1994

A Simplified Method for the Bilinear s-z Transformation

Dan M. Scott
University of Nebraska-Lincoln

A Simplified Method for the Bilinear s–z Transformation

Dan M. Scott

Abstract—**A new technique for performing the bilinear transformation of polynomials is presented. The technique is both simple to understand as well as efficient in its computer implementation. The key to the method is the way in which the successive derivatives of a particular polynomial are computed. A simple recursion formula is used which can be done either by hand, if desired, or by computer. The order of complexity of the algorithm is found to be $O(n^2)$, while storage requirements are $O(n)$, where n is the degree of the polynomial. The new method will handle completely general bilinear transformations. A computer implementation is presented which was found to be satisfactory for both precision and speed.**

I. Introduction

THE bilinear transformation of a variable in a polynomial arises in several situations in the theory of discrete time systems. For instance it is a fundamental step in designing digital filters [1], switched capacitor filters [2]–[4], and sampled data control systems [5] from a specified rational function of a continuous time system. In this respect the operation is the conversion of the polynomial

$$Q(s) = \sum_{i=0}^{n} b_i s^i \tag{1}$$

to the polynomial

$$P(z) = \sum_{i=0}^{n} a_i z^i \tag{2}$$

where

$$P(z) = (\delta z + \gamma)^n Q\left(\frac{\alpha z + \beta}{\delta z + \gamma}\right). \tag{3}$$

Actually, the form of the bilinear transformation which is most often used is when $\alpha = 1$, $\beta = -1$, $\delta = 1$, and $\gamma = 1$ and, except as noted, this is the specific case of the bilinear transformation considered in the papers discussed below.

Closed forms for the a_k in terms of the b_k have been given [6], but such closed form solutions are of little practical consequence since they require much unwieldy and complicated algebraic manipulation [7]. To avoid this, a variety of different computational methods have been proposed by various authors [7]–[15]. Power [7] first presented a matrix method to obtain the coefficients of $P(z)$. Then Power [8] and Fielder [9] improved the computational complexity of the matrix method. Jury [11] showed how the setting up of the matrix might be further simplified, but left the order of complexity of the method unchanged. Jury and Chan [12] have given a comprehensive discussion of the matrix method. They considered a variety of specific bilinear transformations and showed how the matrix method might be used to solve these transformations. A simpler method has been proposed by Davies [13]. In his method a sequence of simpler transformations are performed on the polynomial which together result in the bilinear transformation. Synthetic division plays an important role in his method. Davies' work is computationally superior to the others as it matches their computational complexity and uses less storage space. Also, unlike other papers, Davies shows how his method can be used for almost all possible bilinear transformations (certain special cases were not considered). Ismail and Vakilzadian [15] presented another approach based on the theory of continued fractions.

The method presented here is an alternative which offers great simplicity as well as computational efficiency.

II. The Algorithm

We wish to find $P(z)$ where

$$\begin{aligned} P(z) &= (\delta z + \gamma)^n Q\left(\frac{\alpha z + \beta}{\delta z + \gamma}\right) \\ &= (\delta z + \gamma)^n \sum_{i=0}^{n} b_i \left(\frac{\alpha z + \beta}{\delta z + \gamma}\right)^i \\ &= \sum_{i=0}^{n} b_i (\alpha z + \beta)^i (\delta z + \gamma)^{n-i} \end{aligned} \tag{4}$$

We shall see that it is convenient to write $P(z)$ in the following form:

$$\begin{aligned} P(z) &= \sum_{i=0}^{n} b_i (\alpha z + \beta)^i (\delta z + \gamma)^{n-i} \\ &= \sum_{\substack{i,j \geq 0 \\ i+j=n}} c_{i,j} (\delta z + \gamma)^i (\alpha z + \beta)^j \end{aligned} \tag{5}$$

where the final sum, as indicated, runs over all nonnegative integers i and j whose sum is n. The $c_{i,j}$ coefficients must be defined by

$$c_{i,j} = b_j \qquad \text{for all integers } i, j \geq 0 \text{ such that } i + j = n. \tag{6}$$

Manuscript received February 1992; revised February 1993.

The author is with the Department of Industrial and Management Systems Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588–0518 USA.

IEEE Log Number 9403874.

Now it may seem that writing $P(z)$ in the form (5) is just adding an unnecessary complication by doubly subscripting the coefficients, $c_{i,j}$, when there is no need. However, we shall see that this notation will actually be quite helpful to us in our development. (Values for $c_{i,j}$ when $i+j \neq n$ will soon be defined.)

It is easily seen from (2) that

$$a_k = \frac{1}{k!}\frac{d^{(k)}}{dz^k}\{P(z)\}\Big|_{z=0} \tag{7}$$

The above equation shows how the various coefficients, a_k, of $P(z)$ can be computed if all orders of derivatives of $P(z)$ evaluated at $z = 0$ are known. Our problem, then, is seen to reduce to the *efficient* evaluation of these derivatives. Consider the following differentiation:

$$\begin{aligned} P'(z) &= \frac{d}{dz}\sum_{\substack{i,j\geq 0\\ i+j=n}} c_{i,j}(\delta z+\gamma)^i(\alpha z+\beta)^j \qquad (8)\\ &= \sum_{\substack{i,j\geq 0\\ i+j=n}} c_{i,j}\Big\{i\delta(\delta z+\gamma)^{i-1}(\alpha z+\beta)^j \\ &\qquad +j\alpha(\delta z+\gamma)^i(\alpha z+\beta)^{j-1}\Big\} \\ &= \sum_{\substack{i,j\geq 0\\ i+j=n-1}} [(i+1)\delta c_{i+1,j}+(j+1)\alpha c_{i,j+1}] \\ &\qquad (\delta z+\gamma)^i(\alpha z+\beta)^j \qquad (9)\\ &= \sum_{\substack{i,j\geq 0\\ i+j=n-1}} c_{i,j}(\delta z+\gamma)^i(\alpha z+\beta)^j \qquad (10) \end{aligned}$$

Notice that the $c_{i,j}$ of (10) (where $i+j=n-1$) are defined in terms of the $c_{i,j}$ of (8) (where $i+j=n$) by the recursion formula

$$c_{i,j} = (i+1)\delta c_{i+1,j} + (j+1)\alpha c_{i,j+1}. \tag{11}$$

It should be noted that the *form* of the derivative in (10) is *identical* to the form of $P(z)$ in (8). Thus the same argument could be applied to $P'(z)$ to obtain $P''(z)$. In fact the argument can be repeated as many times as desired to obtain all higher order derivatives of $P(z)$. Specifically, if we repeat the argument k times the result is

$$\frac{d^{(k)}}{dz^k}\{P(z)\} = \sum_{\substack{i,j\geq 0\\ i+j=n-k}} c_{i,j}(\delta z+\gamma)^i(\alpha z+\beta)^j \tag{12}$$

where the $c_{i,j}$ satisfy (6) and the recursion formula (11). (Note that $c_{i,j}$ values when $i+j>n$ are not defined and are of no interest.) From this we easily obtain

$$a_k = \frac{1}{k!}\frac{d^{(k)}}{dz^k}\{P(z)\}\Big|_{z=0} = \frac{1}{k!}\sum_{\substack{i,j\geq 0\\ i+j=n-k}} c_{i,j}\gamma^i\beta^j \tag{13}$$

Note that in case either γ or β is zero we interpret 0^0 as 1. Equations (6), (11), and (13) may now be used to compute $P(z)$.

III. An Example

This section will illustrate the method on the same example used in [15]. For this problem we have the common case where $\alpha = 1$, $\beta = -1$, $\delta = 1$, and $\gamma = 1$. Suppose that the polynomial, $Q(s)$ is given as

$$Q(s) = 12 + 14s + 17s^2 + 11s^3 + 5s^4 + s^5 \tag{14}$$

In calculating the $c_{i,j}$ it is convenient to use a table. The complete table is shown below, followed by a discussion of how it was computed and how it should be interpreted and used.

	0	1	2	3	4	5
0	7200	2544	540	82	10	1
1	4656	1464	294	42	5	-
2	1596	438	84	11	-	-
3	386	90	17	-	-	-
4	74	14	-	-	-	-
5	12	-	-	-	-	-

The numbers on the left edge of the table are the values of i; the numbers on the top edge of the table are the values of j, and the numbers in the body of the table are the $c_{i,j}$. To generate this table the values of $b_0, b_1, \ldots, b_5$ are entered up the diagonal where $i+j=5$. This is from (6). Then the recursion formula (11) is used to generate the remaining entries. For example

$$c_{3,1} = (4)(1)c_{4,1} + (2)(1)c_{3,2} = 4(14) + 2(17) = 90.$$

Proceeding in this fashion the $c_{i,j}$'s are computed along each 'diagonal' in turn until finally $c_{0,0} = 7200$ is computed.

One interpretation of this table is that it shows us representations for $P(z)$and its derivatives. For example, by reading up the diagonal on which $i+j=3$ and making use of (12) we see that

$$\begin{aligned} P''(z) = {} & 386(z+1)^3 + 438(z+1)^2(z-1)^1 \\ & +294(z+1)^1(z-1)^2 + 82(z-1)^3 \end{aligned}$$

We may now use (13) to determine the coefficients a_k of $P(z)$:

$$\begin{aligned} a_0 &= (12-14+17-11+5-1)/0! = 8,\\ a_1 &= (74-90+84-42+10)/1! = 36,\\ a_2 &= (386-438+294-82)/2! = 80,\\ a_3 &= (1596-1464+540)/3! = 112,\\ a_4 &= (4656-2544)/4! = 88,\\ a_5 &= (7200)/5! = 60. \end{aligned}$$

Thus $P(z) = 8 + 36z + 80z^2 + 112z^3 + 88z^4 + 60z^5$, which agrees with the result obtained in [15]. (Note that the alternating pattern of signs in the computation above results from the fact that $\gamma^i\beta^j = (-1)^j$.)

One refinement of this procedure is worth noting. If we define

$$r_{i,j} = b_j \qquad \text{for all integers } i,j \geq 0 \text{ such that } i+j=n. \tag{15}$$

and compute the other $r_{i,j}$'s by the recursion formula

$$r_{i,j} = [(i+1)\delta r_{i+1,j} + (j+1)\alpha r_{i,j+1}]/(n-i-j). \quad (16)$$

The reader may verify that these $r_{i,j}$ values will satisfy

$$r_{i,j} = \frac{c_{i,j}}{k!} \qquad \text{where } k \text{ is defined by } i+j = n-k \quad (17)$$

This means that

$$a_k = \frac{1}{k!} \sum_{\substack{i,j\geq 0 \\ i+j=n-k}} c_{i,j}\gamma^i\beta^j = \sum_{\substack{i,j\geq 0 \\ i+j=n-k}} r_{i,j}\gamma^i\beta^j \quad (18)$$

Use of the $r_{i,j}$'s is slightly superior from a computational standpoint since it lessens the likelihood of an overflow on the computer and reduces the size of the numbers in the table if one is doing hand calculations. (Incidentally, the $r_{i,j}$ will always be integers as long as α, β, δ, γ, and the b_i coefficients are all integers.)

IV. Computational Analysis and Program

The new algorithm was extremely simple to program. It was programmed in C and run on some example problems on a NeXTStation computer. (The NeXTStation is based on a Motorola 68040 CPU running at 25MHz.) The code was based on the refined method mentioned above which uses the r_{ij} values. In the interests of efficiency it was observed that it is not actually necessary to declare a two dimensional array to hold these r_{ij} values. This is because only a few of them are needed at any one time. In fact it turns out that we need only declare an array which is big enough to hold all the coefficients of $Q(s)$ (i.e. $b_0, b_1, \ldots, b_n$). Thus the memory requirements are trivial even for very large problems. For convenience, two other arrays of similar size are declared which are used to hold the powers of β and γ. The number of operations performed is also quite modest. The 'guts' of the algorithm (that is, all except for input/output) requires $2n(n+2)$ floating point multiplications, $n(n+1)/2$ floating point divisions, $n(n+1)$ floating point additions or subtractions. This makes the method competitive with any of the alternatives. For special cases such as the usual $\alpha = 1, \beta = -1, \delta = 1$, and $\gamma = 1$ both the number of operations and the storage requirements can be more than halved—but requirements are already so low that it would be pointless to specialize the code. A listing of the program together with a sample run are given below. First the program listing:

```
#include¡stdlib.h¿
#include ¡stdio.h¿

main()
{
int Order, k, j;
double sum, alpha, beta, delta, gamma;
double *r, *beta_powers, *gamma_powers;

printf("Please enter alpha, beta, delta, and
  gamma in that order ");
scanf("%lf %lf %lf %lf", &alpha, &beta, &delta,
   &gamma);

printf("Please enter the order of Q(s): ");
scanf("%d", &Order);

r = (double *) malloc((Order+1)*sizeof(double));
beta_powers = (double *) malloc((Order+1)*
  sizeof(double));
gamma_powers = (double *) malloc((Order+1)*
  sizeof(double));

beta_powers[0] = gamma_powers[0] = 1;
for (j=1; j¡=Order; j++)
  {
  beta_powers[j] = beta*beta_powers[j-1];
  gamma_powers[j] = gamma*gamma_powers[j-1];
  }

printf("Now enter the coefficients of Q(s) in
  ascending order \n");
for (k=0; k¡=Order; k++) scanf("%lf", r+k);

for (k=0; k¡Order; k++)
  {
  sum = 0.0;
  for (j=0; j¡=Order-k; j++) sum +=
        beta_powers[j]*gamma_powers[Order-k-j]*r[j];
  for (j=0; j¡Order-k; j++) r[j] =
    ((Order-k-j)*delta*r[j] +
       (j+1)*alpha*r[j+1])/(k+1);
  r[Order-k] = sum;
  }
for (k=0; k¡=Order; k++)
  printf("a[%d]=%20.17le\n",k,r[Order-k]);
}
```

And now the sample run:

```
Please enter alpha, beta, delta, and gamma in
that order 1 -1 1 1
Please enter the order of Q(s): 10
Now enter the coefficients of Q(s) in ascending
```

```
order
1 6.392453 20.431729 42.802061 64.882396
74.233429 64.882396 42.802061 20.431729
6.392453 1
a[0]= 5.79299999999705050e-03
a[1]= -1.06581410364015030e-14
a[2]= 1.09289500000024020e+00
a[3]= -1.13686837721616030e-13
a[4]= 2.65985379999981430e+01
a[5]= 4.54747350886464120e-13
a[6]= 1.88283805999996730e+02
a[7]= 0.00000000000000000e+00
a[8]= 4.62768261000000170e+02
a[9]= 0.00000000000000000e+00
a[10]= 3.45250707000000030e+02
```

The example run above is for the same tenth order Butterworth low-pass filter presented in [15]. As observed there, the symmetry of this polynomial means that the coefficients on the odd power terms *should* equal zero. The (small) nonzero numbers observed in the output are due to finite precision of the double precision floating point numbers used in the program.

An investigation into the accuracy of results obtained from this program was performed. Random polynomials of degree 200 were generated, and the s–z transformation was performed both by the program above as well as by a similar (though vastly slower) program written in Mathematica which used a precision of 100 digits. A comparison of the coefficients obtained showed that in all of the examples tested the largest *relative* error obtained was less than 10^{-12}. Though, obviously, computer speed varies considerably from one machine to another, one may get a general idea of the speed of the algorithm by the fact that the average execution time (exclusive of input/output time) for a polynomial of degree 200 on the NeXTStation was 0.175 seconds.

V. Summary

In this paper a new technique for computing an arbitrary bilinear transformation was presented. An analysis of the number of floating point operations necessary using this approach shows that $2n(n+2)$ multiplications, $n(n+1)/2$ divisions, and $n(n+1)$ additions are required. The memory requirements were only $O(n)$ compared to $O(n^2)$ for almost all other methods. This makes it competitive with any of the alternatives. Unlike other approaches, this method is completely general—any bilinear transformation can be performed, even in cases where one or more of the α, β, δ, or γ are zero. The simplicity of the method makes it appropriate for class explanation and hand calculation. An efficient computer implementation was written in C, and was found to be satisfactory for both precision and speed.

References

[1] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[2] M. S. Lee and C. Chang, "Low sensitivity switched-capacitor ladder filters," *IEEE Trans. Circuits Syst.*, vol. CAS-27, pp. 475–480, June 1980.

[3] K. Martin and A. S. Sedra, "Exact design of switched capacitor bandpass filters using coupled-biquad structures," *IEEE Trans. Circuits Syst.*, vol. CAS-27, pp. 469–475, June 1980.

[4] M. Ismail and T. Bacon, "A new approach to synthesis of switched-capacitor filters in the Z-domain," *Proc. 28th Midwest Symp. Circuits Syst.*, Aug. 1985, pp. 386–389.

[5] E. I. Jury, *Inners and Stability of Dynamic Systems*. New York: Wiley, 1974.

[6] E. I. Jury, *Theory and Application of the Z-Transform Method*. New York: Wiley, 1964.

[7] H. M. Power, "The mechanics of the bilinear transformation," *IEEE Trans. Educ.*, vol. E-10, pp. 114–116, June 1967.

[8] H. M. Power, "Comments on the mechanics of the bilinear transformation," *IEEE Trans. Educ.*, vol. E-11, p. 159, June 1968.

[9] D. C. Fielder, "Some classroom comments on bilinear transformation," *IEEE Trans. Educ.*, vol. E-13, p. 105, Aug. 1970.

[10] K. A. Moore, "An APL program for bilinear transformation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-22, pp. 225–226, June 1974.

[11] E. I. Jury, "Remarks on the mechanics of bilinear transformation," *IEEE Trans. Audio Electro Acoust.*, vol. AU-21, pp. 380–382, Aug. 1973.

[12] E. I. Jury and O. W. C. Chan, "Combinitorial rules for some useful transformations," *IEEE Trans. Circuit Theory*, vol. CT20, pp. 476–480 Sept. 1973.

[13] A. C. Davies, "Bilinear transformation of polynomials," *IEEE Trans. Circuits Syst.*, vol. CAS-21, pp. 792–794, Nov. 1974.

[14] C. F. Chen and Y. T. Tsay, "A new formula for the discrete-time system stability," *Proc. IEEE*, pp. 1200–1202, Aug. 1977.

[15] M. Ismail and H. Vakilzadian, "The computer implementation of bilinear $s-z$ transformation using new continued fraction algorithms," *IEEE Trans. Educ.*, vol. 32, pp. 270–279, Aug. 1989.

Dan Scott received his B.S. from Iowa State University in Mathematics and his Ph.D. from Stanford University in Operations Research.

He has taught Mathematics and Physics at Iowa State University, Mathematics at Rose-Hulman Institute of Technology, and currently teaches Industrial Engineering at The University of Nebraska-Lincoln. He has published research in mathematics, chemistry, physics, genetics, and operations research. His current research interests include network flow algorithms, combinatorial optimization, and production scheduling.