Chapter 1

Audio Input and Output

1.1 Buffering

It is not practical to produce sound samples at exactly the rate required by the computer's audio hardware. If it is required to produce sound at CD quality, and the computer were to be interrupted every 20μs or so to produce the next sample, the associated overhead would be unacceptably large and very little of the machine's time would be spent in useful computation. For this reason, without exception, general purpose computers maintain a buffer for the input and output of sound samples.

Buffers which preserve the order in which data is presented are referred to as FIFO (First In First Out) buffers or queues.[1] Figure 1.1 illustrates the important concepts.

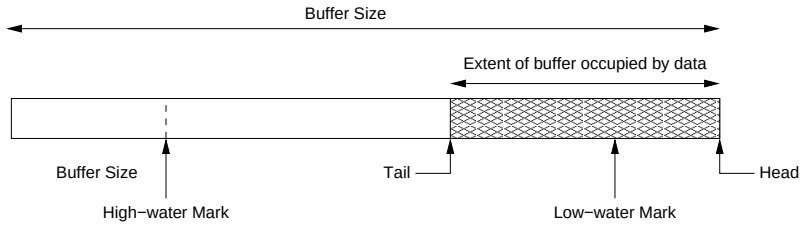


Figure 1.1: A FIFO Buffer or Queue

As audio samples are generated by the application, they are added to the *tail* of the queue, and the quantity of data expands

The buffered data is allowed to grow without any action being taken by the audio hardware until the data reaces the *low-water mark*. The low water mark is set by the system administrators and application program at such a level that, once reached, there can be a reasonable degree of certainty that the data in the buffer will be sufficient to ensure a continuous stream of samples will be delivered to the sound hardware, even if there are changes from time to time in the performance of the sound-generating program. Setting a very low low-water mark means that sound generation will begin very quickly after the first audio samples become available, but risks interruptions in the audio output if there are the briefest pauses in

[1] In normal usage, a *buffer* suggests that unit of preallocated system resource is used to store the data with a consequent compile- or run-time limit imposed on the maximum quantity of data which can be stored. This is the case in most computer sound systems. A *queue*, on the other hand, suggests a dynamic stucture which expands indefinitely until system resources are unavailable. One would tend to refer to a "sound buffer", but to a "print queue".

program execution. Setting a high low-water mark means there may be a noticeable latency between samples becoming available from the application and sound emerging from the computer, but means that even if the system becomes heavily loaded and the program stalls for a considerable period, samples will be available to produce uninterrupted sound. The application program is allowed, within limits, to choose the high- and low-watermarks to adjust the performance of the sound sub-system. For example, a mail program which uses an audio cue to alert the user to incoming mail will by unconcerned by latency and set a high low-water mark to ensure continuous playback under any circumstances, whereas a synthesiser program which generates sounds in response keypresses on a MIDI keyboard needs to make sure that the corresponding sound is emitted almost instantly, and will set a low low-water mark.

The concept of the *high-water mark* is less important in the context of a computer sound system, where the client program and the sound output device are almost always tightly coupled. If the buffer fills beyond the high-water mark, the sound subsystem may instruct the client application to cease the production of samples until the level of samples in the buffer falls below the high-water mark once again. The high-water mark is important in certain networking environments, where it might take time for a "stop transmission" instruction to reach the source, and there must be enough left-over room in the buffer to store information arriving after after the stop instruction has been sent out if data loss is to be avoided. However, this is not a relevant problem in a computer music system, because the application program is either running on the same computer as the sound hardware, or is protected from the overrun problem by the operating system's data transport layer. Consequently, the high-water mark is almost always the same as the buffer size.

1.2 The Rôle of the Kernel

If it were necessary to manipulate the sound hardware directly, every application would be required to understand the internal working of every available piece of sound hardware available on any computer system. This would make the code unwieldy and maintenance onerous. It is the kernel's job to provide a degree of *hardware abstraction*, which is to say that all application programs may access the same interface regardless of the installed hardware. We shall consider the functions available to applications running on a Unix system using the Open Sound System (OSS) interface. This interface is supported by GNU/Linux kernels, as a subset of the now-adopted ALSA (Advanced Linux Sound Architecture). It is also available on a number of other Unix platforms.

1.2.1 Basic OSS functionality

OSS provides a number of devices which the application may address. Of the most interest is the *primary audio device* /dev/audio. The system calls open(2), close(2), read(2) and write(2) permit the sound hardware to emit sound samples generated by an application, or to make available sound samples to be read by the application.

Reading and writing data between an application and a soundcard, unfortuantely, can far more difficult than reading and writing a regular file. Consequently, several extra facilities are available to manipulate the behaviour of the sound hardware *via* the ioctl(2) system call. This interface permits the setting and reading of parameters such as:

Sound hardware capabilities: is the sound harware capable of stereo recording and reproduction? 4 channel? of just monophonic? What are the maximum

and minimum sample rates?

Setting the hardware sample rate: request a particular sample rate from the sound card;

Setting the number of channels: request the data be treated as interleaved samples from or to a number of channels;

Requesting status information: how many samples have been played or recorded since the device file was opened?

Input/output buffer parameter manipulation: set the total size and fragment size (effectivly the low-water mark) of the audio buffers.

The getcaps program (Code Segment 1.1) demonstrates the use of the ioctl(2) system call by testing for the sound system's capabilities. ioctl() takes three arguments: the file descriptor (supplied by open(2)) on which to operate; the *request*; and a *pointer to* a parameter. ioctl() returns -1 in the event of a system error; it attempts to honour the request, but in the event of being unable to do so, reports the action it has managed to take by setting the parameter appropriately. The criteria for complete success, therefore, is a return from ioctl() which is not -1, and the value of the parameter returned being equal to the value of the parameter supplied.

Running the program on my (rather old) computer yields the following:

```
Available sample sizes: 8       16
Supported number of channels:   1       2
Max sample rate:        44100
Min sample rate:        4000
```

1.2.2 Implementing High- and Low-water Marks with OSS

OSS does not implement high- and low-water marks directly, but it does permit the buffer to be divided into a number of *fragments*. This effectively permits the setting of a low-water mark, because no output will occur before there is at least one full fragment in the buffer. For reasons peculiar to the implementation, the size of each fragment must be a power of 2.

Consider the buffer layout shown in Figure 1.2. To achieve this configuartion,

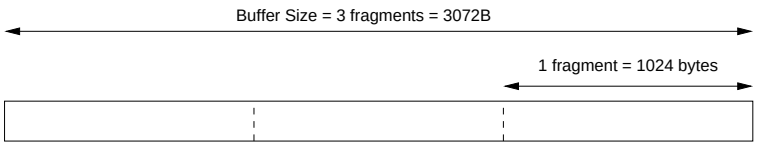


Figure 1.2: A Sound Buffer with Three Fragments

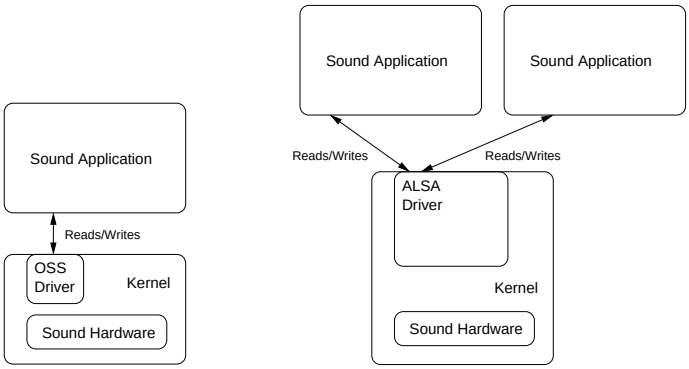
the argument to the ioctl() system call would be $(3 \ll 16) \vee 10 = 196618$ (because $2^{10} = 1024$)

1.3 Sound Dæmons: the Client/Server Approach

The kernel provides a consistent interface for Unix applications to communicate with sound hardware, but problems arise when more than one application wishes to make sound at the same time. Generally, the act of opening the sound device locks out other applications from using it. As indicated in Figure 1.3(a), The first

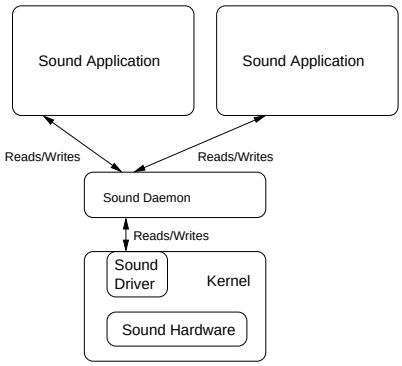
application to grab the sound device "wins" and may cause other applications to run without sound output, or even fail to start at all. The OSS interface defines a mixer device, but this is intended to mix a single sound source from the samples sent to the sound card with other sound sources available from the hardware such as CD/DVD audio, MIDI synthesiser outputs and so on. It will not mix different sample streams from multiple client applications.

As Figure 1.3 indicates, There are two popular solutions to this problem. Some



(a) Direct access to /dev/dsp. The complexity of the kernel model is very small. Some Unices still use this model, but Linux has migrated to ALSA (which contains backwards compatibilty with OSS)

(b) Multi-stream access via the kernel. Kernel complexity is increased, but multiple connections are permitted.



(c) Multi-stream access via a user-mode dæmon. Kernel complexity remains small, but multiple clients can still connect.

Figure 1.3: Alternative Approaches to Sound Hardware Arbitration

platforms provide kernel support for sound devices which can be opened by multiple applications (such as ALSA, the Advanced Linux Sound Architecture, Figure 1.3(b)). Alternatively, a user-space program (like ESD, the Enlightened Sound Dæmon, originally associated with the Enlightenment window manager) can be run to act as a "sound server", Figure 1.3(c), accepting connexions *via* a socket and performing mixing after sample rate and type conversion. The latter approach is attractive from the architectural point of view, adhering to the principle of putting the minimum amount of code in the kernel where it is "locked and loaded": kernel code, once loaded, occupies physical memory until it is unloaded again; it cannot be paged out like user-space programs. Consequently, large and complex kernel

drivers may impact upon the performance of the entire system even if the sound is currently not in use. However, it is questionable whether the overhead associated with copying the sound samples through a socket connexion, even if it on the local machine, outweighs the disadvantage of the large, physical-memory-resident audio driver. Others argue that the closer binding of the application to the hardware possible with the ALSA approach is essential in high-performance audio applications such as hard-disk recorders and real-time music synthesis, and once admitted on that basis, it does not make sense to support the ESD method as well.

Nevertheless, the ESD method is sufficiently popular to warrant it's being covered here. Many open-source applications test for the availability of the Enlightend Sound Daemon and configure themselves accordingly. Code Segment 1.3 shows the relevant parts of program which issues audio samples using ESD. The application program interface and constants used are documented in the file esd.h

1.4 Increasing Portability with PortAudio

Although the esd approach permits a portable method of combining multiple sound streams for many Unix platforms, it isn't sufficiently portable to let you write code which comples on Microsoft platforms, or on the "It's Unix, Jim, but not as we know it" MacOS-X. PortAudio, a library from Ross Bencina and others, does achieve this.

Portaudio normally uses a more sophisticated method of Audio I/O than has so far been described, which will be covered in Chapter **??**. However, it comes with a simple interface called pablio (PortAudio Blocking I/O) which has exactly the functionality we need, and it is this which is used in Code Segment 1.4.

Lines 14–19 specify the data types, sample rate and number of channels which PortAudio will be expected to use. This example uses 32-bit floating-point numbers representing a mono signal. A blocking audio stream is openned at the given sample rate and type, for mono writing. If successful, control is passed to some process which fills the array `samples` with audio data, and then `WriteAudioStream()` is called on line 44 to cause the samples to be played. The audio stream is then closed.

PortAudio always takes the number of samples in the input array as an argument, rather than the size of the array. It is therefore conventional to specify a two-dimensional array, the second dimension being the number of audio channels. The required argument to the PortAudio functions is then the first dimension of this array.

Code Segment 1.1 Reading the Sound Hardware Capabilities

```
/* Getcaps.c */
/* Get and print the capabilities of the sound subsystem */

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <linux/soundcard.h>

int main (void)
{
    int dspfd; /* File descriptor for audio device */
    int inval, outval; /* input and output data for ioctl(2) */

    if ((dspfd = open("/dev/dsp", O_RDONLY)) == -1) {
        fprintf(stderr,"open failed");
        exit(1);
    }
    printf("Available sample sizes:");
    for (inval=8; inval<=32; inval*=2) { /* Check 8..64 doubling each time */
        outval=inval;
        if (ioctl(dspfd, SOUND_PCM_WRITE_BITS, &outval)==-1) {
            fprintf(stderr, "ioctl() failed trying %d bits", inval);
            exit(1);
        }
        if (outval==inval) printf("\t%d", outval);
    }
    printf("\nSupported number of channels:");
    for (inval=1; inval<=8; inval++) { /* Check 1..8 channels */
        outval=inval;
        if (ioctl(dspfd,SOUND_PCM_WRITE_CHANNELS, &outval)==-1) {
            fprintf(stderr, "ioctl() failed trying %d chans", inval);
            exit(1);
        }
        if (outval==inval) printf("\t%d", outval);
    }
    /* Set silly values to find out the max/min sample rates */
    outval = 250000; /* Try for 250kHz */
    if (ioctl(dspfd, SOUND_PCM_WRITE_RATE, &outval)==-1) {
        fprintf(stderr, "Failed to set high sample rate");
        exit(1);
    }
    printf("\nMax sample rate:\t%d", outval);
    outval = 250; /* Try for 250Hz */
    if (ioctl(dspfd, SOUND_PCM_WRITE_RATE, &outval)==-1) {
        fprintf(stderr, "Failed to set low sample rate");
        exit(1);
    }
    printf("\nMin sample rate:\t%d\n", outval);
    exit(0);
}
```

Code Segment 1.2 Setting the Sound Buffer Fragment Size

```
/* Sets the buffer size in fragments and fragment size in bytes.
   The value of p has the number of fragments in the upper 16 bits,
   and log_2 fragment size in the lower. */

int p = SBUF_FRAGSIZE | (SBUFFRAGS << 16);
if (ioctl(dspfd, SNDCTL_DSP_SETFRAGMENT, &p)) {
  fprintf(stderr, "Error whilst setting sound buffering. "
                  "%d fragments of %d bytes each not permitted\n",
                  SBUFFRAGS, 1<<SBUF_FRAGSIZE);
  return 1; /* Flag error */
}
```

Code Segment 1.3 Audio Output via the Enlightened Sound Dæmon

```
#include <esd.h>

{
  int out_sock;
  int play_format = ESD_MONO | ESD_BITS16 | ESD_STREAM | ESD_PLAY;

  /* esd_play_stream_fallback tries to use esd, but uses /dev/dsp if necessary */
  out_sock = esd_play_stream_fallback(play_format, /* Sample format & function */
                                      44100, /* Sample rate down the stream */
                                      NULL, /* host to use (localhost:5001) */
                                      "MySamples"); /* Unique name */
  if (out_sock <= 0) { /* Handle error */
  } else { /* Play samples using write(2) to out_sock */
  }
  close(out_sock);
}
```

Code Segment 1.4 Audio Output via the PortAudio Library using Blocking I/O

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pablio.h"
#include "portaudio.h"

/*
** Note that many of the older ISA sound cards on PCs do NOT support
** full duplex audio (simultaneous record and playback).
** And some only support full duplex at lower sample rates.
**
** Consult distributed source for copyright and attributions.
*/
#define SAMPLE_RATE  (44100)
#define NUM_SECONDS     (5)
#define SAMPLES_PER_FRAME (1)

#define SAMPLE_TYPE paFloat32
typedef float SAMPLE;

int main(int argc, char *argv[])
{
    int i;
    SAMPLE samples[NUM_SECONDS*SAMPLE_RATE][SAMPLES_PER_FRAME];
    PaError err;
    PABLIO_Stream *aStream;

    /* Open simplified blocking I/O layer on top of PortAudio. */
    err = OpenAudioStream(&aStream, SAMPLE_RATE, SAMPLE_TYPE,
                                        (PABLIO_WRITE | PABLIO_MONO));
    if(err != paNoError) {
        fprintf(stderr, "%s: OpenAudioStream() failed", argv[0]);
        Pa_Terminate();
        fprintf(stderr, "An error occured while using the portaudio stream\n");
        fprintf(stderr, "Error number: %d\n", err);
        fprintf(stderr, "Error message: %s\n", Pa_GetErrorText(err));
        exit(-1);
    }

    /* Create some audio samples in the array "samples" */
    CreateSampels(samples);

    /* Write that same block of data to output. */
    WriteAudioStream(aStream, samples, NUM_SECONDS*SAMPLE_RATE);

    CloseAudioStream(aStream);
    exit(0);
}
```

Chapter 2

Additive Synthesis

The synthesis of a pitched musical tone implies the generation of a periodic signal, and possibly the most intuitive method of achieving this is Additive Synthesis. It was the mathematician ?? Fourier who showed in ?? that *any* periodic function can be expressed as the sum of a series of simple sinusiodal waveforms having frequencies which are integral multiples of the periodic function's repetition frequency. This repetition frequency is referred to as the *fundamental frequency*, and is closely related to the pitch of the note. The relative amplitudes of these constituent sinusoidal oscillations (*harmonics* or *partials*) are responsible for the timbre of the rendered note; a relatively high proportion of higher-frequency harmonics produces a complex or brighter tone such as the sound of a reed instrument, whereas a concentration in the lower harmonics produces a simpler, flute-like effect. Additive Synthesis is the process of generating a complex periodic wave by the addition of signals from a bank of sinusoidal oscillators with independently controllable frequency and phase.

Since each of the oscillators in an additive synthesizer is under independent control, it is not mandatory for each of the signals generated by the synthesizer to have a frequency which is some whole-number multiple of the fundamental frequency. The harmonic relationship of the various signals is so easily compromised in such a system that the term *partials* is often used instead. The constituent partials are not necessarily related to the pitch of a note in acoustic musical instruments by simple integer frequency, so this convention is not inappropriate.

It would seem at this stage that additive synthesis is the only synthesis algorithm one is required to implement, since it is capable of the generation of arbitrarily complex timbre with any degree of control available to the controlling computer. However, this point of view is a rather simplistic one and fails on two important counts: the quantity of data required to control such a synthesizer is prohibitively large; and the computational demands made by such an algorithm turn out to be excessive. Whilst the latter point can be addressed by the addition of specialised hardware, or might become less and less significant as the speed of general purpose processors increases, the former point is always a concern since it demands that the operator of the synthesizer is either required to furnish vast quantities of control data, or loose some generality or subtlety of control of the instrument. Nevertheless, additive synthesis is still used from time to time where its great flexibility and potentially high signal quality make it a satisfactory choice.

2.1 Direct Synthesis

The simplest method for the construction of a complex waveform by additive synthesis is undoubtedly *direct synthesis*. We shall construct an oscillator routine which

adds in a sinusoidal component to a vector of given length; this may then be called repeatedly for each oscillator, eventually building up the desired waveform. In this simple example, the output will be of constant amplitude.

Code segment ?? demonstrates the creation of an oscillator. An oscillator is a *stateful* entity; it is necessary to preserve the current phase of the oscillator between successive calls so that the waveform can be continued from where it was terminated after the previous call. The routine `g_partial1` (code segment ??) is then called three times to add a fundamental component and two additional partials at three and five times this frequency. The resulting waveform is an approximation to a square-wave, shown in figure ??. This `g_partial1` routine is exceedingly simple; in order to calculate the contribution of an oscillator at any given time, repeated calls are made to the supplied mathematics library routine `cos()`. Because no use is made of the intrinsic characteristics of the problem by this algorithm, its execution is rather ponderous. For a practical additive synthesis which might be used in real time, it is necessary to find some more efficient way of generating sinusoidal signals. Some candidate methods, such as those employing look-up tables, are speed efficient but use rather large quantities of memory; others use a mathematical model of a mechanical oscillator by simulating the displacement of a spring and mass system at small time increments. The following sections consider both of these approaches.

2.2 The Look-up Table

Using direct calculation to implement an oscillator is not efficient. Calling the sin(3) or cos(3) functions causes the particular sample to be calculated "from scratch" rather than take account of samples calculated before. It is indeed possible to generate simple harmonic oscillations "on the fly" with very little calculation overhead by using information in the previous two samples. This will be demonstrated in Section 2.7. However, as a first stab at implementing an oscillator, look-up tables will be considered.

A lookup table contains precomputed values of the waveform to be emitted. In the simplest case, a table is used which simply contains the numeric values of all of the samples necessary to represent a complete cycle of the desired waveform. The contents of the table can then be passed directly to the write(2) system call causing a sound to emerge from the computer's audio device.

For a more general solution, one may wish to vary the amplitude and frequency of the emitted wave. It is infeasible to store a separate table for all possible amplitudes and all possible frequencies which may be required. Instead, we store sufficient information to specify one cycle of the waveform at the required accuracy, and step through the lookup table not at one position per sample, but at whatever rate is necessary to produce the required output frequency. In Chapter ?? we saw how the quantisation of an analogue signal introduces noise, and that finer quantisation leads to less noise. This quantisation noise is introduced in the random errors arising from the difference between the precise value of the continuous signal and the quantised value of the discrete one. This begs the question, what exactly is "sufficient information" to accomplish this, and are there any trade-offs which can be made between computational effort and memory usage so that a compromise can be found which is best suited to a particular computer architecture?

2.3 Determining the Table Length

By way of example, take the generation of a cosine wave by table-lookup. A cosine wave has a period of 2π, and it will only be necessary to store one cycle in the table.

Given the system sample rate and bit depth, one may work out directly the number of table entries that are required.

Because the lookup table's job is to provide samples to fill an output buffer, it isn't necessary to provide an entry in the table for each value required by the lowest frequency wave. This would result in a table which was very wasteful of storage space, because there are many repeated entries. Instead the table size can be chosen by making sure that there is no possibility of an value being looked up which is in error because of the quantisation *in time* of the table's contents. Here "in error" means that the returned value is correct with in the resolution of the system's quantisation *in amplitude*. The two factors are related by the maximum gradient of the function stored in the table: if there are large gradients, many samples must be stored because even a small rounding error on the time axis leads to a significant error in the amplitude of the recovered value.

Fortuantely, the cosine function is a relatively friendly one, and we know that

$$\left\lceil \left| \frac{d}{dt} \cos(t) \right| \right\rceil = \lceil |-\sin(t)| \rceil = 1 \tag{2.1}$$

If a 16-bit quantisation scheme is used, there will be 65536 discrete levels available for the audio output, and at the maximum rate of change of the stored function, stepping $\frac{1}{65536}$th of the way through the table will result in a different sample value being returned. A first attempt at a lookup-based oscillator with a 16-bit audio accuracy will therefore use a table of length 64Ksamples, or if short integers are stored in the table, 128Kbytes.

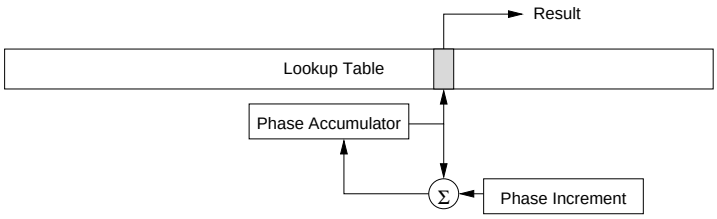


Figure 2.1: Accessing a Lookup Table with a Phase Accumulator

Traditionally, accessing the table is done using a *phase accumulator* as shown in Figure 2.1. For each sample which needs to be produced, a constant *phase increment* is added to the phase accumulator. When the value of the new sample is calculated by taking the phase accumulator (modulo the lookup table size) as an index into the table, and reading the resulting value. Often, the phase accumulator is treated as a fixed-point number rather than a simple integer, allowing fractional increments along the register to be represented. In this example, even though only 16 bits are required to index a table 64Ksamples in length, a 32-bit integer could be used. Only the top 16 bits might be used to index the lookup table; the bottom 16 might be considered a fractional part. The position of the "fixed point" is arbitrary, but in this example, if it were required to advance the phase accumultor 12.25 places per sample, the number $12.25 \times 2^{16} = 819200$ would be used as the phase increment.

2.4 Shortening the Lookup Table

The application of some processing to the intermediate results from the lookup table can significantly reduce the storage requirements. The first and most obvious way to reduce the table length is to exploit the symmetry of the cosine wave. Figure 2.2

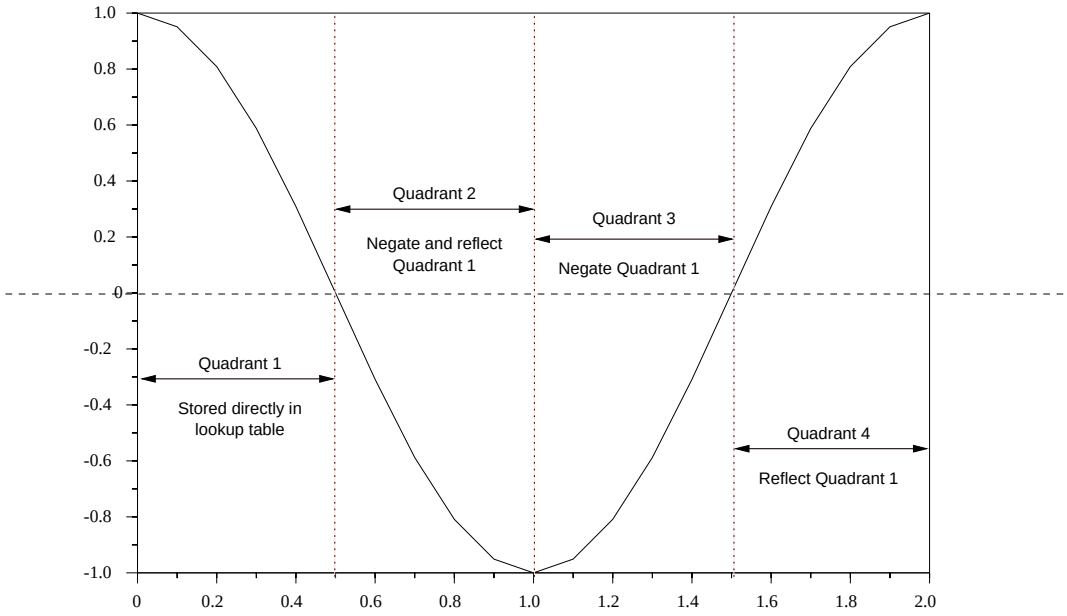


Figure 2.2: Storing Only Part of the Lookup Table

shows how the waveform can be divided up to achieve this end. The first quarter-cycle is read directly from the table; subsequent quadrants are transformed as shown in Table 2.1. l is the assumed table length, of which $l/4$ is acually stored.

Value in Phase Accumulator	**Value from Table**
$0 \leq$ Phase Acc $< l/4$	LUT[Phase Acc]
$l/4 \leq$ Phase Acc $< l/2$	$1 -$ LUT[$(l/2)$-Phase Acc]
$l/2 \leq$ Phase Acc $< 3l/4$	$-$LUT[Phase Acc-$(l/2)$]
$3l/4 \leq$ Phase Acc $< l$	LUT[l-Phase Acc]

Table 2.1: Looking up a value from a quarter-wave table

The result of this is that the length of the lookup table can be shortened by a factor of four if the values it contains demonstrate the same symmetry as the cosine wave. More complex waveforms can be used if they are pre-calculated in a way which assures this symmetry: if a single complex waveform is required rather than a general-purpose additive synthesizer module, then this can be achieved very simply by initialising the look-up table with the sampled required waveform rather than a sinusoid. This is the fundamental *modus operandi* of the sampling synthesizer, and we discuss this later in Chapter ??. The computational cost of the table's shortening is at most two tests and a subtraction, which is very small indeed, so even on modern machines where memory consumption is less important a consideration, this technique may still be valuable.

2.5 The Oscillator as an Object

If the lookup-table oscillator is to be a useful component of a larger system, such as an additive synthesiser which sums the outputs from many such oscillators, it is good practice to encapsulate the code and assoicated data structures in a single class definition, permitting oscillator objects to be instantiated at will. It is quite possible to write good object-oriented code using the C programming language, and relying upon the authors style and naming conventions to isolate different classes of objects, but since compiler technology provides efficient and safer methods of encapsulating objects, we should begin to design a look-up-table oscillator as we mean to go on and use an object-oriented language such as C++.

The generic name for an object which synthesises audio samples is a *generator*. These objects must provide methods to get data a sample at a time, which the application can combine in the desired fashion to produce the audio output. However, this is an inefficient way of synthesising sound. Consider the additive synthesiser which has already been suggested, and suppose that the output is generated by summing together three oscillators. It would be necessary to make three method invocations per sample, so to fill a buffer with, say 48000 samples, 144000 subroutine calls will be made.

Instead of making a generator which produces only one sample per call, it makes sense to fill a buffer of samples of a particular length with data in "one go". Certain parameters, such as oscillator frequency and amplitude, may be considered constant for the length of the buffer being filled. If it is absolutely crucial to vary the amplitude or frequency on a per-sample basis, the generator method can simply be called with a length of 1. In almost all situations, the rate at which the amplitude or frequency of the oscillator needs to be varied rarely exceeds once every 20 samples, so the generator can be used to produce data bursts. Even calling the generator method every 20 samples rather than every single one can represent a substantial saving, and in some cases the bursts of data produced can be much larger.

The rate at which samples are emitted from the sound card is referred to as the *audio rate*, and the rate at which contol parameters are varied is called the *control rate*. It should now be possible to write an abstract class definition for a generator.

2.5.1 Class Hierarchy and Methods

In view of the fact that it is often necessary to sum the output of many generators to produce the final result, it would be convenient if the gen() method took a pointer to a buffer as its argument, and modified the results of the buffer.

Apart from that, nothing can be said of a generator, so its class definition, in generator.hh (Code Segment 2.1)is short and sweet:

If only sinusoidal oscillations are required, the following observations can be made with regard to the oscillator:

- The oscillator is a subclass of generator (one can properly say "An oscillator is a sort of generator");
- The internal condition of an oscillator is characterised entirely by the contents of the phase accumulator, the phase increment per sample and a multiplication factor (corresponding to phase, frequency and amplitude);
- At instantiation, the amplitude and frequency components for the oscillator are required parameters;
- Methods are required to change the current amplitude and frequncy of an oscillator, but otherwise its state should be globally unmodifiable to guarantee phase continuity;

Code Segment 2.1 Generator: an Abstract Superclass

```
//
// Generator is the abstract superclass of all synthesis and
// filtering functions
//

#include "buffer.hh"

class Generator {
public:
  // All generators modify a buffer of floating-point samples
  virtual void gen(Buffer *buf, int count) = 0;
} ;
```

- The lookup table from which samples are drawn can be shared amongst all instances of the oscillator.

This influences the contents of oscillator.cc (Code Section 2.2).

An oscillator is a subclass of generator, so it contracts to supply the method gen(). gen() takes a pointer to a buffer and a length parameter; count samples are added into the buffer at the given amplitude and frequency.

The constructor accepts an initial frequency and amplitude, which may be altered during the oscillators lifetime using the set() method. Convenience methods are also provided which change the amplitude and frequency independently. Double precision floating point numbers are used to pass information to the oscillator, but its output is in single precision.

The oscillators internal state is stored in private variables. An inner class LUT supplies the lookup table, and this is initialised to contain the first quadrant of the cosine wave. The only instance of the LUT class is declared to be static so that one lookup table is shared between all extant instances of the oscillator.

oscillator.cc contains the guts of the class.

The quadrant lookup method decribed in Section 2.4 is implemented in gen(), and the initialisation of the lookup table Oscillator::LUT::LUT() is handled by its constructor. Since Oscillator::lut is declared as static, this initialisation occurs only once when the first oscillator is instantiated.

The Buffer class is used to hold oscillators intermediate results as an array of floating point numbers, and provdes a method to write its contents to the appropriate audio device when passed a file descriptor. Its prototype is shown in Code Segment 2.4

The main program (Code Segment 2.6)uses three oscillators to generate an approximation to a square wave, and is very simple thanks to the object-oriented design. Generating a buffer with the square-wave approximation takes only four lines of code. The samples, which are floating point values, are scaled and converted to shorts before being written to the audio device.

2.5.2 Observations

This oscillator is a genuinely encapsualted object, but is rather simplistic in some respects. The program could possibly be improved by taking the following into accout:

- The code is riddled with constants. Look-up table length is a constant, as is sample frequency. For the code to be genuinely reusable, it would be necessary to make these parameters adjustable when an oscillator is constructed.

- The buffer is of fixed type: an array of float. In a more flexible scheme, it would be desirable to permit different types of buffer.

2.5.3 Small Angle Interpolation

In these days of multi-megabyte personal computers, one might suggest that if it is necessary to achieve a speed-at-all-cost solution to the problem of sine-wave generation, then the look-up table approach has a great deal to recommend it. However, there are advantages to reducing the required size of the table. Modern P.C.s now incorporate a high speed cache consisting of fast memory devices which are able to retain a copy of parts of the main system memory. Obviously, the capacity of this closely coupled memory is considerably less than that available to the entire system, yet the ability to accommodate the whole of the look-up table within it would significantly enhance an oscillator's speed of computation. If most accesses to main memory are to areas which have already been stored in the cache, then the system is said to have a high proportion of 'cache hits', and the performance is improved. If the sine-wave oscillator is only part of a very much larger program, which perhaps makes use of many other complex data structures, it becomes undesirable to use large quantities of RAM to store tables as the appropriate pages may have been 'swapped out' to the hard disk before the time of reckoning; the user's program might be forced to pause for several milliseconds before computation continues pending the recovery of the appropriate data from disk. This will only occur when the total demand generated by the synthesis program is comparable with the amount of free system memory, but experience with electronic music packages on P.C.s shows that the available RAM tends to disappear at an alarming rate when more complex pieces are compiled. Whichever of these models applies to a particular system, a beneficial effect on speed of computation might be achieved by reduction in the length of the look-up table, so long as the computational overhead is not too great.

It is possible to reduce the length of the look-up table by the simple expedient of omitting some of the entries. However, we have already seen that this compromises the quality of the oscillator due to rounding errors in the accumulated phase which occur when the look-up table address is generated. In order to overcome this, we can use an interpolation that relies upon the following trigonometrical identity:

$$\sin(A+B) = \sin(A)\cos(B) + \cos(A)\sin(B) \tag{2.2}$$

If A is the nearest angle which is represented in our shortened look-up table, and B is the small difference between this and the desired angle, both expressed in radians, it is possible to make use of two further reductions as follows:

$$\left.\begin{array}{rcl} \sin(x) & = & x; \\ \cos(x) & = & 1 - x^2/2 \end{array}\right\} \text{for small x} \tag{2.3}$$

and if x is small, x^2 may be neglected, reducing equation 2.2 to:

$$\sin(A+B) = \sin(A) + B\cos(A). \tag{2.4}$$

It is possible to read values for $\sin(A)$ and $\cos(A)$ direct from the look-up table, so at the cost of two fetches, a look-up and an add, the table can be shortened significantly without loss of accuracy. The difference between 1 and $\cos(x)$ is approximately $x^2/2$, so for a sixteen-bit digital audio program, waveform entries retain accuracy at a spacing of 0.5 milliradians ($1/5^\circ$) as opposed to 0.1 milliradians ($1/200^\circ$) in the raw state. This represents a saving of factor of 40 in the length of the table for sixteen-bit look-up, which may well be worth considering. Code segment ?? shows the enhanced program.

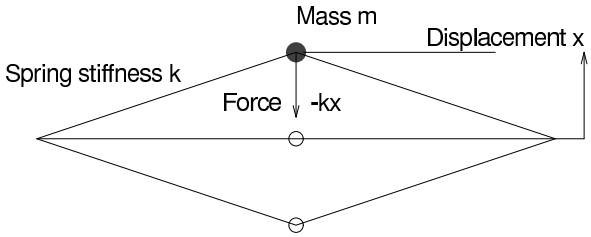


Figure 2.3: Simple Oscillating Mass and Spring

2.6 The Mechanical Oscillator Analogy

All of the algorithms so far involve the calculation of a cosine given a certain phase angle. The phase variable is retained between successive calls of the calculation of samples, and is the representation of the oscillator state. However, direct calculation, whether by the use of a library call or by table look-up, might not represent the most economical method of producing sampled values of a sinusoidal oscillator. Instead, it may be possible to produce the required values by simulating the behaviour of a system which naturally oscillates in this way, rather than the calculation of some more abstract mathematical function.

Such a system is shown in figure 2.3. It consists of a rather over-simplified view of a vibrating string. A real vibrating string, such as a violin string, would have its mass distributed over its whole length, and would oscillate in quite a different way from the model used here. However, we will show that this very simple model will suffice for the generation of sine-waves, which is all that is required for the moment. When the mass m is pulled upward through a displacement x_0, the "string" exerts a force F in the opposite direction equal to the displacement multiplied by the spring's stiffness, k. Since $F = ma$, the acceleration a of the mass is simply $a = -kx/m$. But the acceleration is simply the rate of change of velocity, and the velocity rate of change of displacement, so we have a second-order differential equation:

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x \tag{2.5}$$

for which the solution is known to be $x = x_0 \cos(\sqrt{\frac{k}{m}}t)$. By setting $\frac{k}{m}$ to $2\pi f$, and x_0 to the required amplitude, it should be possible to generate the series of samples we require by simulating this mechanical system in software. If our sample period T is sufficiently small, we can approximate velocity at sample n, $v_n = (x_n - x_{n-1})/T$. This yields three equations describing the mecanical oscillator:

$$a_n = -(k/m)x_n \tag{2.6}$$
$$v_n = v_{n-1} + Ta_n \tag{2.7}$$
$$x_n = x_{n-1} + Tv_n \tag{2.8}$$

By using $v_{n-1} = (x_{n-1} - x_{n-2})/T$, and substituting for a_n in equation 2.7, we immediately obtain the difference equation

$$x_n = \frac{1}{1 + T^2\frac{k}{m}}(2x_{n-1} - x_{n-2}) \tag{2.9}$$

This can be coded up easily, storing as the state of the system the last two values for x_n instead of the running total of phase.

Figure **??** shows the output of such an oscillator graphically; the code that produced this output is in code segment 2.7. Unfortuately, the assumption that

t^n	$\frac{n}{s^{n+1}}$
e^{at}	$\frac{1}{s-a}$
$\cos \omega t$	$\frac{s}{s^2+\omega^2}$
$\sin \omega t$	$\frac{\omega}{s^2+\omega^2}$
$\cosh at$	$\frac{s}{s^2-a^2}$
$\sinh at$	$\frac{a}{s^2-a^2}$

Table 2.2: Table of Standard Laplace Transforms

it is possible to simulate such a system as this accurately by assuming that we have sufficiently small time increments to count as negligible has been exposed as a falsehood. Instead of oscillating continuously in a well behaved, sinusoidal manner, the oscillator we have produced exhibits damping; the oscillations die away as time goes on. This is clearly an unsuitable state of affairs; although it may be required under some circumstances to produce such a waveform, our current requirement is for a continuous sinusoid which exhibits very little change in amplitude as time progresses. To understand and correct this problems requires us to delve deeper into the consequences of discrete-time sampling, and find out the significance of the assumptions made regarding the calculation of velocity and acceleration.

What we have been trying to achieve is the solution of a second-order differential equation. Mathematicians represent the rate-of change of position (velocity) as $\dot{x}$ or dx/dt, and acceleration as the rate of change of the rate of change of position, namely $\ddot{x}$ or d^2x/dt^2. Let us introduce the concept of a *differential operator* $\mathcal{D}$, which performs the differentiation of a function with respect to time. Equation 2.5 can now be written more concisely as $\mathcal{D}^2(x) = -kx$. This notation can be helpful in expressing differential equations concisely, but is not particularly helpful in solving them. The solution of equation 2.5, an undamped oscillatory system, is reasonably straightforward, but in subsequent chapters we shall require solutions of equations in higher orders of $\mathcal{D}$. Clearly, a mathematical tool for manipulating such problems is most desirable.

The Laplace Transformation is such a tool. It enables a function of time, such as $x(t)$ in the current example, to be transformed into another function of a *different* variable, conventionally $\mathcal{L}x(t) = X(s)$. One of the properties of the Laplace transform is that it converts problems involving differential equations into problems involving simple linear equations. For example, given that $\mathcal{L}x(t) = X(s)$, it can be shown that $\mathcal{L}\dot{x} = sX(s) - x(0)$ and that $\mathcal{L}\int x(t)\,dt = X(s)/s + x(0)$. Broadly speaking, instead of integrating, one divides by s, and instead of differentiating, one multiplies by s. The inclusion of the terms in $x(0)$ satisfy any initial conditions of the system.

Converting a function of time to its Laplace transformation is a matter of evaluating the integral

$$\mathcal{L}\{f(t)\} = \lim_{T\to\infty} \int_0^T f(t)e^{-st}\,dt \tag{2.10}$$

but fortunately, many standard transformations are well known; some are shown in table 2.2. The inverse transformation is a considerably more involved process, but from the engineering viewpoint, if the answer is sufficiently simple to be useful, it will probably either be in the table, or an obvious product or quotient of functions given in the table. Having introduced such a procedure, it turns out that through a better understanding of what the variable s really means, the design of digital oscillators becomes a much easier and more intuitive process. The design of digital filters, required for chapter 4, also relies heavily on this short-hand Laplace notation.

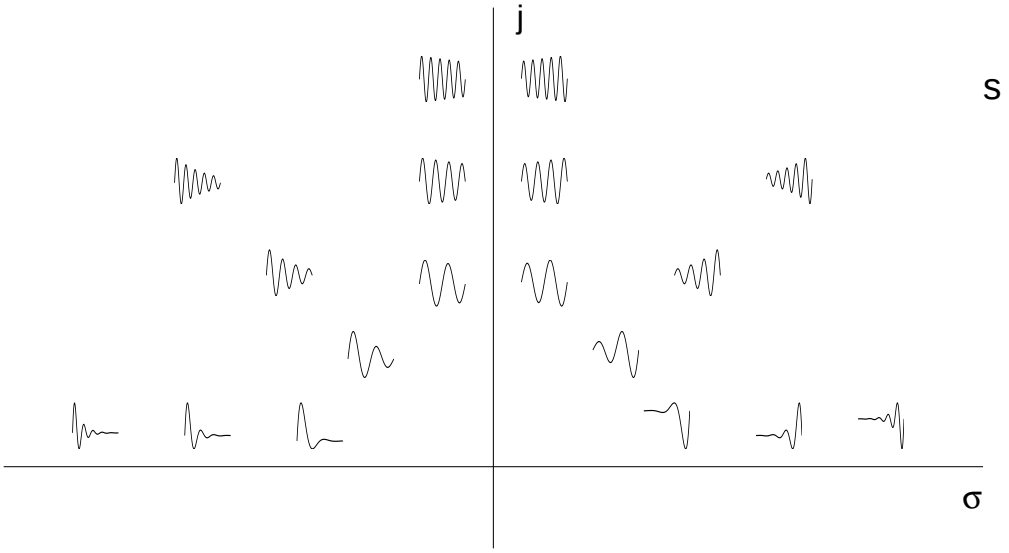


Figure 2.4: System Behaviour associated with s-plane Poles

2.6.1 Interpreting the Laplace Transform

The sort of differential equations which are of most interest to us are those with oscillating solutions such as $\sin(\omega t)$ and $\cos(\omega t)$. These have a general Laplace transformation which, after suitable algebraic manipulation, has the form

$$X(s) = \frac{(s - a_1)(s - a_2)\dots}{(s - b_1)(s - b_2)\dots} \tag{2.11}$$

where a_n and b_n are complex numbers. The values a_n are called the *zeros* of $X(s)$ because the value of $X(s)$ must be zero if $s = a_n$ for any particular n. By similar reasoning, if s is set to one of the values of b_n, then $X(s)$ tends to infinity at that point. The values b_n are referred to as the *poles* of X. If a pole and zero occurs coincidentally, then the situation is a little more difficult, but we shall examine the result when beginning to design filters.

If we consider the value s as a complex plane so that $s = \sigma + j\omega$, the positions of the poles and zeros reveal interesting information about the behaviour of the system after an initial impulse (such as pulling the mass so that the string is extended or compressed in this example). The positions of the poles is of particular interest; figure 2.4 shows what behaviour is associated with the presence of a pole at a given point in the s-plane. Poles always appear in *complex conjugate pairs* (as do zeros), at equal distance above and below the σ axis. For this reason, only the top half of the plane is shown in the figure.

Poles to the right hand side of the $j\omega$ axis are associated with unstable responses which grow exponentially with time. This is neither a useful nor desirable result, and indeed, it is important when implementing an electronic or computer program simulation to ensure that there are no poles here. Poles which have a negative value of σ (being located on the left-hand-side of the $j\omega$ axis) exhibit more reasonable behaviour; the oscillation dies away as time progresses, and the simulated behaviour is that of a damped system, as if the mass is immersed in a viscous fluid, or a shock-absorber had been fitted to the spring. This is a more useful response, and closely models the observed behaviour of the first attempt program to simulate a spring and mass. However, the best solution for our purposes lies where the poles are *on* the $j\omega$ axis; these solutions oscillate sinusoidally indefinitely at a constant amplitude,

which is just the required behaviour for an oscillator. The frequency thus obtained is given by the distance away from the origin along the $j\omega$ axis. It is now required to transform this s-plane, which is a description of a *continuous* system, into one which can be modelled accurately by a computer which operates with *discrete*, or sampled time. This is the approximation which let down the first attempt, but now that the s-plane representation of the desired system is available, a more satisfactory solution is possible.

2.6.2 Moving from the s to the z plane

The attempt to model a simple oscillator by solving a second order differential equation is flawed because of the use of a delay to calculate the first and second differentials. The approximation in the case of the first differential was to assume that the velocity was essentially constant throughout a sample period, whereupon the velocity was calculated as the difference in position at subsequent samples divided by the inter-sample time. Of course, in the real system, the velocity is changing throughout this time, and the approximation is only true in the limit where the sample frequency is infinitely greater than the frequency at which the oscillator is running. It is necessary to understand the true nature of the delay element which was used in the calculation of velocities and accelerations; then the effect of such delays can be built into the computer model, and a practical implementation of a sinusoidal oscillator without the use of a look-up table will be possible.

Let us proceed by considering the difference between the Laplace transformations two signals which are identical except that there is a time delay between them. If we consider the first signal to be $x(t)$, the function giving the second is $y(t) = x(t-a)$ where a is the delay time. Recalling the way in which Laplace transforms are evaluated, it is perhaps intuitive that delaying the signal by an amount a is equivalent in integral terms to *advancing* the exponential function by an equal amount, and that since $e^{s(t-a)} = e^{-as}e^{st}$, the delayed signals Laplace transformation $Y(s) = X(s)e^{-as}$.

The properties of the z-plane take the effects of these finite delays into account. The z-plane is in fact identical to the s-plane, and any system represented in the z-plane has the same number of poles and zeros as it would in the Laplace s-plane representation. $z = e^{sT}$, where T is the sample rate of the system, and this has a distorting effect on where the points in the s-plane end up when represented in the z domain (see figure 2.5). The system is sampled, so only frequencies of between $\pm\frac{\omega T}{2}$ (the Nyquist limit) are usually considered. This horizontal strip of the s-plane is mapped *in its entirety* inside the unit circle in the z-plane. Any poles which represent damped oscillations are mapped inside this circle, and the more heavily damped the system is, the nearer to the origin the poles appear. Regions of the s-plane associated with instability, the right-hand half of the s-plane, are mapped to the area outside the unit circle. The poles of most interest to us, representing sustained, constant amplitude oscillations, are placed *on* the unit circle. If we simulate a system with a pair of poles on the unit circle, the angle θ represents the frequency of the oscillation, reaching the maximum frequency of the Nyquist limit when both poles appear at $z = -1 + j0$.

If we investigate our first, rather crude attempt at constructing an oscillator, it turns out that the position of the pole-pairs did not lie on the unit circle, but rather on a circle of radius $\frac{1}{2}$ passing through the origin and $z = 1 + j0$. This explains the fact that the damping increases as the sample rate is reduced, and also predicts that an error in *frequency* will occur as well as the damping effect. Let us now take these results into consideration and design a purely algorithmically-based oscillator which is better suited to our purpose.

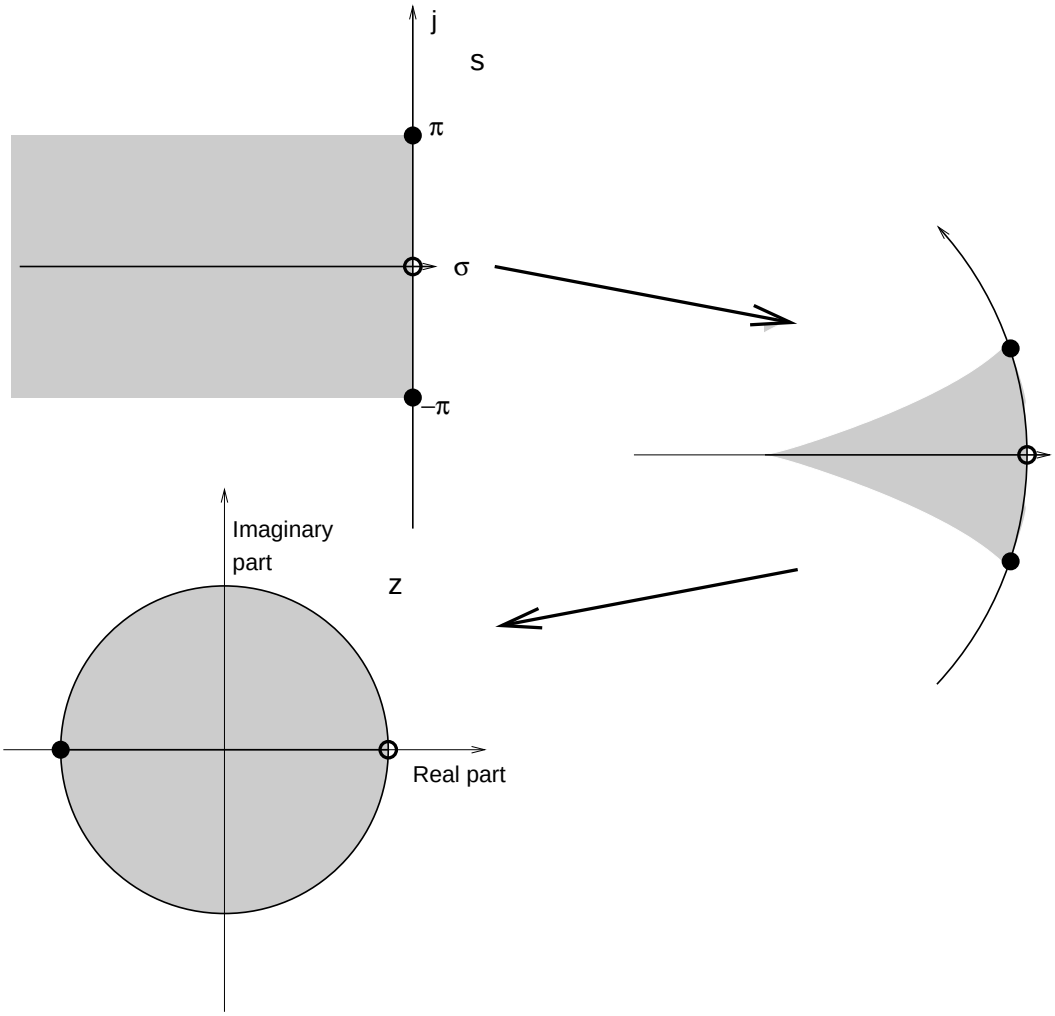


Figure 2.5: Mapping the s-plane onto the z-plane

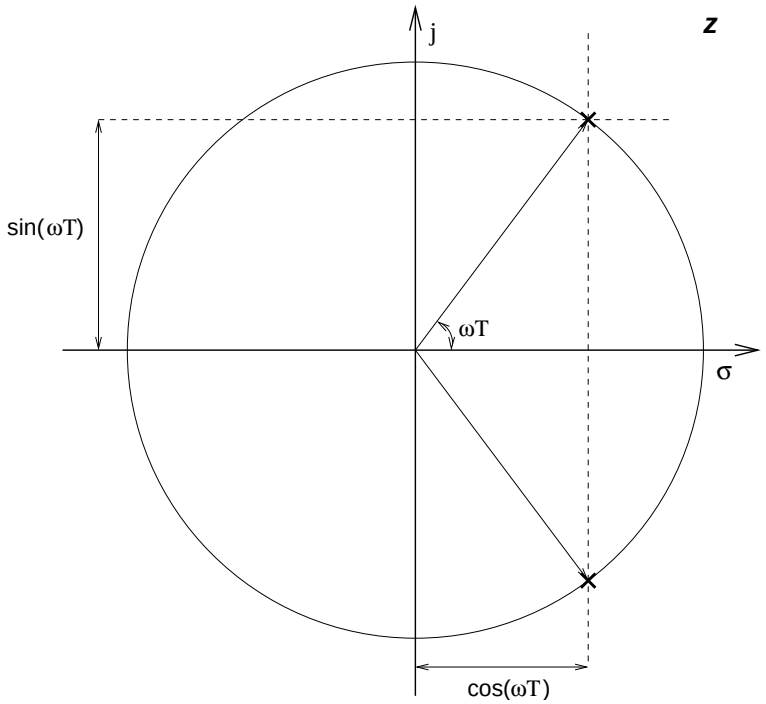


Figure 2.6: Position of Poles on the z-plane for an Oscillator

2.7 The Spring and Mass Revisited

In Section 2.6, an example oscillator was constructed which turned out to be flawed because invalid assumptions were made. The assumptions were that the oscillator's velocity and acceleration were essentially constant throughout the small time increment between samples. Armed with a better appreciation of the consequences of working with a system which contains time-sampled rather than continuous data, it is possible to avoid these errors.

In the previous example, we used a single-sample delay to determine "instantaneous" velocity and a further delay to provide an accleration estimate. In the z-domain, *delaying* a time function is equivalent to division by z (recall that $z = e^{st}$ and the consequences of delaying the time function on the s-plane result). For this reason, the delays which were used in the previous example (misguidedly) to permit the estimation of velocity and acceleration can be expressed as division by z. Texts dealing with discrete-time systems often represent a delay of one sample-time as a box labelled z^{-1}. It is now a matter of converting our oscillating function specification of two poles appearing on the unit circle into a function of z, and then implementing that function using the computer to perform the delays and mathematical operations as before.

At the location of the poles, the value of the required function of z is infinite. The position of the poles on the unit circle is determined by the required frequency, making an angle of $\pm\omega_0 T$ with the positive real axis, where ω_0 is the required frequency of oscillation in Hertz multiplied by 2π, and T is the period between samples: see Figure 2.6. The poles therefore end up at $z_1 = (\cos\omega_0 T, \sin\omega_0 T)$ and $z_2 = (\cos\omega_0 T, -\sin\omega_0 T)$. Since $e^{j\theta} = \cos\theta + j\sin\theta$, this means that this *transfer function* $H(z)$ can be written directly:

$$H(z) \quad = \quad \frac{1}{(z - e^{j\omega_0 T})(z - e^{-j\omega_0 T})}$$

$$= \frac{1}{z^2 - z(e^{j\omega_0 T} + e^{-j\omega_0 T}) + 1}$$
$$= \frac{1}{z^2 - 2z\cos(\omega_0 T) + 1} \quad (2.12)$$

since $\cos\theta = \frac{1}{2}(e^{j\theta T} + e^{-j\theta T})$. Now, $H(z) = \frac{Y(z)}{X(z)}$, where $y(n)$ is the output of the oscillator, and $x(n)$ is the input, or the initial impulse given the the system to set it in oscillation. Rearranging,

$$\begin{aligned} Y(z)(z^2 - 2z\cos(\omega_0 T) + 1) &= X(z) \\ z^2 Y(z) - 2z\cos(\omega_0 T)Y(z) + Y(z) &= X(z) \end{aligned}$$

$$z^2 Y(z) = X(z) + 2z\cos(\omega_0 T)Y(z) - Y(z) \quad (2.13)$$

Division by z is equivalent to a delay of one sample time, so multiplication by z is equivalent to taking the value of a function one sample-time hence. Transforming back into the reality of the time domain, we obtain

$$y(n+2) = x(n) + 2\cos(\omega_0 T)y(n+1) - y(n)$$

This is rather difficult to realise, because we need to know the values of the function $y(n)$ two samples before they have been calculated! However, we do not really care about a two sample delay in the output; as long as its presence is understood the effect it has on phase can be compensated by starting the oscillator at an appropriate time. Instead, we evaluate

$$y(n) = x(n-2) + 2\cos(\omega_0 T)y(n-1) - y(n-2) \quad (2.14)$$

using the following scheme, as shown in the code fragments of Figures 2.8 and 2.9. The header and C++ files replace those in the lookup-table example of Section 2.2 This expression has an input term, $x(t-2)$ which represents the excitation of the oscillator, whereas Figure **??** shows the input as $x(t)$. This tells us that the "pluck" which sets our oscillating string in motion should be applied to the oscillator two sample-periods before we need the output to be valid.[1] This is quite good enough for use in additive synthesis, where we are only interested in the oscillator's *steady state*.

2.7.1 Plucking the String

In fact, it is possible to cheat about how to start the oscillator when we need accurate control over amplitude and phase. It will be readily apparent from Figure **??** that the next output of the system is a function of the current output and the previous two outputs *only*. Instead of "ringing" the oscillator by applying an impulse at its input, the required two values of previous samples can be calculated directly an inserted into the state variables so that, as far as the algorithm is concerned, the system has arrived at that state as a result of calculating previous samples. This approach has the advantage that not only the amplitude of oscillation, but also the precise phase may be controlled easily, without any worry about how the oscillator may behave in its transient phase, shortly after being kicked into action by a short, sharp shock at the input, and before it settles down to its equilibrium state.[2]

[1] In some applications this is unsatisfactory. In Chapter 4 a method will be described which avoids this problem, when the use of the z-plane for the construction of filters is discussed.

[2] For the purpose of simulating the sound generated by acoustic musical instruments, eliminating this brief period of instability is actually undesirable. The transient phase is what gives the organ note its "chiff" or the *spicatto* violin its "bite". This will be addressed in Chapter **??**, but in considering the needs of additive synthesis, we seek to generate spectrally pure sine waves.

2.7.2 Generating a Complex Tone and Avoiding Instability

Building an oscillator in software using only two additions and two multiplications per sample might be regarded as a coup in processing terms, but this method is not without its disadvantages. All of the preceding analysis has assumed that the computer undertaking the calculations does not suffer from the rounding errors which are an unavoidable consequence of finite word length. Realistically, the computer would store a floating-point number in 64 bits, not all of which are available for the storage of the significant digits of the number itself. Some rounding, however small, must take place in the course of mathematical operations.

What is the effect of such rounding? The oscillator coefficients (i.e. those constants by which the delayed samples are multiplied) are rounded approximations of their true values, and these in turn dictate the positions of the poles in the z-plane. Moving the poles around the unit circle causes a change in frequency, but such a trivially small change as might result from this level of inaccuracy could not possibly upset even the most acutely pitched of musicians. A more serious problem is that, due to rounding error, the poles might not fall *on* the unit circle, but *slightly outside it*. Recall that the outside of the unit circle in the z-plane is mapped onto the right-hand side of the s-plane, and the possibility of a serious deficiency becomes apparent. Referring back to Figure 2.4, the presence of a pole-pair in this region results not in a stable, fixed-amplitude oscillation, but an oscillation which grows exponentially, approaching an infinite level. This problem is cumulative. Since the largest numbers your computer can represent are a lot closer to zero than infinity, the program will run for a short time and give up with a loud shriek and a message to the effect that an arithmetic overflow has occurred. An error in the opposite direction is almost as undesirable; the program runs, but the tone slowly dies away at a rate determined by the size of the rounding error. A complex tone will make use of many such oscillators running simultaneously, so the nett result would sound something like a well-tuned choir singing a single chord running out of air, partial-by-partial falling away as its oscillator reaches the limit of its staying power.

The state variables of this oscillator are reset every time the frequency is updated, but using double-precision floating-point arithmetic, for which an essentially all personal computers now contain fast hardware support, rounding errors are not likely to be a problem over quite a large number of samples. This trick enables a fast algorithm to be used for the vast majority of the work, with the slow, accurate algorithm being used once in a while to ensure that everything is kept in order.

Code Segment 2.2 The Oscillator Class

```
//
// An oscillator produces a cosine wave at a given frequency and
// amplitude and adds the result to a buffer of a given length.
//

#ifndef OSCILLATOR_HH
#define OSCILLATOR_HH

#include <iostream> // just for messages – removed in "production version"
using namespace std;

#include "generator.hh"
#include "buffer.hh"

class Oscillator : Generator {
private:
  class LUT {
  public:
    float tbl[16385];
    LUT();
  } ;
  static LUT lut;
  int phase_acc;
  int phase_inc;
  double amplitude;
public:
  // Convenience methods:
  void set_freq(double freq);
  void set_amp(double amp) { amplitude = amp; }
  void gen (Buffer *buf) { gen(buf, buf->getLen()); }
  // Essential methods:
  void set(double freq, double amp) { set_freq(freq); set_amp(amp); };
  Oscillator(double freq, double amp) {
    cout << "Creating osc: freq " << freq << " amp " << amp << "\n";
    set(freq, amp);
    phase_acc = 0;
  };
  void gen(Buffer *buf, int count);
} ;
#endif
```

Code Segment 2.3 Oscillator Methods

```
// oscillator.cc
#include <math.h>
#include <iostream> // just for messages – removed in "final version"
using namespace std;

#include "oscillator.hh"
#include "buffer.hh"

Oscillator::LUT Oscillator::lut;

Oscillator::LUT::LUT() {
    for (int i=0; i<16384; i++)
      tbl[i] = cos(0.5*M_PI*i / 16384.0);
    cout << "generated lookup\n";
}

void Oscillator::gen(Buffer *buf, int count)
{
  double luval;
  int i=0;

  while (count--) {
    int idx = phase_acc >> 8; // get rid of fractional part

    // get value from table. Which quadrant is it in? (64Ksample tbl)
    if (idx < 32768) { // 1st or 2nd quadrant
      if (idx < 16384) luval = lut.tbl[idx];
      else luval = -lut.tbl[32768 - idx];
    } else { // 3rd or 4th quadrant
      if (idx < 49152) luval = -lut.tbl[idx - 32768];
      else luval = lut.tbl[65536 - idx];
    }

    // Adjust amplitude of lookup result and sum to buf
    buf->data[i++] += amplitude * luval;

    // Increment and wrap phase accumulator
    phase_acc += phase_inc; phase_acc %= (65536 << 8);
  }
}

void Oscillator::set_freq(double freq)
{
  // set up the phase increment value to reflect the frequency.
  // 32Khz sample rate, 8 bits fractional, and 65536 samples LUT

  phase_inc = static_cast <int> (256 * 65536 * freq / 32000);
  cout << "phase_inc set to " << phase_inc/256.0 << " samples.\n";
}
```

Code Segment 2.4 An Audio Buffer Class

```
// buffer.hh

#ifndef BUFFER_HH
#define BUFFER_HH

#include <string.h>

class Buffer {
private:
  int length;
public:
  float *data;
  Buffer(int size) { data = new float[size]; length = size; };
  ~Buffer() { delete data; };
  void clear() { memset(data, 0, length * sizeof(float)); };
  int getLen() { return length; };
  void play(int fd);
};

#endif
```

Code Segment 2.5 The Audio Buffer Class' play() method

```
// buffer.cc

#include <unistd.h>
#include "buffer.hh"

void Buffer::play(int fd)
{
  short sb[length];
  // cast buffer to shorts
  for (int i = 0; i < length ; ++i)
    sb[i] = static_cast<int>(32767.0 * data[i]);

  // ...and play it.
  write(fd, sb, 2*length);
}
```

Code Segment 2.6 Using Lookup-table Oscillators

```
// main.cc
//
// Generates a one-second burst of a square wave synthesised from its
// first three harmonics

#include <iostream>
using namespace std;

#include <unistd.h>
#include "oscillator.hh"
#include "buffer.hh"
#include "audio_open.hh"

main()
{
  Oscillator o1(440.0*1, 0.5/1); // Fundamental, f0
  Oscillator o2(440.0*3, 0.5/3); // 3.f0 has rel amplitude 1/3
  Oscillator o3(440.0*5, 0.5/5); // 5.f0 has rel amplitude 1/5

  Buffer *b = new Buffer(32000); // Output buffer is 32Ksamples

  int fd = audio_open("/dev/dsp", 16, 1, 32000);

  // Build square wave.
  b->clear();
  o1.gen(b);
  o2.gen(b);
  o3.gen(b);

  // Write it.
  b->play(fd);
  close(fd);
}
```

Code Segment 2.7 A First Attempt to Model an Oscillating Mass

```
/* DIFFOSC.C */
/* Recursive Oscillator,
   does not take quantisation effects into account */

#include <stdio.h>

main()
{
double xn1, xn2;
double xn, alpha, konm, T;
int sample, i;

   printf("Sample Period? (Try 0.1) >");
   scanf("%lf", &T);
   konm = 0.3;
   xn1 = 1.0;
   xn2 = 1.0 - T*T*konm;
   alpha = 1.0/(1.0+T*T*konm);
   sample = 0;
   i = 30;
   while (i) {
      xn = alpha*(2.0*xn1-xn2);
      if (xn1 > xn && xn1 > xn2) {
         printf("Max at sample %d:%10.5f\t",sample, xn1);
         --i;
      }
      if (xn1 < xn && xn1 < xn2) {
         printf("Min at sample %d:%10.5f\n",sample, xn1);
         --i;
      }
      xn2 = xn1;
      xn1 = xn;
      ++sample;
   }
}
```

Code Segment 2.8 Prototypes and Data Structures for a Recursive Oscillator

```
//
// An oscillator produces a cosine wave at a given frequency and
// amplitude and adds the result to a buffer of a given length.
//

#ifndef OSCILLATOR_HH
#define OSCILLATOR_HH

#include <iostream> // just for messages – removed in "production version"
using namespace std;                                                    10

#include "generator.hh"
#include "buffer.hh"

class Oscillator : Generator {
private:
  double cos_omega_t;
  double state[2];
  double amplitude;
public:                                                                 20
  // Convenience methods:
  void set_freq(double freq);
  void set_amp(double amp) { amplitude = amp; }
  void gen (Buffer *buf) { gen(buf, buf->getLen()); }
  // Essential methods:
  void set(double freq, double amp) { set_freq(freq); set_amp(amp); };
  Oscillator(double freq, double amp) {
    cout << "Creating osc: freq " << freq << " amp " << amp << "\n";
    state[0] = state[1] = 1.0;
    set(freq, amp); // also adjusts state[1] appropriately             30
  };
  void gen(Buffer *buf, int count);
} ;
#endif
```

Code Segment 2.9 Methods for a Recursive Oscillator

```
// oscillator.cc
#include <math.h>

#include "oscillator.hh"
#include "buffer.hh"

void Oscillator::gen(Buffer *buf, int count)
{
  double oscval;
  int i=0;                                                    10

  while (count--) { // Calculate samples based on precedents
    oscval = 2.0*cos_omega_t*state[0] - state[1];
    state[1] = state[0];
    state[0] = oscval;
    // Adjust amplitude of lookup result and sum to buf
    buf->data[i++] += amplitude * oscval;
  }
}
                                                              20
void Oscillator::set_freq(double freq)
{
  // set up the phase increment value to reflect the frequency.
  // 32Khz sample rate. Force initial state to current phase angle

  double omega_t = freq*2.0*M_PI/32000.0;
  cos_omega_t = cos(omega_t);
  // Fool the oscillator into thinking its always been running at
  // the new frequency by calculating the previous state from this one
  state[1] = state[0] > state[1] ? // is the output rising? if so...  30
    cos(2.0*M_PI - acos(state[0]) - omega_t) : // 3rd/4th quad
    cos(acos(state[0]) - omega_t); // else 1st/2nd quadrant
}
```

Chapter 3

Fourier Transformation and Synthesis

IN PREPARATION

So far, the idea of considering a musical tone to be built up of constituent partials has been an implicit assumption in all the preceeding discussion. When an oscillating string is considered there is a clear fundamental mode with maximum vibration amplitude of the string at its mid point upon which other residual oscillations can add at integer multiples of this basic frequency. It is less clear that complex musical sounds can be represented in this way. In this section the representation of sound in terms of inharmonic continuous spectra is presented. This viewpoint appears at first to make our study of sound impossibly complicated, in fact it offers a powerful simplification which begins to give us clues as to how we ourselves hear, perceive and recognise sound.

3.1 An introduction to spectral Transforms

The first concept to establish is the idea of the repetition rate of a sound signal. There is little repetition in the conversation noise at a party. This is partly because there are overlapping sound sources, people talking on top of each other, and partly because only rarely, if at all, would the conversation topic be repeated exactly. Not surprisingly, the spectrum of such a sound signal contains power at all frequencies within a given band between 50Hz and 4kHz as shown in Fig 1a and would change with time. If we monitored this spectrum as the party fell silent when one person began to sing the spectrum of the sound would simplify. There would be power only at specific frequencies and these would form a complex but precise pattern. If the sound source became highly repetitious, for example the continuous singing of a vowel, we would observe the spectrum to begin to conform to our understanding of harmonics on stringed instruments and resonant air columns in organ pipes, as shown in Fig 1b.

The more continuous and repetitive the sound, the more the sound power is confined to precise harmonically related frequencies. Adding some mathematics at this point we could say of the sound was changing its nature on a timescale of T seconds then we would expect the frequency spread of its harmonics to be about 1/T Hz. This is similar, but not the same, as saying because a sinewave repeats its oscillation every Ts seconds its frequency is 1/Ts Hz. We are describing the effect of changing, or disturbing our 1/Ts Hz sine wave every T seconds. The power in the sound signal then becomes spread by 1/T Hz around the orginal harmonic.

Since the nature of music is to convey time changing patterns of sound we would

expect typical spectra to be continous. We begin, however, with the simple case of a non-sinusoidal repetitive signal.

3.1.1 Fourier Series and the Fourier Transform

The Mathematician, J?? Fourier (18??-19??) set out the mathematical principles underpinning the concepts described above, though it is interesting to note that his enquiries were motivated by thermodynamics and heat flow through steam engines rather than acoustic waves through the air.

A starting point from trying to get a physical insight into Fourier's approach is to consider the simplist, non-trivial case of a non-sinusoidal repetitive signal as shown in Fig 2.

Need to refer abck to basic signal theory here from chapter 1.

The voltage-time signal shown above could represent the microphone output of a reptitive sound source. Because it repeats with a period T seconds we can express the frequency content of this signal as the summation of a series of harmonics being simple interger mulitples of the fundamental frequency 1/T Hz. The harmonics may add in with arbitrary amplitude and phase but it should be clear from the outset that the contribution of each harmonic to the final signal is unique. For example, a lack of power in the fifth harmonic can not be compensated for by a mixture of third and seventh harmonics. Each component in the Fourier series summation is vitally present or absent and adds independently in linear systems to reconstruct the orginal signal. No other combinations of harmonics could represent this signal. Mathematically we would say that the harmonic sine functions are orthgonal to one another. This relationship to geometrical axes is useful to pursue at this point. In a coordinate system defined by x, y and z values there is no offset in the z dimension which could be described by a combination of x and y values because x, y and z are orthogonal.

The spectrum of the signal in Fig 2 is shown in Fig 3.

Chapter 4

Subtractive Synthesis

Additive synthesis enabled us to generate any sound we could think of, but not necessarily any sound we *liked*. It was difficult to control the sounds produced by an additive process for two main reasons. The first was that the quantity of data required for subtle control of timbral changes during the evolution of a note would become prohibitively large for a real system; but the second, possibly more important reason was that, in spite of the flexibility offered by additive processes, there is no correlation between the way an additive synthesiser works and the way an acoustic musical instrument produces its sound. This information is imparted to the additive synthesiser by the programmer, who must laboriously calculate the synthesis parameters by, for example, Fourier Analysis of an already-existing sound's spectrum. On the positive side, the parameters supplied to an additive synthesiser might have some intuitive meaning; partials with higher frequencies add "brightness" to the sound, and even and odd harmonics effect the timbres in their own particular ways: but defining a new sound *from scratch* using additive techniques is indeed a daunting task. Perhaps a more suitable method might be found by examining the way in which more conventional instruments produce sound, and by deducing a model which can behave similarly with far fewer control parameters than the additive synthesisers of Chapter 2.

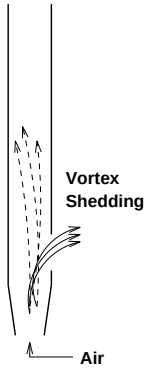


Figure 4.1: Pipe

Figure 4.1 shows an organ pipe emitting sound. Air is forced in at the base, and passes over a tongue near the bottom of the pipe. As the air is forced in, it does not flow smoothly, but forms a turbulent vortex which alternately forces air up the pipe and outwards from the hole below the tongue. Without the pipe fitted, this alternation or "vortex shedding" would be an essentially random process, resulting in the hissing sound associated with the escape of compressed air. When the pipe is added, it resonates in such a way that, after a short "chiff" of random noise at the beginning of the note, a steady oscillation is established. Different pipes produce different timbres. The pipe is acting as a "filter" by taking the white-noise generated by the escaping air, and preferentially amplifying a select group of frequencies from the very wide range available from the noise source. Instead of producing a complex musical tone by the *addition* of many individual partials, the organ pipe starts with the wide range of frequencies available from the vortex shedding process, and enhances just a few of them to produce a musical note. Scaling the stimulus and result signals to be the same amplitude, we might consider this process as *subtractive*; the unwanted partials are effectively rendered inaudible

by the selective enhancement process.

4.1 Digital Filtering

In Section 2.7, the position of a pair of poles was determined in order to construct an *oscillator* (Figure 2.6 on page 21). The algorithm subsequently realised (Code Segment 2.9) was in fact a special case of a digital filter. We explained that the oscillator worked by "ringing" after it had been "struck" by an initial impulse. However, one might offer the alternative explanation that the initial input, an impulse containing *all* frequencies, is being *filtered* so that it contains only the fundamental sinusoid. It turns out that, by examining the position of the poles on the z-plane, one can predict the frequency response of the associated digital filter without recourse to higher mathematics.

The question arises: is it easy (or, indeed, possible) to calculate the *frequency response* of a system by looking at the position of the poles in the z plane? Conversely, can any desired frequency response be obtained by *placing* poles on the z plane at easily calculable positions? In order to answer this question, we must recall that the poles were the result of a *transfer function* (equation 2.12 on page 22) which has the general form:

$$\begin{aligned} H(z) &= \frac{Y(z)}{X(z)} \\ &= \frac{A(z-z_1)(z-z_2)\dots}{(z-p_1)(z-p_2)\dots} \qquad (4.1) \\ &= \frac{\sum_{n=0}^{n_z} a_n z^{-n}}{1-\sum_{n=1}^{n_p} b_n z^{-n}} \qquad (4.2) \end{aligned}$$

where the constant A is sometimes referred to as the *system gain*, n_z is the number of zeros, and n_p is the number of poles. In Chapter 2, equation 2.12 had a constant value for $Y(z)$; $X(x)$ was a quadratic which was expanded out rather than expressed in the form of the product-of-factors shown in equation 4.2. The poles, represented by crosses, occur at conjugate positions (i.e. equally spaced above and below the horizontal σ axis), as do any zeros, usually represented by circles on z-plane diagrams. The example of Chapter 2 was multiplied by z^2 in order to make it possible to implement as a computer program; this is the same as setting $Y(z) = 1(z-0)(z-0)$, tacitly placing two co-incident poles at the origin.

The procedure to calculate the frequency response associated with a given pattern of z-plane poles and zeros isn't too difficult. The system is dealing with discrete-time (sampled) data, so Nyquist tells us that the maximum sensible frequency we can consider[1] is half the sampling frequency, $\frac{f_s}{2}$.

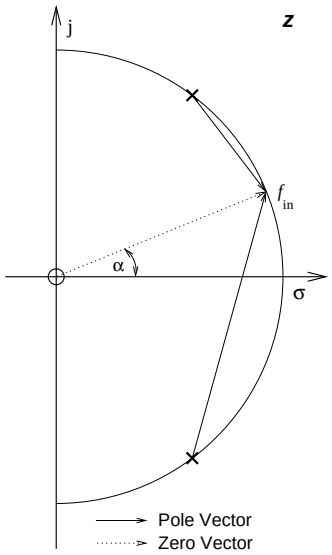
- Take a point on the unit circle at angle α. This corresponds to an input frequency of $\frac{\alpha}{2\pi} f_s$; that is to say that the frequency starts at 0Hz at $z = 1$ and increases through to the Nyquist frequency by the time the point has traversed the upper semi-circle to reach $z = -1$.

- Draw vectors to this point from all the zeros on the z plane. Multiply all of these vectors together[2] to arrive at a product vector $\mathbf{Z}$.

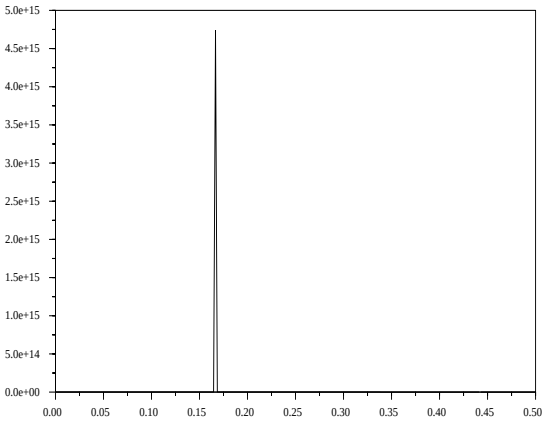
[1] Of course, the system is capable of processing any *range* of frequencies which does not span a multiple $\frac{2n+1}{2} f_s (n, \text{integer} \geq 0)$, but we know of no *audio* signal processing systems which take advantage of this!

[2] Multiplication of vectors is achieved by multiplying their *lengths* but adding the *angles* which they make with the positive σ axis, moving in an anti-clockwise direction

- Repeat for the vectors connecting the point on the unit circle with all of the *poles* on the z-plane, calling the result $\mathbf{P}$.
- The response of the system to an input at the frequency considered is $\frac{\mathbf{Z}}{\mathbf{P}}$



(a) Poles & Zeros



(b) Frequency Response

Figure 4.2: Frequency Response Calculation

Figure 4.2 shows this calculation in action. In this simple example, there are only two pole-vectors and two zero-vectors (both zeros lie at the origin, so the vectors overlap in the Figure). The pole-zero diagram in Figure 4.2(a) is the one used in Chapter 2. Although only the right-hand side of the circle appears in the interests of compactness, the entire upper semi-circle would be used to arrive at the frequency response curve. Note that this rather useful result gives us not only the amplitude of the response, but simultaneously gives the phase (the result of the division also being a vector). The frequency response graphs such as the one in

Figure 4.2(b) are simply plots of the length of this vector as the frequency point moves from a position corresponding to an input frequency, f_{in}, of zero around the unit circle until it reaches $f_{\text{in}} = \frac{f_s}{2}$. The horizontal axis on the frequency plots are normalised into the range $[0, 0.5]$, so this must be multiplied by f_s to arrive at the real frequency.

The very sharp peak in the graph at one-third of the Nyquist frequency occurs when the f_{in} marker reaches the pole on the unit circle. The distance from the calculation point to the pole is then zero, so the $\mathbf{P}$ vector becomes zero in magnitude. This gives rise to an amplitude response which rises towards infinity; the fact that it reaches a trifling 5×10^{15} on this graph is due only to arithmetic inaccuracies in the computer used to calculate it. No wonder that the system represented by this pole-zero diagram is so capable at selecting the single sinusoidal oscillation required for additive synthesis! For our current purposes, we shall require better-behaved systems with more moderate peaks.

4.1.1 Designer Filters

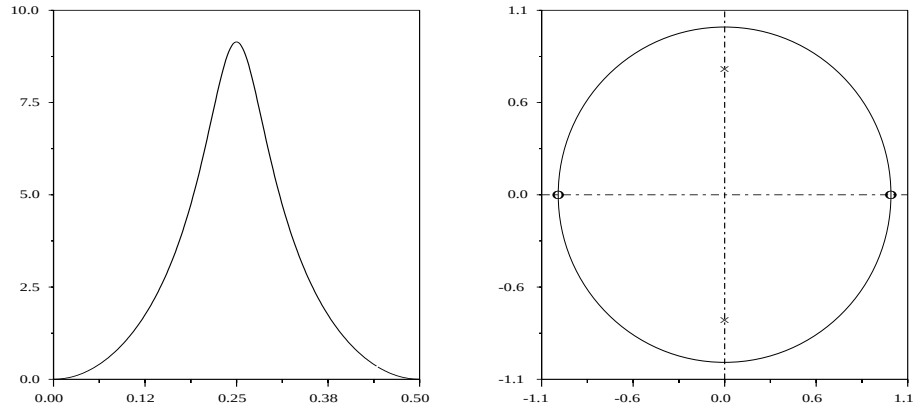
For many purposes, the generic filter designs shown in Figure 4.3 and 4.4 can be modified to accomplish the job in hand. Figure 4.3(a) has poles placed on the imaginary axis a small distance inside the unit circle, and has zeros at the points corresponding to $f_{\text{in}} = 0$Hz and $\frac{f_s}{2}$. The overall frequency response shows that no signal is passed at these extremes of input frequency, but that there is a larger gain at a frequency of $\frac{F_s}{4}$ The position of this peak is dictated by the angular position of the poles, and the peak can be made sharper (by moving the poles closer to the circle) or broader (by moving them towards the origin).

Figure 4.3(b) shows the opposite type of filter: a band-reject filter. This sort of filter is often used in instrumentation to remove annoying noise signals at 50Hz or 60Hz which may have been picked up from the mains power supply. The example shown here has been designed to reject a narrow range of relatively low frequencies, so the zeros lie on the unit circle making a only a small angle with the real axis. At frequencies which are far from the rejection frequency, the pole which lie just inside the unit circle very close to the zeros has a vector which almost exactly cancels out the zero vector. As the frequency is reduced, the measurement point passes towards the zero, eventually passing through it. At this point, even though the pole vector is very small (which would indicate a very large gain), the zero vector has a length which is absolutely zero, so no signal passes from the filter's input to its output. The filter shown has its zeros at an angle of $\frac{\pi}{12}$ or 30°, which will eliminate frequencies of $\frac{f_s}{24}$. The distance of the zeros from the origin is 0.995, and that of the poles 0.95. This gives the very sharp rejection shown in the associated frequency response graph.

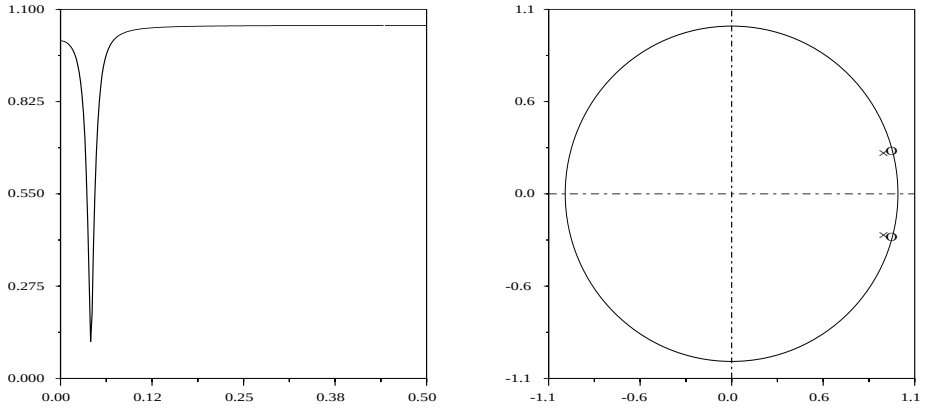
Figures 4.4(a) and 4.4(b) show how simple low- and high-pass filters can be formed with real poles and zeros. These are sufficient if all that is required in general emphasis of higher or lower frequencies, but much more advanced designs yielding far better filters of only slightly increased complexity will be demonstrated in Section 4.3.

4.2 Building a Filter

In this section we demonstrate how to build a bandpass filter starting off from the z-plane representation and finishing off with a working C program. Let's construct a band-pass filter with the frequency response similar to that shown in Figure 4.3(a), but designed to produce a 1kHz tone from wide-band noise at the relatively modest sample rate of 16000 samples-per-second.



(a) Simple Band-pass Filter



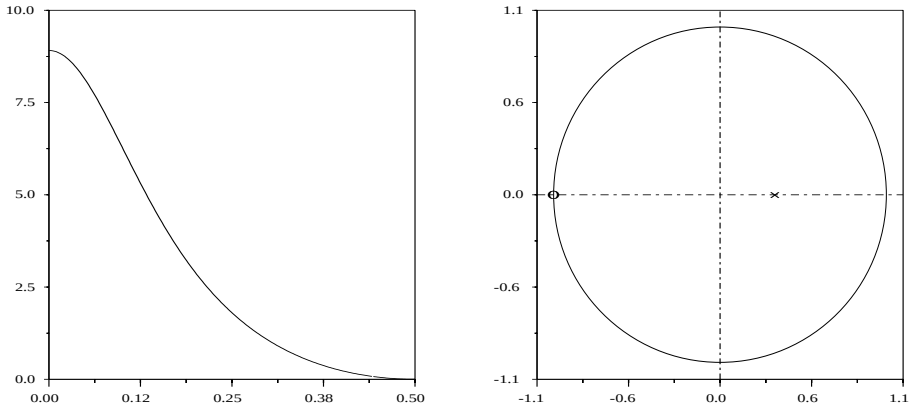
(b) Simple Band-reject Filter

Figure 4.3: Some Digital Filter Examples (i)

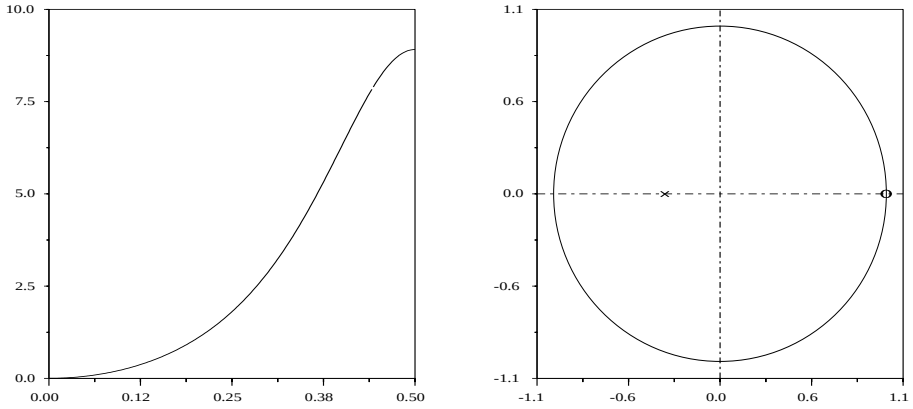
Recalling that the centre-frequency of the filter varies with the angle around the semi-circle, starting at 0Hz at $z = 1$ and increasing to 8000Hz when $z = -1$ (if the sample frequency is 16kHz), it will be evident that the angle required between the poles and the horizontal axis is $\frac{1}{8} \times 180°$, or $(\pi/8)$ radians. Choosing a position close to the unit circle will produce a sharper peak in the filter response, so for a start, we'll take a distance of 0.95 from the origin. Using a calculator with trigonometric functions, its easy to evaluate the positions of the poles p_1, p_2:

$$\begin{aligned} p_1 &= 0.95\left(\cos\frac{\pi}{8} + j\sin\frac{\pi}{8}\right) \\ p_2 &= p_1^* \\ &= 0.95\left(\cos\frac{\pi}{8} - j\sin\frac{\pi}{8}\right) \end{aligned} \tag{4.3}$$

Plugging the numbers into equation 4.3 gives the result (to seven significant



(a) Simple Low-pass Filter



(b) Simple High-pass Filter

Figure 4.4: Some Digital Filter Examples (ii)

figures) $p_1 = 0.8776856 + 0.3635493j$ and $p_2 = 0.8776856 + 0.3635493j$.

Now to convert this to a transfer function. Placing zeros at ± 1 is easy; $z^2 - 1$ has zero value at these points. The quesion then arises: what function has zero value at p_1 and p_2? This function will be the denominator of the transfer function. One can immediately write down the result in factorised form: $(z + (-p_1))(z + (-p_2))$. Unfortunately, this expression involves complex numbers, and its hard to see how this relates to an audio synthesis situation where the numbers stored on the disk or the numbers sent out to the Digital-to-Analogue converter are certainly real. However, it was previously asserted that poles and zeros appeared in complex conjugate pairs, and therein lies the solution. Considering the general case where there is a pole at $-p = x + jy$:

$$\begin{aligned}(z+p)(z+p^*) &= (z+x+jy)(z+x-jy) \\ &= z^2 + z(x-jy) + (x+jy)z + (x+jy)(x-jy)\end{aligned}$$

$$= \quad z^2 + 2x \cdot z + (x^2 + y^2) \tag{4.4}$$

Equation 4.4 *has no imaginary part*, so it is easy to implement using the same techniques as the oscillator in Chapter 2.

Taking our particular example, where $x = -0.923900$ and $y = -0.382683$ in Equation 4.4, the transfer function obtained is:

$$H(z) = \frac{z^2 - 1.00000}{z^2 - 1.7553711z + 0.9025} \tag{4.5}$$

Recall that a delay of one sample is equivalent to division by z (i.e. multiplication by z^{-1}), so to obtain the transfer function in a form which is suitable for direct reailisation, both the neumerator and denominator of Equation 4.5 are multiplied through by z^{-2}:

$$H(z) = \frac{1 - z^{-2}}{1 - 1.7553711z^{-1} + 0.9025z^{-2}} \tag{4.6}$$

When a data-flow diagram for an *oscillator* was presented (Figure **??**), the rather special case of all the zeros being at the origin was invoked. This meant that the zeros part of the transfer function reduced to a pure delay, so that the two poles could be implemented using a feed-back structure. The band-pass filter presented here has a numerator which is also a polynomial in z, so it is necessary to introduce *feed-forward* terms. Since the order of the numerator polynomial is the same as the denominator, it is possible to economise[3] and use the same "delay elements" for them both.

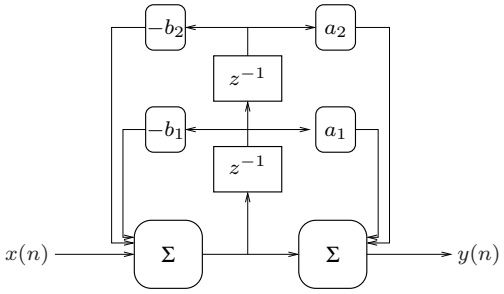


Figure 4.5: Band-Pass Filter Realisation

In Figure 4.5, the delay elements for the feed-forward and feed-back parts of the filter have been combined.

The maximum gain of this filter occurs when the input frequency is 1kHz, so to scale the output and avoid clipping, we should calculate this gain so that the wideband noise can be applied at a reasonable amplitude. Calcuating the pole and zero vectors at $\alpha = \pi/8$ and evaluating the gain gives:

$$\begin{aligned} \vec{z}_1, \vec{z}_2 &= (0.0761205 - 0.3826834j), (-1.9238795 - 0.3826834j) \\ \vec{p}_1, \vec{p}_2 &= (-0.0461940 - 0.0191342j), (-0.0461940 - 0.7462327j) \\ \text{Gain} = \left|\frac{\mathbf{Z}}{\mathbf{P}}\right| &= |20.434516 + 1.264956j| = 20.473631 \end{aligned} \tag{4.7}$$

[3]This is an important elementary structure called the "canonical bi-quadratic section": canonical because it is the minimum possible configuration to achieve this end; bi-quadratic because both the numerator and denominator of its transfer function are quadratic.

4.3 Classic Filters

So far, we have considered the effect on a system's frequency response of placing its poles and zeros at arbitrary points on the s (and hence z) plane, but it is now time to ask the question, "is this the best possible place to put these poles and zeros to obtain a particular characteristic?" To obtain a better understanding of how to get the best frequency response to suit a particular need, we shall return to the s plane. There are two reasons for this: the s plane is more analogous to "real-life" (it describes continuous, rather than sampled systems); and the mathematics required for filter design historically arose from the consideration of these systems rather than the more recent discrete-time digital ones.

Designing a first order filter (one with a single pole or zero which occurs on the real axis of the s plane) is a simple matter. Taking low-pass filters as an example, one simply defined the point at which higher frequencies should start to become attenuated, and then places the zero at the appropriate distance from the origin on the negative real axis. In mechanical terms, this is just like submerging the oscillating mass in section 2.7 in a bath full of fluid. If the fluid is thick, with the viscosity of treacle for example, only very low frequency attempts to move the piston up and down are unaffected, whereas if the fluid has the viscosity of air, relatively high frequency oscillations are possible before the movement is resisted. When constructing an oscillator, poles are placed on the imaginary axis in the s plane (equivalent to the unit circle in the z plane) so that the system is oscillating in a vacuum, uninhibited by anything like air resistance, and may therefore continue to oscillate indefinitely.

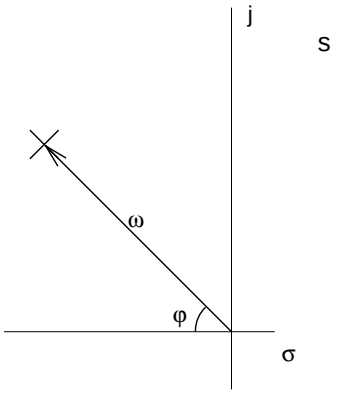


Figure 4.6: f & Q

With a second-order system (one with a pair of poles) like the oscillator and the simple filters considered in section 4.2, it is possile essentially to vary the viscossity of this fluid from nothing at all (poles on the imaginary axis in the s plane, or the unit circle in the z plane) to rather thick (poles on the real axis in the s plane). Filters with the same *cutoff frequency* now have different amounts of *damping* associated with them. It is necessary to formalize the concept of damping, because making more complex filters to approximate a desired frequency response is really just a matter of finding where to place poles on the s plane so that they represent particular frequencies with particular *damping factors*.

The property of a system which reflects its damping factor is its Q, *quality factor*, or, less frequently, the *magnification factor*. Whilst this has a strict mathematical origin, it can be thought of in more than one way depending upon the situation which is under consideration. Equivalently, the Q of a system is:

- the ratio of the magnitude of oscillation of the system to the magnitude of the source of oscillation, when the source is oscillating at the system's resonant frequency;

- the ratio of the resonant frequency to the bandwidth of the system, where bandwidth is defined as the range of frequencies between which the system's oscillation falls short of the maximum oscillation by a factor of no more than $\sqrt{2}$;

- 2π times ratio of the peak energy stored in the system to the energy lost into the damping medium in each complete cycle.

A *higher* value of Q implies *less damping*, and a *sharper peak* in the frequency response. Figure 4.6 shows how a pair of poles can be placed onto the s plane given the desired characteristic frequency and damping factor. The other pole is placed at the conjugate position, an equal distance below the axis. The frequency is given by the distance ω from the origin to the pole divided by 2π, and the Q is determined by the angle marked ϕ: $Q = 1/(2\cos\phi)$. The minimum value of Q occurs when ϕ is zero, and the pole is on the real axis. In this case, $Q = \frac{1}{2}$. In the case of the oscillator, where the pole was placed on the imaginary axis, $\cos\phi$ reduces to 0 and Q becomes ∞.

Having transformed from the s-plane to the z-plane, things become rather less clear cut, but broadly speaking, poles *well inside* the circle characterise systems with *low* values of Q, and poles *close* to the unit circle charaterise systems with *high* values of Q. We shall see later how to map exactly the postions of s-plane poles onto the z-plane, and also learn why an approximation is usually used in practice. Before that, it is necessary to answer the quesion "what is the best location for n poles to give a particular filtering function?"

4.3.1 Choosing the Best Kind of Filter

Having limited the order (i.e. number of poles) in a filter, it is no longer possible to build one which is "perfect" in that it completely removes all partials above (or below) a certain frequency whilst leaving the others untouched. Instead, it is necessary to answer the question "what is the best perfomance obtainable in terms of P" where P is some property of the filter. By choosing the citeria for the best P, we are in fact selecting a particular equation is s (and hence z) which will describe the filter's behaviour.

It turns out that *any* filter can be represented by connecting second-order filters in series, with perhaps an additional simple first-order filter if the total order is odd. For example, a 7th-order filter can be made from three second-order filters connected in series with an additional first-order one to make the filter 7th-order overall. A second-order filter is represented by a pair of poles on the s-plane, and consequently has a Q and a *cut-off frequency* associated with it. By mixing and matching different Qs and cut-off frequencies, it is possible to achieve the filter response which is best in the particular way which is considered important. In fact, there are many such parameter sets which can be optimised, and the filters which do a particular job best for a given number poles are usually named after the mathematicians who invented (discovered?) them. One may demand that the filter *delays* all signals by exactly the same time, regardless of their frequency; such a filter would be called a linear-phase filter because the phase delay experienced by a partial passing through it would be proportional to the partial's frequency. Thompson-Bessel filters perform best in this respect, but are rather poor at cutting off frequencies close to the cut-off point. Amongst the sharpest cup-offs for a given number of poles are produced by Cauer or eliptic filters, but these do particularly nasty things to the phase of a harmonically rich input signal, and can often produce subjectively undesireable brittleness of tone. They were developed by the German telephone company to cram as many telephone converstations down a wire as possible for a given complexity of circuit, and it is reported that when an American competitor found out that filters of such performance had been demonstrated, it despatched its entire research and development staff to the library with instructions not to return until they understood how they worked!

These types of filter are rather specialist, although they may be provided by

signal processing packages, so the reader is encouraged to experiment. For present purposes, it is sufficient to focus on the two most common types of filter and the associated placement of poles. The most common type is the *Butterworth* filter; it has the best possible rejection of undesired signal elements whilst retaining a *maximally flat* reponse to desired signals. The second sort of filter is the *Chebychev* filter, which sacrifices a little of the pass-band flatness of the Butterworth to provide a *maximally steep cut-off*. This kind of filter is particularly useful, because one can improve further the steepness of the cut-off by compromising the flatness of the pass-band, without increasing the number of poles in the filter. Figure 4.7 shows the responses of a 6th-order Chebyshev (solid line) and Butterworth (dotted line) filter; the former has been allowed a degree of pass-band ripple, resulting in a faster cut-off.

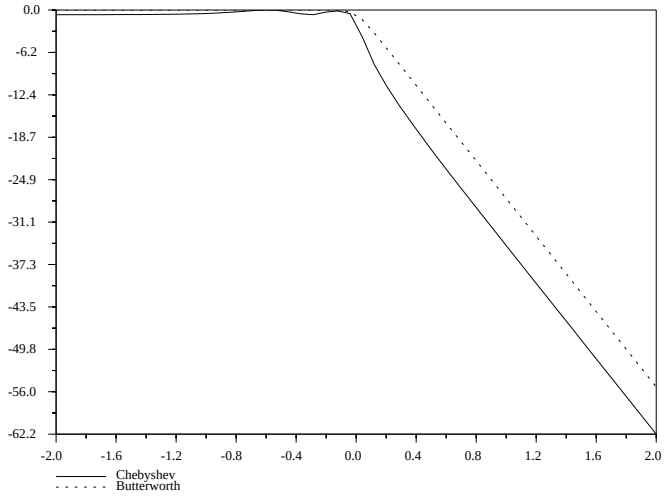


Figure 4.7: 6th-order Butterworth & Chebyshev Filter Response

Both the horizonal and verticle scales on Figure 4.7 are logarithmic, so "0.0" on the frequecy axis represents the 1Hz cutoff frequency, -1 represents 0.1Hz, 1 represents 10Hz and so on. This makes it easier to see how the power of the output signal falls off with the same gradient for each type of filter, but that the transition between the pass-band and the stop-band is more abrupt for the Chebyshev. In this case, the Chebyshev filter was permitted to have 1dB ripple in the pass-band, and this can also be seen in the figure. Notice that although the ripples become closer together as the frequency increases, their size does not alter, and that there are half as many ripples as the order of the filter (3 ripples for a 6th-order filter). If a steeper roll-off is required, then the order of the filter needs to be adjusted; increasing the filter's makes the roll-off steeper, and reducing it makes the roll-off shallower.

In the sections that follow, only the construction of a 1Hz low-pass filter is considered. Whilst this is something of a restriction, it is usually easier to design such a "normalised" filter, and then to transform the parameters so obtained to achieve the desired response.

4.3.2 The Butterworth Filter

The Butterworth filter provides the sharpest cup-off subject to the best possible smoothness in the pass-band frequency response, so where are the s-plane poles placed to satisfy these criteria?

Since it is required that the pass-band have no ripples, it is necessary to give all of the second-order sub-filters the same cut-off frequency. Setting the cut-off frequency of one of the filters to a different value would compromise either the flatness in the pass-band, or the steepness in the stop-band. It can be shown[4] that the *square* of the transfer function (we are considering signal *power* rather than amplitude here) for a filter of order n has the form

$$|T_n|^2 = \frac{1}{1+\omega^{2n}} \tag{4.8}$$

The positions of the poles implied by this transfer function may now be determined exactly; the will occur where

$$s^{2n} = 1 \tag{4.9}$$

for a filter with order n. We can only use the poles which are situated on the left-hand half of the s-plane, as these correspond to stable systems. Attempts to place poles on the right-hand half of the plane results in instability with any input causing an exponentially rising output signal to be generated. The roots of Equation 4.9 lie on the unit circle *in the s-plane*, which gives a transfer function referred to as Butterworth's Polynomial, shown in Figure 4.10.

$$B_n = \begin{matrix}1\\ \text{or}\\ (s+1)\end{matrix} \prod_k s^2 + 2\cos\psi_k s + 1 \tag{4.10}$$

where ψ_k are determined by the following simple rules:

1. If n is odd, there is a pole at $\psi = 0$;
   if n is even, there are poles at $\pm 90°/n$
2. Poles are separated by $180°/n$

This yields the required Qs for each section as

$$Q = \frac{1}{2\cos\psi_k} \tag{4.11}$$

As the filter order increases (the designer demands sharper and sharper cut-offs), the poles move closer and closer to the imaginary axis where $\psi = \pm 90°$, and higher values of Q are requrired. Higher Qs palce demands on the filtering program by demanding greater precision arithmetic, so whether the filter is realised as a circuit based on electronic components or, as in our case, a computer program, there is always a limit to the roll-off which can be achieved. Fortunately, digital systems using floating point arithmetic are able to approach quite close to the ideal.

4.3.3 The Chebyshev Filter

The Butterworth filter demands that the pass-band response was maintained absolutely flat, and that the steepest possible cut-off be achieved without violating

[4] Taking the Taylor series for $(1+\omega^{2n})^{-\frac{1}{2}} = 1 - \frac{1}{2}\omega^{2n} + \frac{3}{8}\omega^{4n} - \cdots$, it follows directly that $\left.\frac{d^{2n}T}{d\omega^{2n}}\right|_{\omega=0} = -\frac{1}{2}$ and $\left.\frac{d^{k}T}{d\omega^{k}}\right|_{k} = 0$ for all other k, implying maximum flatness

this requirement. This immediately raises the question: "how much steeper a cut-off can be achieved if the requirement for an absolutely flat pass-band response is compromised?"

The family of filters which have the steepest possible roll-off for a given ripple in the pass-band is called Chebyshev filters. Starting from the transfer function of the Butterworth filter in equation 4.8, we proceed as follows. The wish to compromise the flatness of the pass-band by a known, controlled amount, so the term ω^{2n} is replaced by some function of ω which is to be decided, giving:

$$|T_n|^2 = \frac{1}{1 + F_n^2(\omega)} \tag{4.12}$$

Now we require to find the function F_n (for a filter of order n) which, when applied in equation 4.12 yields a response with a known pass-band ripple and as steep a cut-off as possible in the stop-band.

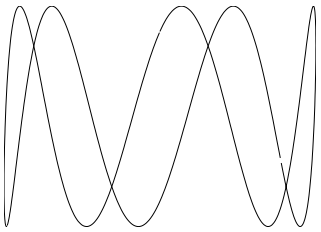


Figure 4.8: Lissajous Figure

In fact, Chebyshev himself had little to do with the design of electronic or digital filters. He did, however, uncover exactly the function for which we are searching whilst studying steam engines. If there were any natural justice, we should be calling this type of filter a Lissajous filter. Electronics enthusiasts, as well as those who enjoy watching 1970's science-fiction serials, will be familiar with the patterns obtained by connecting two sinewave oscillators to an oscilloscope. If the frequency the oscillator controlling the vertical movement of a dot is an exact multiple of the frequency controlling the dot's horizontal position, a *Lissajous Figure* is obtained. The one shown in Figure 4.8 is obtained when the horizontal frequency is one-fifth the vertical, and the oscillators are also out of phase by 45°.

Lissajous figures are closely related to the Chebyshev functions C_n. It is a property of the nth Chebyshev function that

$$C_n(\cos\phi) = \cos(n\phi) \tag{4.13}$$

In Chapter **??**, we shall return to this relationship, as it becomes very useful for applying controlled distortions to sound samples; pure sinusoids are simply multiplied in frequency by n. However, for the purposes of filtering, it is more important to think of it in the rearranged form:

$$C_n(\phi) = \cos(n\cos^{-1}\phi) \tag{4.14}$$

These functions C_n can be used to build filters with controlled passband ripple and steep cut off, by rewriting equation 4.12 thus:

$$|T_n(j\omega)|^2 = \frac{1}{1 + \epsilon^2 C_n^2(\omega)} \tag{4.15}$$

In the range $0 \leq \omega \leq 1$, this gives a constant ripple of magnitude determined by ϵ. Looking at Figure 4.8 once again, it is possible to see why this is the case. These functions have ripples which get closer together as the end of the range is approached, but the *size* of the ripple remains constant. It's quite easy to evaluate the magnitude of the frequency response of such a filter in the pass-band simply by substituting values from equation 4.14 into (4.15), but what happens when we

move outside the passband, where $\omega > 1$, is less clear. Under these circumstances it is requrired to evaluate $\cos^{-1}\omega, |\omega| > 1$. It turns out that C_n is indeed defined[5] under these circumstances, and that

$$C_n(\omega) = \cosh(n \cosh^{-1}\omega), \quad |\omega| > 1. \tag{4.16}$$

The nett result of this is that the frequency response of the system wobbles around while the input signal has a frequency in the passband, but disappears off "at right-angles to reality" as soon as $\omega > 1$, producing a very sharp cut-off indeed. Look back to Figure 4.7 (on page 42) to see how much the cut-off of a filter can be improved by admitting just a small amount of ripple in the passband.

Locating the Chebyshev Poles

The position of poles in the s-plane for a Chebyshev filter of given order can be obtained by the normal method of substituting s/j for ω in the transfer function (4.15). The result, after much tedious working, for the general case, is that the poles lie along an elipse in the s-plane, elongated in the direction of the imaginary axis. If the poles have coordinates (σ_k, ω_k), then for a filter of order n $(1 \leq k \leq n)$ it turns out that:

$$\begin{cases} \sigma_k &= \pm \sinh a \sin \frac{2k+1}{2n}\pi \\ \omega_k &= \cosh a \cos \frac{2k+1}{2n}\pi \end{cases} \tag{4.17}$$

where $a = \frac{\sinh^{-1} 1/\epsilon}{n}$.

In case the mathematics proves too onerous, there is a graphical method of locating these poles' postitions. This is illustrated in Figure 4.9, and the procedure is as follows:

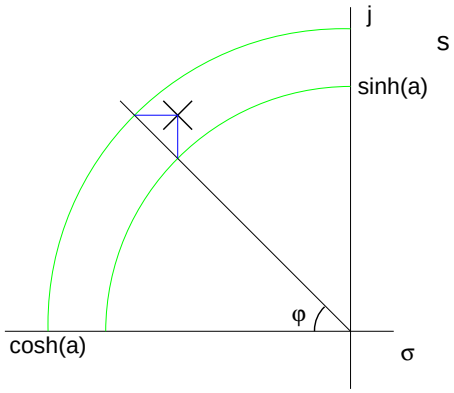


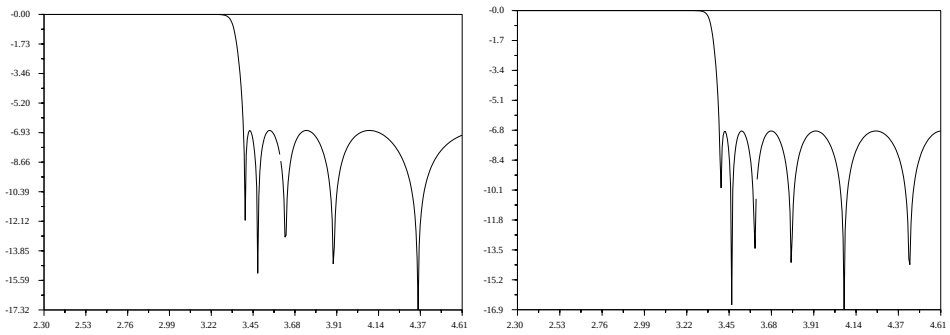
Figure 4.9: A Geometric Method to Locate Chebyshev Poles

[5]Suppose $\omega > 1$. Begin by evaluating z such that $\cos jz = \omega$, and proceed like this:

$$\begin{aligned} \cos jz &= \frac{1}{2}\left(e^{j^2 z} + e^{-j^2 z}\right) \\ &= \cosh z \\ &= \omega \end{aligned}$$

One can then calculate values for $\cos^{-1}\omega$ because

$$\begin{aligned} z &= \cosh^{-1}\omega \\ \cos^{-1}\omega &= j \cosh^{-1}\omega \end{aligned}$$



(a) Unwarped Response (b) Warped Response

Figure 4.10: Warping effect of the Bi-linear Transformation

1. Decide on the order of the filter, n, and the ripple ϵ which is tolerable in the passband.
2. Calculate $a = \frac{\sinh^{-1} 1/\epsilon}{n}$.
3. Construct two circles centred at the origin in the s-plane: the larger has a radius $\cosh a$; the smaller with radius $\sinh a$.
4. Draw radii at the angles appropriate for a *Butterworth* filter of the desired order.
5. Construct the right-angled triangles shown in the figure. The poles are still symmetrical about the horizontal axis, so the triangle is constructed upwards from the radius in the top half of the graph, and below the radius in the bottom half.
6. The required pole locations lie at the right-angled vertex of the resulting triangles.

4.4 Returning to the z-plane

Just as the initial oscillator "didn't quite work" when no account was taken of the fact that the system operated with descrete time-sampled data rather than continuously-valued "real-world" signals, it is also required to transform the solution obtained in the s-plane into a set of poles and zeros in the z-plane.

The exact solution to this problem is to replace all of the s which appear in the transfer response using the transformation $z = e^{-st}$. Unfortuately, this produces intractable solutions which are difficult to realise in a computer program. However, it is usually the case that only an *approximation* to the filter's frequency response is sought; the *exact* form of the response is usually fairly arbitrary anyway in a musical application, and is quite likely to be "tweaked" to produce an æsthetically acceptable effect rather than a technically precise one.

The apporximation which can be made is to replace all s which appear in the transfer function as follows:

$$s = 2\frac{1 - z^{-1}}{1 + z^{-1}} = 2\frac{z - 1}{z + 1}. \tag{4.18}$$

Equation 4.18 is called the *bi-linear transformation*, because it consists of the ratio of two functions which are linear in z. Such a simplification carries a penalty with it;

the *frequency axis* of the system's frequency-response graph becomes distorted. If the original system had a response $f(\omega)$, the new system produced by the bi-linear transformation has a response $f(\Omega)$ where

$$\Omega = 2\tan(\omega/2). \tag{4.19}$$

The final effect, as seen in Figure 4.10 is that the frequency response is "warped" at higher frequencies, but mapped with reasonable linearity at low frequencies. More accurately, the frequency axis has been warped by a factor $\omega = 2\arctan(\Omega/2)$.

4.5 Band-stop, Band-pass and High-pass filters

Designing filters other than low-pass ones is a matter of *transforming* the problem from a low-pass one by substituting functions of s into the continuous-domain solution, then using the bilinear transform to obtain the corresponding digital filter. The convertions in Table 4.1 can be thought of as recipies to make the desired type of filter from the prototype low-pass one. ω_p is the cutoff frequency of the analogue prototype filter, and $\hat{\omega}_c$ the new cut off frequency of the high-pass filter. $\hat{\omega}_{c1}$ and $\hat{\omega}_{c2}$ are the upper and lower transition frequencies of the band-pass and -stop filters. Usually, when commencing the design of one of these filters, the prototype lowpass can have its angular cutoff frequency set at 1 ($f_p = 1/2\pi$Hz) which causes all of the ω_p terms to disappear.

Type	**Transformation**
Low-pass (frequency scaling)	$s = \dfrac{\hat{s}}{\omega_p \hat{\omega}_c}$
High-pass	$s = \dfrac{\omega_p \hat{\omega}_c}{\hat{s}}$
Band-pass	$s = \omega_p \dfrac{\hat{s}^2 + \hat{\omega}_{c1}\hat{\omega}_{c2}}{\hat{s}(\hat{\omega}_{c2} - \hat{\omega}_{c1})}$
Band-stop	$s = \omega_p \dfrac{\hat{s}(\hat{\omega}_{c2} - \hat{\omega}_{c1})}{\hat{s}^2 + \hat{\omega}_{c1}\hat{\omega}_{c2}}$

Table 4.1: Transformations of s

4.6 Some Examples

Here are three examples. The first shows how the calculations in the preceding section can be used to create a simple low pass filter cutting off at 1kHz. In the second, a high-pass filter is constructed of the sort which was once (in the days of the long-playing gramophone record) popularly used to remove hum and motor noise from the audio stream. The cutoff frequency has been selected rather higher than a real rumble filter, so that its effect will be easily audible on the lower quality sound system available on most personal computers. The second example illustrates the use of a mathematical software package to determine the poles for a band-pass filter which will have application in telecommunications as well as in subtractive synthesis.

4.6.1 A Basic Low-pass Filter

Let's run through the design process for a basic 1kHz low-pass, second order, Butterworth filter and to demonstrate the design process. This design assumes a sample

rate of 44100Hz, but we normalise our sample rate to 1Hz to simplify the working, so adjusting the cutoff frequency appropriately gives:

$$\begin{aligned} \omega_c &= \frac{2\pi(1000)}{44100} \\ &= 0.1424759 \end{aligned} \tag{4.20}$$

Because we are about to use a bi-linear approximation to convert from the s-plane to z, we need to pre-warp the cut-off frequency so that after the distortion of the frequency axis this introduces, the cut-off will move back to 1kHz again. This yields:

$$\hat{\omega}_c = 2\tan(\omega_c/2) = 0.1427174$$

The poles of a second-order low-pass prototype Butterworth filter which cuts off at $\omega_p = 1$ lie on the unit circle in the s-plane at angles 135 and 225 degrees. That is to say $p_1, p_2 = (-\frac{\sqrt{2}}{2} \pm j\frac{\sqrt{2}}{2})$. We shall scale the response response of the filter by making the substitution $s = \frac{\hat{s}}{\hat{\omega}_c}$

$$H(s) = \frac{1}{(s-(-\sqrt{2}/2+j\sqrt{2}/2))(s-(-\sqrt{2}/2-j\sqrt{2}/2))} \tag{4.21}$$

$$= \frac{1}{1+\sqrt{2}s+s^2} \tag{4.22}$$

$$H(\hat{s}) = \frac{1}{1+9.9091908\hat{s}+49.096031\hat{s}^2} \tag{4.23}$$

Finally, using the bi-linear transform and substituting $2\frac{z-1}{z+1}$ for s gives

$$H(z) = \frac{1+2z+z^2}{177.56574-390.76825z+217.2025z^2}$$

Implementation is straightforward after dividing top and bottom by z^2 and removing a factor to make the coefficient of z^0 unity in both the numerator and denominator.

4.6.2 An Anti-rumble Filter

We will design a high-pass filter which cuts off at 500Hz using the same sample rate as before. This frequency is a decade higher than most rumble filters, but should have a clearly audible effect when auditioned even on small loudspeakers. A Butterworth design is used, as this offers a good compromise between a sharp transition between pass- and stop-bands with a tolerable amount of phase distortion. A sharper response than the filtering in the previous section will be desirable to provide good rejection in the stop band, so a fourth-order design is adopted.

As before:

$$\begin{aligned} \omega_c &= \frac{2\pi(500)}{44100} \\ \hat{\omega}_c &= 2\tan(\omega_c/2) = 0.0712681 \end{aligned} \tag{4.24}$$

The relevant roots for a fourth-order Butterworth filter lie on the unit circle $5\pi/8$, $7\pi/8$ and their conjugates, giving poles at $-0.9238795 \pm 0.3826834j$ and $-0.3826834 \pm 0.9238795j$. Multiplying out:

$$H(s) = \frac{1}{1+2.6131259s+3.4142136s^2+2.6131259s^3+s^4} \tag{4.25}$$

This can be transformed into a high-pass filter by replacing s with $\frac{\hat{\omega}_c}{s}$. Rearranging gives:

$$H(\hat{s}) = \frac{s^4}{0.0000258 + 0.0009459s + 0.0173413s^2 + 0.1862324s^3 + s^4} \tag{4.26}$$

Applying the bilinear transform, substituting $2\frac{z-1}{z+1}$ for $\hat{s}$ in (4.26) finally yields the required filter:

$$H(z) = \frac{0.9108492 - 3.6436708z + 5.4658955z^2 - 3.6441764z^3 + 0.9111025z^4}{0.8298769 - 3.4742263z^2 + 5.457982z^3 - 3.8136091z + z^4} \tag{4.27}$$

Once again, the expression must be adjusted to contain only negative powers of z and a gain extracted. Figure 4.11 shows the frequency response of the filter in Equation 4.27.

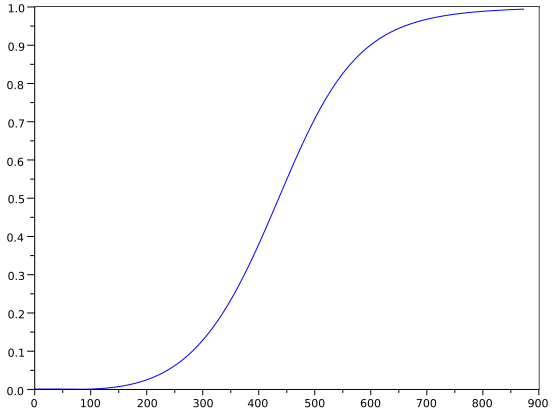


Figure 4.11: Frequency Response of Anti-Rumble Filter (detail)

4.6.3 Using Scilab

The free, open-source mathematical package SciLab is one of many with built-in functions for the calculation of filter parameters. The iir function, in particular, directly calculates $H(z)$ of a given order for low-, high-, and band-pass filters, in addition to band-reject filters, of a variety of filter families.

A help-page with example code is available on-line.

Appendix A

Tools for Writing Audio Programs

Because of the very special constraints placed upon audio applications, and in particular the heavy processing and low latency demands, most "power" applications tend to be written in a lower-level language such as C. As applications have become more and more sophisitcated and thus difficult to maintain, C's relatively primitive structure has caused it to give way primarily to object-oriented languages such as C++. Some applications use very different languages such as Lisp or Guile, but these tend to be used on top of DSP libraries written in more basic languages, to which they bring a richness and structure.

It is assumed that the reader is familiar with C or C++ at least to the extent that s/he can read example code written in it. The description of such a language in its entirety is beyond our scope, and many other texts are available which provide excellent introductions. However, many or most may be unfamiliar with the tools which are commonly used to support the construction of larger programs. In this section, we examine some of the tools available for the construction and maintenance of larger pieces of software, and survey some higher-level tools and packages which are useful in prototyping.

A.1 Using the C(++) compiler

Let us suppose that the programmer has written a piece of software which prints "hello world" on the terminal in C and wishes to compile it. If the software is contained in a single file, say, hello.c, a binary can be generated directly from the command line, thus:

```
gcc hello.c -o hi
```

We are instructing the C compiler to read the source file, produce a binary, link it, and make an executable called hello. Running the program is a simple matter of typing `./hi` at the prompt (the leading `./` is required because the current directory is not normally included in a program search path for reasons of security).

Unfortunately, it simply isn't feasible to expect any real program to be contained within a single file. Suppose we wish to prompt the user for a name, and then make a more personal greeting. We may want to break the program up into three distinct parts: a subroutine which asks for the user's name; a subroutine which issues the greeting; and a main program which glues the rest together in the right order. For easier maintenance we would split give each of the parts a separate file. The main program, in C, appears in Code Segment A.1.

Code Segment A.1 A Polite Greeting Program

```
#include <stdio.h>
#include "ask.h"
#include "greet.h"

int main(void)
{
    char *who;

    who = ask_name();
    greet_emit("Hello there, ", who);
}
```

We'll assume there are other files, ask.c and greet.c which provide the associated subroutines.

Each file can be compiled to object code separately and then linked together to make one program. We'd do that like this:

```
gcc -c main.c -o main.o
gcc -c ask.c -o ask.o
gcc -c greet.c -o greet.o
gcc -o hi main.o ask.o greet.o
```

Apart from increasing maintainability by keeping the files small, this approach can save a lot of time. For example, if only one file is changed, only that file needs to be recompiled. If main.c was edited, only the first and last of the gcc commands above needs to be issued to reconstruct the program called hi.

There are plenty of other options one can pass to gcc and its relations. `-Wall` makes the compiler very fussy and issue a warning when anything might possibly be wrong with the source code, even if there isn't an error as such. It is particularly useful during the development phase. `-g` causes the compiler to retain all the symbolic information after linkage, enabling effective debugging. After the software is believed to be working and is released, speed and size might be more important than the above considerations, so the flags might be removed and replaced with `-O2` (letter O) indicating that level-2 optimisation of the binary is required.

A.2 Using make

Any program can be compiled by issuing sufficent gcc commands, but as the program becomes larger and the number of source files increases, all that typing rapidly becomes at first tedious and then impossible. The audio synthesis application Quasimodo contains, at the time of writing, some 514 C++ source files, and would be virtually impossible to build with out some kind of automatic tool. Further, there is a danger that the correct files are not recompiled. Suppose in the above example, a bug is discovered. Changes are made to one source code file, and after a great deal of head-scratching, to another. But now the programmer forgets that the first file has changed, and compiles only the second. When the program is run, the bug might still be there, even though its cause has been detected and removed! This is where make can help: it will look at all the source files in a project, see which have source files which have been edited since the last compilation, and recompile them for you. It will also compile *all files depending on that file*, so if you change a header file used by many different C files, make will take care of that too.

All you need to do is provide a file for make to read. In the case of a simple program which keeps all its source code in a single directory, the instructions are fairly short, if a little cryptic. The Makefile for hi appears in Code Segment A.2.

Code Segment A.2 A Makefile for a Simple, Single-directory Program

```
1   PROGNAME := hi
    SRCS := main.c ask.c greet.c

    OBJS := $(SRCS:.c=.o)

    $(PROGNAME): $(OBJS)
            $(LINK.c) -o $@ $(OBJS)

    .PHONY: clean
10  clean:
            $(RM) \#*\# *.o *~ core .depend $(PROGNAME)

    .PHONY: depend
    depend:
            $(CPP) -M $(SRCS) > .depend

    ifeq (.depend,$(wildcard .depend))
    include .depend
    endif
```

To understand what's going on in this file, you need to know that make is based on a series of *rules* which tell it how some files are dependent upon the contents of others.

The first two lines in the Makefile are simple variable assignments. It is conventional, but not compulsory, to use upper case for variable names in make. PROGNAME is the name of the target program, and SRCS is the list of source files which need to be compiled and linked together to make it. The rest of the file is completely generic, and will work for any simple program which sits in one directory, so you can just hack the Makefile you used in the last project instead of writing one from scratch.

The third non-blank line is also an assignment, but this time it uses a *substitution expression* to set the variable OBJS to be a list of all the files named in SRCS, but with the ".c" on the end of the filenames replaced with ".o". We could just as easily have written `OBJS := main.o ask.o greet.o`, but then we would risk changing the list of source files and forgetting to update the list of object files to make it match.

Now follow the *rules*. The form of a rule is simple. The first line contains a *target* followed by a `:` and a list of *dependencies*, which is to say things which need to be up-to-date in order to build the target. So to build hi ($(PROGNAME)), all the object files need to be up-to-date. If they aren't, make will have to invoke one of its other rules to make them up-to-date. If they are, it can invoke the rule's *action*.

The action appears on the non-blank lines following the target. Each action line must start with a tab character (it is a common error to use spaces instead of a tab). The action for this rule is to *build a C program* ($(LINK.c) — make understands how to do this internally) sending the output (-o) to a file with the same name as the target ($@) by linking together all of the object files ($(OBJS)).

Having got the basic funcationality in place, it is good to provide some utility functions too. By default, make tries to build the first target it sees, but we have added a couple of *phony* targets. `make clean` will delete all of the files left over

after compilation (including the program itself). This is very useful if you want to deliver a "clean" copy of the source code. The recipient might wish to recompile the program on a different computer, for example, and having old object files and binaries around is bound to cause problems. The target is a phony one because, whereas `make hi` (or `make` by itself) would result in the production of a file called hi, `make clean` doesn't produce a file called "clean".

We've almost finished now, but while the relationship between the program and its object files is explicit, we've not yet involved the dependencies of header files, or of the object files on their source files. Even in our trivial example, we might edit ask.h, for example, and would then want to recompile all the files which included it. And header files may depend on other header files. Rather than maintaining the dependency list by hand, which would be almost as bad as not using make at all, we can ask the C pre-processor to scan all of the source files and print out the dependency list. `make depend` does this for us: it runs cpp with the "-M" flag, and saves the results in a file called .depend. If this file exists, it's included at the end of the Makefile as if it had been typed there. cpp -M trusts nobody, so it even scans through the system header files to make sure things will be recompiled if they change (only the system administrator can change a system header file). If you look in .depend, you'd expect to see the names of some files you didn't write in there.

To elucidate, let's build the program. First we will construct the dependency list, then look at it, then build and run the "hi" program. Finally, we'll clean up. At each stage, make tells us what it's doing.

```
Greet-C$ ls -a
.  ..  Makefile  ask.c  ask.h  greet.c  greet.h  main.c
Greet-C$ make depend
cc -E -M main.c ask.c greet.c > .depend
Greet-C$ cat .depend
main.o: main.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
 /usr/lib/gcc-lib/powerpc-linux/2.95.2/include/stddef.h \
 /usr/lib/gcc-lib/powerpc-linux/2.95.2/include/stdarg.h \
 /usr/lib/gcc-lib/powerpc-linux/2.95.2/include/va-ppc.h \
 /usr/include/bits/types.h /usr/include/libio.h \
 /usr/include/_G_config.h /usr/include/bits/stdio_lim.h ask.h greet.h
ask.o: ask.c  /usr/include/string.h /usr/include/features.h \
 /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
 /usr/lib/gcc-lib/powerpc-linux/2.95.2/include/stddef.h ask.h
greet.o: greet.c greet.h
Greet-C$ make
cc    -c -o main.o main.c
cc    -c -o ask.o ask.c
cc    -c -o greet.o greet.c
cc     -o hi main.o ask.o greet.o
Greet-C$ ls -a
.   .depend   ask.c  ask.o    greet.h  hi      main.o
..  Makefile  ask.h  greet.c  greet.o  main.c
Greet-C$ ./hi
What's your name?
Nick
Hello there, Nick
Greet-C$ make clean
rm -f \#*\# *.o *~ core .depend hi
Greet-C$ ls -a
.  ..  Makefile  ask.c  ask.h  greet.c  greet.h  main.c
```

A.3 Using the Debugger

Building and running a program doesn't always go that smoothly. Let's suppose we made a pig's ear of writing the ask_name function in the file ask.c. Code Segment A.3 shows the incorrect version.

Code Segment A.3 A Programming Error causes a Segmentation Fault

```
#include <string.h>
#include "ask.h"

char *ask_name(void)
{
  char instr[64];
  char *result;

  printf("What's your name?\n");
  scanf("%63s", instr);

  result = (char *)malloc(strlen(instr) + 1);

  strcpy(result, instr);
  return *result;
}
```

Building and running this causes a crash (although it's obvious why, we'd crave your indulgence for the sake of this example).

```
nick@cmt1:~/Coursenotes/Audio-Programming/Greet-C$ make
cc -g   -c -o main.o main.c
cc -g   -c -o ask.o ask.c
ask.c: In function ‘ask_name’:
ask.c:14: warning: return makes pointer from integer without a cast
cc -g   -c -o greet.o greet.c
cc -g    -o hi main.o ask.o greet.o
nick@cmt1:~/Coursenotes/Audio-Programming/Greet-C$ ./hi
What's your name?
Nick
Segmentation fault
```

We'd like to run the debugger on this to see what's going on, so it will be necessary to modify the Makefile slightly. We need every invocation of the C compiler to include -g to retain all of the debugging information. Fortuantely, make uses a built-in variable CFLAGS, so it is simply a matter of adding this flag to that variable. The line

```
CFLAGS += -g
```

is added before the rules. It will then be necessary to rebuild the entire program and run the debugger against it. It is in fact possible to run gdb, the GNU Debugger, inside the emacs editor, but we prefer to elegance of ddd, the Data Display Debugger front-end by Dorothea Lütkehaus and Andreas Zeller.

```
nick@cmt1:~/Coursenotes/Audio-Programming/Greet-C$
    make clean && make depend
rm -f \#*\# *.o *~ core .depend hi
cc -E -M main.c ask.c greet.c > .depend
```

```
nick@cmt1:~/Coursenotes/Audio-Programming/Greet-C$ make
cc -g   -c -o main.o main.c
cc -g   -c -o ask.o ask.c
ask.c: In function ‘ask_name’:
ask.c:14: warning: return makes pointer from integer without a cast
cc -g   -c -o greet.o greet.c
cc -g    -o hi main.o ask.o greet.o
nick@cmt1:~/Coursenotes/Audio-Programming/Greet-C$ ddd hi
```

When the window appears, click on the Run button. Text input and output is in the text pane at the bottom of the application, so type your name in there. You'll see that a segmentation fault occurred inside a call to the strlen function. Since we can assume strlen is pretty robust as it is used quite a bit, it was probably a bad argument that caused the crash.

Click the Up button on the remote control to see how we got there. Clicking up a few time tells us that greet_emit begat printf, and printf begat vprintf, and vprintf begat strlen wherein the problem surfaced. Looking at my implementation of greet_emit in the source window and double-clicking on the arguments causes them to be displayed as shown in the screenshot. Tracing a further level up the stack shows

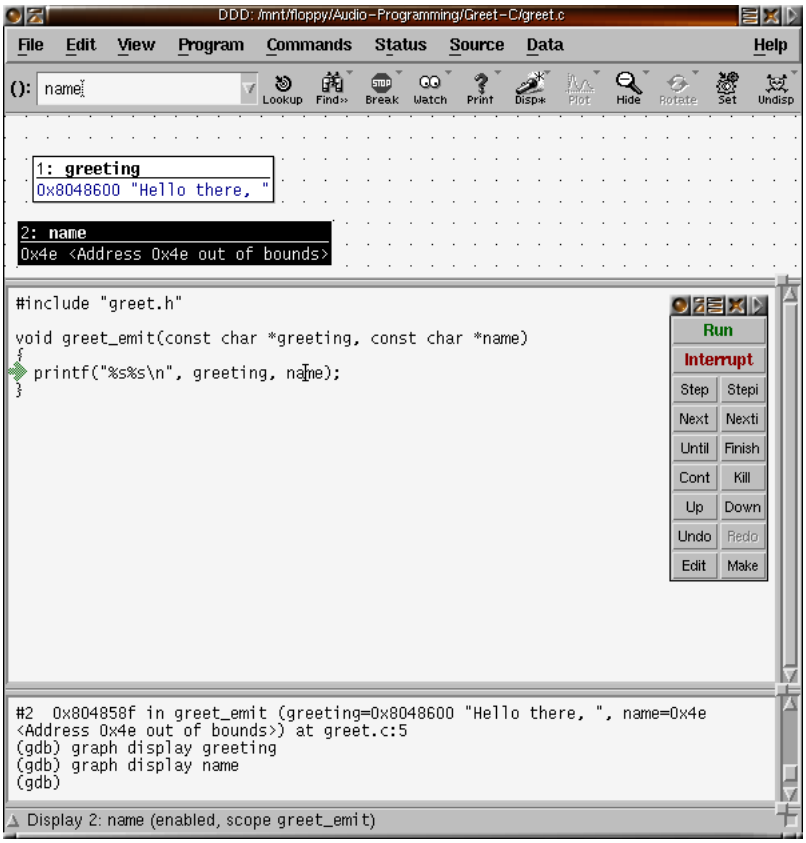


Figure A.1: Screenshot of a Debugging Session with ddd

that this second argument was the return value of the name-reading function... can you see what's wrong yet? Perhaps take another look at the compiler warning?

ddd supports, among many other things, the setting of breakpoints, the display of complex structures like linked lists and trees in a highly intuitive fashion, and many different source languages; it really is a joy to use. We recommend you make

a habit of using it as soon as a problem arises rather than messing around for hours putting in printf statements and recompiling.

A.4 Compiling Libraries and Mixing Languages

Having developed the asking and greeting software and throrougly tested it, we may wish to make a library from it. This has the advantage that any future program which may need to be written having the reqirement of politeness can use this well-tested code without having to compile the object files and link them to the final executable. One can think of a library as a "semi-linked" program: all of the hard work which needs to be done to link together a group of object files into a larger entity has been done already, and the whole lump can be glued onto a new application in one go.

In some operating systems, there is a further advantage: libraries can be shared between applicatons. For example, the C run-time library, containing the functions described in stdio.h, stdlib.h etc., is linked to executables automatically. Since the majority of applications running on a computer system will be using this library, it makes sense to load the code for it only once instead of wasting time and memory by having each program load its own copy.

The normal sort of library, which is just a sort of lump of objects, is called a *static* library, and usually has the file extension `.a` standing for *archive*. The more sophisticated library which is loaded on demand at run-time, is called a *shared* library and usually has the extension `.so` meaning *shared object*.

A.4.1 Building the Static Library

In the library we are going to make available the ask_name() and greet_emit() functions. These are in the object files ask.o and greet.o so two new variables are added to the top of the make file: these contain the names of the sources and objects which are to form the library, libpolite, as opposed to the main program. A new rule is added with an alias `staticlib` (this way, we can type `make staticlib` to check out that the library is compiling correctly). The whole Makefile no comes to Code Segment A.4:

Making a shared library is more difficult for the computer but not much more difficult for the programmer. This is because making a shared library consitutes a "partial linking". Whereas the static library is really just a ball of objects in one file, the shared library, which is linked at run-time has other issues which must be taken into consideration. What if two different programs choose to load the library at different virtual addresses? Where is the library loaded from? What happens if one shared library makes references to another (which might not be loaded yet)?

Fortunately, from the programmer's point of view, things are not really much more complicated. The commands to generate a shared library can be added to the Makefile quite simply:

```
libpolite.so: $(LIBOBJS)
        $(LD) -shared -o $@ $^

.PHONY: sharedlib
sharedlib: libpolite.so
```

With the main program dependency changed from `staticlib` to `sharedlib`, a dynamically linked version of the program can be built just as easily.

Some caveats:

Code Segment A.4 A Makefile to build a Program and its Static Library

```
PROGNAME := hi
SRCS := main.c
LIBSRCS := ask.c greet.c
CFLAGS += -g

OBJS := $(SRCS:.c=.o)
LIBOBJS := $(LIBSRCS:.c=.o)
LDFLAGS += -L .

$(PROGNAME): $(OBJS) staticlib
        $(LINK.c) -o $@ $(OBJS) -lpolite

libpolite.a: $(LIBOBJS)
        $(AR) -rv $@ $^

.PHONY: staticlib
staticlib: libpolite.a

.PHONY: clean
clean:
        $(RM) \#*\# *.o *~ core .depend libpolite.* $(PROGNAME)

.PHONY: depend
depend:
        $(CPP) -M $(SRCS) $(LIBSRCS) > .depend

ifeq (.depend,$(wildcard .depend))
include .depend
endif
```

1. Building and running the program results in an error about being unable to locate the shared library:

   ```
   Greet-C$ make
   cc -g   -c -o main.o main.c
   cc -g   -c -o ask.o ask.c
   cc -g   -c -o greet.o greet.c
   ld -shared -o libpolite.so ask.o greet.o
   cc -g  -L .  -o hi main.o -lpolite
   Greet-C$ ./hi
   ./hi: error in loading shared libraries:
      libpolite.so: cannot open shared object file:
                    No such file or directory
   ```

   This is because the program is expecting to find the library in the trusted system directories, where only the administrator has write permission. Since you can't put the library there, you will have instead to set an environment variable: `export LD_LIBRARY_PATH=.` so that the library can be found.

2. Changing from a static to shared library in the Makefile can mess up the dependencies. If you forget to delete the static library when you build a dynamic one, there is the possibility that the wrong one will be linked. This is compounded by the default behaviour of the compiler which is that on seeing the -l flag, it will prefer to link a shared library rather than a static one *regardless of which is newer*. So after editing the Makefile, do `make clean && make depend`

— this is good practice whenever the makefile is edited, other than the simple addition of extra source code files.

3. Usually, you will want to have a separate directory for each of the libraries you wish to build. A top-level Makefile will change to each directory in turn and execute make inside it. Using this technique, very large applications can be built using many different libraries. Header files are often placed in a different directory from the source code, so they are still available to provide library function prototypes after the libraries have been built and installed.

A.4.2 Mixing Source Languages

Using all of the tools we've come across so far, it is possible to mix source languages and thus reuse legacy code. This can become very important in a large system: many thousands of man-hours might have been invested in a C program and its libraries, but a new system is being written in C++. Let's link libpolite with the new and improved version of our greeting program, hipp ("hi-plus-plus").

hipp uses state-of-the-art object-oriented technology. It defines a *class* `butler` which responds to the user. When a butler is created, it is trained in what to say. The files wrapper.cc and wrapper.h, which add this functionality to the library we have already, are shown in Code Segments A.5 and A.6 respectively.

Code Segment A.5 A C++ Wrapper for the Polite Greeting Library

```
// File wrapper.cc
// C++ wrapper for libpolite

#include <iostream>
#include <stdlib.h>
#include "wrapper.h"
using namespace std;

extern "C" char *ask_name(void);
extern "C" void greet_emit(char *, char*);

butler::butler(char *g) {
  greeting = g;
  name = ask_name();
}

butler::~butler() {
  free(name);
  cout << "All cleaned up\n";
}
```

We also need to change the Makefile to accommodate the extra program hipp, which can be done by adding the following variables and rules in the appropriate places:

```
CXXPROGNAME := hipp
CXXSRCS := wrapper.cc hipp.cc
CXXFLAGS += -g

CXXOBJS := $(CXXSRCS:.cc=.o)
```

Code Segment A.6 A C++ Header File for the Polite Greeting Library

```
// wrapper.h
// Header file for C++ library wrapper

// Include file only once
#ifndef WRAPPER_H
#define WRAPPER_H

#include <iostream>
using namespace std;

class butler {
 private:
  char *greeting, *name;

 public:
  butler(char *); // Constructor
  ~butler();      // Destructor
  void greet(void) { cout << greeting << name << '\n'; }
};

#endif
```

```
$(CXXPROGNAME): $(CXXOBJS) sharedlib
        $(LINK.cc) -o $@ $(CXXOBJS) -lpolite

.PHONY: cxxdepend
cxxdepend:
        $(CPP) -M $(CXXSRCS) $(LIBSRCS) > .depend
```

While we're at it, add `$(CXXPROGNAME)` to the list of things removed my `make clean`. Lastly, the new main program needs to be constructed, and that happens in file hipp.cc (Code Segment A.7)

Code Segment A.7 Main C++ Program for an Application using the Polite Library

```
// Main file hipp.cc for the hipp program

#include "wrapper.h"
using namespace std;

int main(void)
{
  butler b("You rang, Mr ");

  b.greet();
  return 0;
}
```

Building is quite easy with the new make script:

```
Greet-C$ make clean
```

```
rm -f \#*\# *.o *~ core .depend libpolite.* hi hipp
Greet-C$ make cxxdepend
cc -E -M wrapper.cc hipp.cc ask.c greet.c > .depend
Greet-C$ make hipp
g++ -g   -c -o wrapper.o wrapper.cc
g++ -g   -c -o hipp.o hipp.cc
cc -g   -c -o ask.o ask.c
cc -g   -c -o greet.o greet.c
ld -shared -o libpolite.so ask.o greet.o
g++ -g  -L .  -o hipp wrapper.o hipp.o -lpolite
Greet-C$ ./hipp
What's your name?
Nick
You rang, Mr Nick
All cleaned up
```

Some comments:

1. Using C++, it is easier (though still quite hard!) to keep track of memory allocation. Notice that, even though we didn't *explictly* ask for it, the memory allocated to store the user's name was freed up when the butler was destroyed at the end of its useful life (subroutine main). By writing a destructor, we can make it harder (though still not hard enough!) to have programs with memory leaks.

2. The Makefile is now quite out of hand, with far too many targets. To have the library and a main program in the same directory one might consider unlucky; to have a library and *two* main programs in the same directory is carelessness.