EIN KLEINER FILTER COMPILER

Kenneth Steiglitz

CS-TR-279-90

August 1990

Ein Kleiner Filter Compiler

Kenneth Steiglitz
Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

A "little" compiler (actually, a C pre-processor) is described that translates an Intermediate Filter Language (IFL) into filter code. The main motivation is to provide an easy way to experiment with filters for digital sound synthesis. The compiler itself comprises 60 lines of C and produces C. The new keyword **tap** is available, which creates a buffer for storing the current value of the signal, which is then available earlier or later in the computation as a feedback or feed-forward term, respectively. C code in the filter description is passed largely unchanged, except for variables of the form S*i*, which is the signal value from the *i*-th buffer. In this way, the usual FIR and IIR filters can be represented easily, as well as much more general filters and signal generators. The compiler is portable, requires small computer resources, uses text input that can be generated by other programs, and is easy to modify.

1. Introduction

A central problem in computer music is the control of timbre. Extensive trial and error is often necessary to tailor sound color. This paper describes a very simple way to carry out that experimentation when sound is generated with digital filters. The basic approach is to describe a digital filter with text in an intermediate language (IFL) and then compile that into code which implements the filter itself.

The filter compiler is not meant to be competitive with the elaborate and general implementation systems developed for simulating digital systems. (For a description of a recent and impressive system, and a literature review, see [LH89].) Rather it is intended to show that a very modest program, comprising about 60 lines of C, can be useful for this particular application. The compiler uses very small resources, requires no graphics support, and very little memory and cpu time. The filter is described by a text file using the intermediate language IFL. Besides simplicity and portability, the approach has two advantages over graphics input: — first, it means we can write other programs to generate IFL for classes of interesting filters; second, it means that IFL descriptions of filters can be easily combined and edited.

2. Preliminaries

The compiler (*ein*) reads the filter description from standard input, and produces code for the function *nexty* as standard output. This must then be compiled and run with the fixed buffered output program *main*, which just repeatedly calls *nexty*. Thus, a command sequence in UNIX for producing sound looks like:

```
ein < $1 > nexty.c
cc –o filt main.o nexty.c
filt | scale > T
cat HEAD T > TT
sndplay TT
```

Here *ein* is the filter compiler, *scale* is a buffered scaling program that scales the output magnitude to the range of the converter, and HEAD is the header required for sound files on the NeXT machine. The sources *ein.c*, *main.c*, and *scale.c* are listed at the end of this paper.

3. The filter language

The easiest way to explain the way a filter is specified to the compiler is by an example: the simple plucked-string filter [JS83, KS83] shown in Fig. 1.

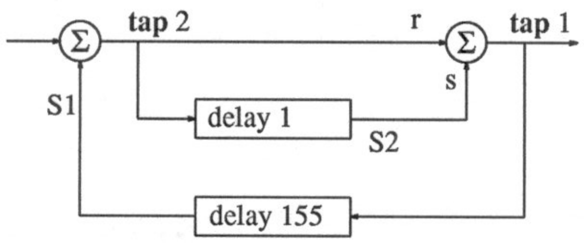

Fig. 1 Signal-flow graph for a plucked-string filter. Two **tap**'s are required, one for each loop.

The Intermediate Filter Language code to describe this filter, with a unit impulse as input, is:

```
#define R 0.99
if ( t == 0 ) { y = 1; } else { y = 0; }
y = y + R*S1;
tap 2 1
y = 0.5*y + 0.5*S2;
tap 1 155
```

First, there are always two pre-defined variables: *y* for the signal at any point in the flow graph at which we happen to be computing (**double**); and *t*, the global time in sample instants (**int**).

A line beginning with the command **tap** *i k* has two effects. First, it causes the buffer *Bi* to be created, which stores the value of the signal at that point. Second, the variable *Si* is recognized anywhere else in the code as the contents of that buffer, which is the signal at that point delayed by *k* sampling intervals. Except for the translation of *Si* variables, lines not beginning with the keyword **tap** are transparent to the compiler.

If a **tap** appears before the first use of its signal, the signal is called *feed-forward*; otherwise, *feedback*.

We can now interpret the IFL above. We proceed from left to right in the signal flow graph, writing a line for each signal operation. The first line defines the gain constant *R*. The second line produces an input which is a unit impulse at $t = 0$. Next, the line $y = y + R*S1;$ corresponds to the closing of the feedback loop, using signal S1. The command **tap 2 1** creates the feed-forward loop for signal 2, which is combined with the signal y in the next line, to form the simple FIR filter with transfer function $0.5 + 0.5z^{-1}$. Finally, **tap 1 155** creates the feedback loop for signal S1 with delay 155.

4. Some details of the compiler

Because of the simplicity of the specification language IFL, the compiler itself is no more difficult than a Freshman programming assignment. In a single left-to-right pass of the IFL input file, each non-**tap** line is simply passed to the output, using the appropriate values from buffers for the *Si*. The only slightly delicate issue is the way the circular buffers for storing delayed signals are handled.

When a **tap** i l line is encountered, a new buffer *Bi* of length N is declared to be an array, together with the **int** index into that array, *IDi*. Then code is generated that stores the signal at that point in the buffer. Thus, the last line in the IFL code above generates the code

```
B1[ID1++] = y;
ID1 = ID1%155;
```

The index $ID1$ is incremented after it is used, and then taken modulo the length of the buffer.

The only tricky point concerns the definition of the length N of the buffer. Suppose L is the desired loop delay, the parameter in the IFL code. When the loop is a feedback loop, $N = L$; when the loop is a feed-forward loop, $N = L + 1$. It is then not hard to verify that the buffer contents $Bi[IDi]$. provides a signal with the desired delay. The extra buffer storage location in the case of a feed-forward loop is necessary because in that case the **tap** command is encountered during the same ''clock'' cycle that the signal is used.

It also follows from this arrangement that the case $L = 0$ is allowed for a feed-forward loop, corresponding to a buffer of length 1; the present signal value is simply saved for use during the same clock cycle of the filter. This is critical for the generality of the specification language, which we discuss next. A signal may be used both before and after its corresponding **tap**, but the user must remember that because the signal is used before its **tap**, the length of its buffer is that of a feedback signal, and uses after its tap are delayed one fewer sampling interval than the designated delay. As an example, if signal $S1$ is used both before and after **tap** 1 1, uses before the tap are delayed 1 interval, but uses after are delayed 0 intervals.

The complete source for the function *nexty.c* for the plucked-string example is shown below; the file *decl.h* contains the declarations and is also generated by the compiler.

```
#include <math.h>

double nexty(t)
int t;
{ double y;
#include "decl.h"

#define R 0.99
if ( t == 0 ) { y = 1; } else { y = 0; }
y = y + R*B1[ID1];
B2[ID2++] = y;
ID2 = ID2%2;
y = 0.5*y + 0.5*B2[ID2];
B1[ID1++] = y;
ID1 = ID1%155;
return y;
}
```

5. Generality

Any signal-flow graph G that represents a realizable, linear, constant-coefficient digital filter can be represented by IFL. To see this, remove the positive delays from G to obtain G', and topologically sort the nodes; that is, order the nodes so that all arcs go from left to right. This is possible because G is realizable, which implies that G' is acyclic. The order of the nodes determines the order of the primitives in IFL. The arcs in G' can be implemented with feed-forward (left-to-right) arcs with delay zero and the remaining arcs with feed-forward arcs with delays $L \geq 1$ and feedback arcs with delays $L \geq 1$.

6. Root Locus for the Plucked-String Filter

The plucked-string algorithm of Karplus and Strong [KS83] is a remarkably effective and efficient way to generate complex sound, using only a fixed number of arithmetic operations per sample. In this section we look a little more closely at the plucked-string algorithm as a digital filter, and show how root locus can be used to understand such filters. We can view the plucked-string algorithm as a digital filter consisting of a single zero on the negative real axis, around which we place a feedback loop with a delay (see Fig. 1). The transfer function is thus:

$$H(z) = \frac{r + sz^{-1}}{1 - rz^{-p} - sz^{-p-1}} \tag{1}$$

which corresponds to a loop delay of p samples and an FIR filter in the forward path with transfer function $r + sz^{-1}$. When this filter is excited with a burst of random noise, the resulting sound is perceived as being very close to that of a plucked string. To see get some insight into why, it is useful to examine the pole locations. Approximate analyses are given in [JS83, KS83], but the root locus method provides an effective way to visualize the pole distribution.

It is convenient to replace the parameters r and s with two others that are each more closely related to one aspect of the filter's behavior. We use

$$G = r + s \ ,$$

the zero-frequency gain of the feedback loop; and

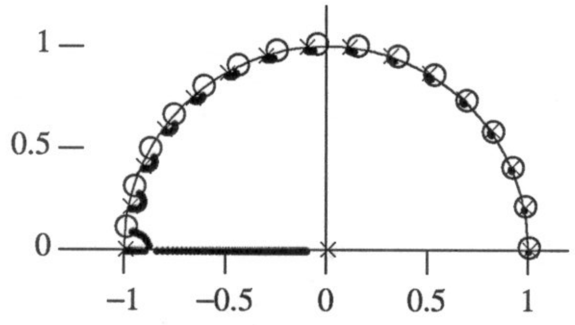

Fig. 2 Root locus of poles of a plucked-string filter as the real zero moves from $z = -2$ to $z = -0.1$, for fixed zero-frequency gain $G = 0.99$.

$$u = -s/r \ ,$$

the location of the real zero. The characteristic equation then becomes

$$z(z^p - G) - u(z^{p+1} - G) = 0$$

The fact that this is linear in u for fixed G allows us to interpret this as a usual root-locus, with ''poles'' at $z = 0$ and the p-th roots of G; and ''zeros'' at the $(p+1)$-st roots of G. Thus, as u moves from $z = 0$ to $z = -\infty$, the poles of the filter transfer function move from the p-th roots of G (and $z = 0$) to the $(p+1)$-st roots of G. This shows clearly why (when $G \geq 0$) the poles shift down in frequency — and more so at high frequencies than low, so that the resonant frequencies are inharmonic. As an example, Fig. 2 shows the locus of pole positions in a filter with $p = 30$ and $G = 0.99$ as the position of the zero is varied from $z = 0$ to $z = -2$. The plot shows also that the higher frequency poles are pulled inside the circle more than the lower frequency ones, and explains simply why the higher harmonics decay faster than the lower (again, when $G \geq 0$).

7. Examples

We conclude with some examples implementing some common sound synthesis algorithms.

Plucked String with Vibrato

Adding an allpass section in the loop of the plucked-string filter allows us to tune the pitch because it introduces phase shift [JS83]. The IFL below determines a plucked-string filter with a time-varying allpass section in the forward loop.

```
#define R 0.99
#define T (((double)t) /22050.0)
#define a(t) 1.0*( sin(8*M_PI*T*T))

if ( t == 0 ) { y = 1; } else { y = 0; }
y = y + R*S1;
tap 2 1
y = 0.5*y + 0.5*S2;
tap 3 1
y = a(t)*(S4–y) + S3;
tap 4 1
tap 1 255
```

The difference equation defining the allpass section is

$$y[t] \quad = \quad a(t)(\ y[t-1] \ - \ x[t]) \quad + \quad x[t-1]$$

where

$$a(t) \quad = \quad \sin(8\pi(t/sr)^2)$$

and *sr* is the sampling rate, in this case 22050 samples/sec. This varies the pitch at an increasing frequency as time progresses.

Brasslike FM Instrument

To illustrate the transparency of C code, a simple FM instrument, approximating a brasslike timbre, is shown next. The parameters are taken from [M85].

```
#define f0 300.0
#define sr 22050.0
#define T ((double)t)

double fc, fm;
int t1, t2, t3;
double a, i;
double omegac, omegam;

if ( t == 0 )
 { fc = f0;
   fm = f0;
   omegac = (fc /sr)*2*M_PI; /* from Hz to rad per sample */
   omegam = (fm /sr)*2*M_PI;
   t1 = (int)(0.05*sr);       /* from msec to sample number */
   t2 = (int)(0.3*sr);
   t3 = (int)(0.4*sr); }

if (t <= t1) {a = (T /t1);}
 else if (t <= t2)
  {a = 1;}
   else if (t <= t3)
    {a = (t3–T) /(t3–t2);}
     else { a = 0;}
if (t <= t1) {i = 1.0 + 3.5*(T /t1);}
 else {i = 4.5;}

y = a*sin(omegac*T + i*sin(omegam*T));
```

Reverberation Filter

Finally, the IFL below shows the preceding FM instrument followed by the reverberation filter described in [M90], consisting of 6 lowpass comb sections feeding an allpass section.

```
#define f0 300.0
#define sr 22050.0
#define T ((double)t)

double fc, fm;
int t1, t2, t3;
double a, i;
double omegac, omegam;
double ysave;
double y1, y2, y3, y4, y5, y6;

if ( t == 0 )
 { fc = f0;
   fm = f0;
   omegac = (fc /sr)*2*M_PI;  /* from Hz to rad per sample */
   omegam = (fm /sr)*2*M_PI;
   t1 = (int)(0.05*sr);        /* from msec to sample number */
   t2 = (int)(0.3*sr);
   t3 = (int)(0.4*sr); }

if (t <= t1) {a = (T /t1);}
 else if (t <= t2)
  {a = 1;}
   else if (t <= t3)
    {a = (t3-T) /(t3-t2);}
     else { a = 0;}
if (t <= t1) {i = 1.0 + 3.5*(T /t1);}
 else {i = 4.5;}

ysave = a*sin(omegac*T + i*sin(omegam*T));

y = ysave;
tap 1 1103
tap 2 1104
y = S1 - 0.24*(S2 - S3) + 0.6308*S4;
tap 3 1
tap 4 1103
y1 = y;

y = ysave;
tap 5 1235
tap 6 1236
y = S5 - 0.26*(S6 - S7) + 0.6142*S8;
tap 7 1
tap 8 1235
y2 = y;


y = ysave;
tap 9 1345
tap 10 1346
y = S9 - 0.28*(S10 - S11) + 0.5976*S12;
tap 11 1
tap 12 1345
y3 = y;

y = ysave;
tap 13 1499
tap 14 1500
y = S13 - 0.29*(S14 - S15) + 0.5893*S16;
tap 15 1
tap 16 1499
y4 = y;

y = ysave;
tap 17 1499
tap 18 1500
y = S17 - 0.30*(S18 - S19) + 0.5810*S20;
tap 19 1
tap 20 1499
y5 = y;

y = ysave;
tap 21 1499
tap 22 1500
y = S21 - 0.32*(S22 - S23) + 0.5644*S24;
tap 23 1
tap 24 1499
y6 = y;

y = y1 + y2 + y3 + y4 + y5 + y6;

tap 25 132
y = 0.7*(S26 - y) + S25;
tap 26 132

y = 0.9*ysave + 0.1*y;
```

The lowpass comb filters are implemented directly from the transfer function

$$H(z) \quad = \quad \frac{z^{-D} \; - \; g_1 z^{-D-1}}{1 \; - \; g_1 z^{-1} \; - \; g_2 z^{-D}}$$

8. Conclusions

We have described a very simple, ‘‘little’’ compiler that makes it easy to implement a wide variety of filters for digital sound synthesis.

Acknowledgements

I am indebted to Paul Lansky for all kinds of help. An earlier, and awkward, implementation of the filter compiler, in Pascal, is described in [S90]. This work was supported in part by NSF Grant MIP-8912100, and U. S. Army Research Office - Durham Contract DAAL03-89-K-0074.

References

[JS83] D. A. Jaffe and J. O. Smith, ‘‘Extensions of the Karplus-Strong Plucked-String Algorithm,’’ *Computer Music J.*, vol. 7, pp. 56-69, 1983.

[KS83] K. Karplus and A. Strong, ‘‘Digital Synthesis of Plucked-String and Drum Timbres,’’ *Computer Music J.*, vol. 7, pp. 43-55, 1983.

[LH89] E. A. Lee, W-H. Ho, E. E. Goei, J. C. Bier and S. Bhattacharyya, ''Gabriel: A Design Environment for DSP,'' *IEEE Trans. on ASSP*, vol. 37, pp. 1751-1762, Nov. 1989.

[S90] K. Steiglitz, ''A Filter Compiler for Digital Sound Synthesis,'' *1990 Int. Conf. Acoustics, Speech, and Signal Processing,* Albuquerque, NM, April 3-6, 1990.

[M90] F. R. Moore, *Elements of Computer Music,* Prentice-Hall, 1990.

[M85] J. A. Moorer, ''Signal Processing Aspects of Computer Music: A Survey,'' in *Digital Audio Signal Processing: An Anthology,* J. Strawn (ed.), W. Kaufmann, 1985.

EIN.C

```
#include <stdio.h>
#include <ctype.h>
#include <strings.h>

#define MAXLINE 250  /* max. line length */
#define MAXLIST 128  /* max. number of buffers */
#define TAP "tap"    /* keyword to create buffer for signal storage */

FILE *fopen(), *fdecl;  /* file decl.h, declarations for nexty*/

char line[MAXLINE];
int useds[MAXLIST]; /* whether signal has been used */

main()                                                              main
{ extern char line[];

  printf("#include <math.h>\n\n");
  printf("double nexty(t)\n");          /* beginning of function nexty */
  printf("int t;\n");
  printf("{ double y;\n");
  printf("#include \"decl.h\"\n\n");  /* definitions for nexty() */
  fdecl = fopen("decl.h", "w");
  while ( fgets(line, MAXLINE, stdin) != NULL )
   { if( strncmp(TAP, line, 3) == 0 ) creatbuf();
     else replace();}
  printf("return y;\n}");          }

getint(s, q) /* get next integer in line buffer s after position *q */   getint
char s[];    /* on exit, *q is position after last digit in integer */
int *q;
{ int t;
  while (!isdigit(s[*q])) (*q)++;
  for ( t=0; isdigit(s[*q]); t = 10*t + s[(*q)++] - '0');
  return t;                                                }

creatbuf()                                                          creatbuf
{ extern char line[];
  extern int useds[];
  int p; /* pointer to beginning of line */
  int id, delay; /* signal id, delay of buffer */

  p = 0;
  id = getint(line, &p);
  delay = getint(line, &p);
  if ( !useds[id] ) delay++;
  printf("B%d[ID%d++] = y;\n", id, id);
  printf("ID%d = ID%d%%%d;\n", id, id, delay);
  fprintf(fdecl, " static int ID%d;\n", id); /* put declarations into decl.h */
  fprintf(fdecl, " static double B%d[%d];\n", id, delay);          }

replace()                                                           replace
{ extern char line[];
  extern int useds[];
  int i, id;

  for ( i=0; line[i] != '\0'; )
   if(line[i] == 'S')
    { useds[id = getint(line, &i)] = 1;
      printf("B%d[ID%d]", id, id);}
   else { putchar(line[i]); i++; }       }
```

MAIN.C

```
#include <stdio.h>
#define BUFSIZE 512  /* words */
#define NSECS 2      /* no. of seconds of sound */
#define SR 22050     /* sampling rate */

main()
{ int tau, i, n;
  double nexty();
  float outbuf[BUFSIZE];
  int lastbuf;

  tau = 0;
  for (n=0; n<(NSECS*SR)/BUFSIZE; n++)
   { for (i=0; i<BUFSIZE; outbuf[i++] = nexty(tau), tau++);
     write(1, outbuf, 4*BUFSIZE);}
  lastbuf = (NSECS*SR)%BUFSIZE;
  for (i=0; i<lastbuf; outbuf[i++] = nexty(tau), tau++);
  write(1, outbuf, 4*lastbuf);
}
```

SCALE.C

```
/* float to scaled short int */

#include <stdio.h>
#define BUFSIZE 512
#define NSECS 2      /* no. of seconds of sound */
#define SR 22050
#define MAXOUT 32767
#define max(A, B) ((A) > (B) ? (A) : (B))
#define absolute(X) ((X) >= (0.0) ? (X) : (-X))

float inbuf[BUFSIZE];
float store[NSECS*SR];
unsigned short outbuf[BUFSIZE];

main ()
{
 float largest, r;
 int i, n, t, k;
 int nbufs, lastbuf;

 largest = 0;
 t = 0;
 while ((n = read(0, inbuf, 4*BUFSIZE)) > 0)
  { for (i=0; i < n/4; i++)
     { largest = max(largest, absolute(inbuf[i]));
       store[t++] = inbuf[i];}
  }
 nbufs = t/BUFSIZE;
 lastbuf = t%BUFSIZE;
 r = (float)MAXOUT/largest;
 for (k=0; k<nbufs; k++)
  { for (i=0; i<BUFSIZE; i++)
      outbuf[i] = r*store[k*BUFSIZE+i];
      write(1, outbuf, 2*BUFSIZE);
  }
 for (i=0; i<lastbuf; i++)
  outbuf[i] = r*store[k*BUFSIZE+i];
  write(1, outbuf, 2*lastbuf);
}
```