Try various combinations of Pause and Resume. Also try setting up Multiple Instances of your orchestra (for the moment it doesn't matter if they're the same), and try Pauseing and Resuming in various ways. Also try changing the Volume settings.

Now given that you have identical orchestras and identical scores, try starting them together with slightly different values in setTempo. When does the lack of sync become annoying? Now add a "conduct" feature to one of them. Can you use that to bring them back into line?

Frequencies within the Orchestra

Suppose we next wished to introduce a small vibrato, whose rate is 1/50 the frequency of the note (i.e. A440 is to have a vibrato rate of 8.8 Hz.). To do this we will generate a control signal using a second oscillator, then add this signal to the basic frequency derived from p5. This might result in the instrument

```
        instr   1
kamp    line    0, p3, 10000
kvib    oscil   2.75, cpspch(p5)/50, 1
a1      oscil   kamp, cpspch(p5)+kvib, 1
        out     a1
        endin
```

Here there are two control signals, one controlling the amplitude and the other modifying the basic pitch of the audio oscillator. For small vibratos, this instrument is quite practical; however it does contain a misconception worth noting. This scheme has added a sine wave deviation to the cps value of an audio oscillator. The value 2.75 determines the *width* of vibrato in cps, and will cause an A440 to be modified about one-tenth of one semitone in each direction (1/160 of the frequency in cps). In reality, a cps deviation produces a different musical interval above than it does below. To see this, consider an exaggerated deviation of 220 cps, which would extend a perfect 5th above A440 but a whole octave below. To be more correct, we should first convert p5 into a *true decimal octave* (not cps), so that the deviation *interval* above is equivalent to that below. In general, pitch modification is best done in true octave units rather than pitch-class or cps units, and there exists a group of pitch converters to make this task easier. The modified instrument would be

```
        instr   1
ioct    =       octpch(p5)
kamp    line    0, p3, 10000
kvib    oscil   1/120, cpspch(p5)/50, 1
asig    oscil   kamp, cpsoct(ioct+kvib), 1
        out     asig
        endin
```

This instrument is seen to use a third type of orchestra variable, an *i*-variable. The variable *ioct* receives its value at an *initialization* pass through the instrument, and does not change during the lifespan of each note. There may be many such *init time* calculations in an instrument. As each note in a score is encountered, the event space is allocated and the instrument is initialized by a special pre-performance pass. *i-variables* receive their values at this time, and any other expressions involving just constants and *i*-variables are evaluated. At this time also, modules such as **line** will set up their target values (such as beginning and end points of the line), and units such as **oscil** will do phase setup and other bookkeeping in preparation for performance. A full description of init-time and performance-time activities, however, must be deferred to a broader consideration of the orchestra syntax.