Algorithms from Blossoms

Stephen Mann
Computer Science Department
University of Waterloo
Research Report CS-96-29

Abstract. Blossoming is a theoretical technique that been used to develop new CAGD theory. However, once we have developed this theory, we need to devise algorithms to implement it. In this paper, I will discuss converting blossoming equations into code, noting techniques that can be used to develop efficient algorithms. These techniques are illustrated by considering the operations of basis conversion and polynomial composition.

§1. Introduction

Blossoming has been used successfully to analyze and develop new CAGD theory. Once we have developed a new theory, we must to convert our blossom equations into algorithms. Although there is usually an obvious transformation, the resulting code will be extremely inefficient. In this paper, I will show methods for creating efficient code from blossom equations.

Blossoming analysis is based on the *blossom*, which is defined as a symmetric and multi-affine (affine in each argument) map of n arguments. The following theorem [8] states that polynomials and blossoms are essentially the same.

The Blossoming Principle. *There is a one-to-one correspondence between degree n polynomials, $F : X \to Y$, and n–affine blossoms, $f : X^n \to Y$, such that*

$$F(u) = f(\underbrace{u \ldots u}_{n}),$$

where X and Y are spaces of arbitrary dimension.

Here, "multiplication" is tensor multiplication. When taking the tensor of scalars, I will separate them with commas. Note also that some of the figures use comma separated argument lists (eg., $f(a,b)$ instead of $f(ab)$).

Ramshaw and others have successfully used blossoming to analyze existing CAGD algorithms and to develop new theory. These theoretical results

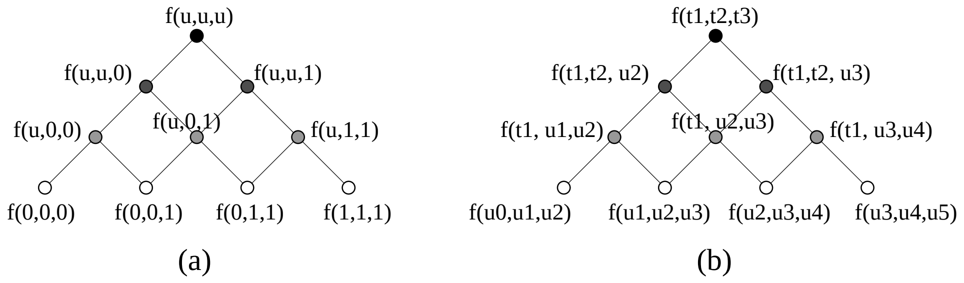

Fig. 1. de Casteljau evaluation of (a) a Bézier curve and (b) a blossom.

are expressed in equations involving sums of the blossom at various sets of arguments. The heart of blossom equations are evaluations of the the blossom. To convert these equations into code, we must perform two tasks. First, we must iterate over the required argument sets, and second, we must evaluate the blossom at these argument sets.

Normally, we evaluate the blossom using a variation of de Casteljau's algorithm. The de Casteljau's algorithm uses repeated linear interpolation to evaluate Bézier simplices. In addition to evaluating a Bézier simplex for position, the intermediate results can be used to compute derivatives. Figure 1 (a) shows the data flow diagram for a de Casteljau evaluation of a cubic Bézier curve parameterized over $[0,1]$. In this diagram, the blossom values $f(0,0,0), f(0,0,1), f(0,1,1), f(1,1,1)$ correspond the the control points of the Bèzier curve.

We can extend the de Casteljau algorithm to blossoms by starting with the blossom values $f(u_0u_1u_2)$, $f(u_1u_2u_3)$, $f(u_2u_3u_4)$, and $f(u_3u_4u_5)$, and by computing $f(t_1t_2t_3)$ as shown in Figure 1 (b), where $u_1 \leq u_2 \leq u_3 < u_4 \leq u_5 \leq u_6$. Note that these knots and control points are the knots and control points for a single segment B-spline. (Although the examples given in this figure are for cubic curves, these ideas generalize to polynomials of any degree with domain simplices of any dimension.)

Although we often want to perform a complete evaluation of the blossom, sometimes we only want a *partial evaluation*. For example, in Figure 1 (b), we might want to stop the de Casteljau evaluation after having evaluated f at t_1, yielding the light gray control points. Note the close relationship between partial evaluation and knot insertion: In Figure 1 (b), the light gray points together with $f(u_0u_1u_2)$ and $f(u_3u_4u_5)$ are the B-spline control points resulting from inserting the knot t_1 into the B-spline specified by the u_i and corresponding blossom values.

The way to make efficient algorithms from blossom equations is by making effective use of these intermediate blossom values. In particular, the most expensive step of the de Casteljau algorithm is the first one. It is these values that we need to best reuse in order to devise efficient algorithms.

In this paper, I will illustrate the following three techniques for devising efficient algorithms from blossom equations:

(1) Evaluate the blossom only once for each permutation of its arguments.
(2) Reuse of partial evaluations.
(3) Converting to a better basis.

The first technique is well known; the other two have been used in an ad hoc fashion for a variety of algorithms. I will present these techniques by illustrating their use for basis conversion and for polynomial composition. The point of this paper is to illustrate the techniques. Thus, I will not give many details about these algorithms, and I will restrict my discussion to curves of degrees 2 and 3 (although I will note when the algorithms generalize to higher degrees and higher dimensional domains). However, in Appendix A, I discuss one of the basis conversion algorithms, since while Barry and Goldman discuss a restricted form of the algorithm, the generalization discussed in Appendix A has not been presented elsewhere.

§2. Basis Conversion

The basis conversion problem is the following: Given a polynomial represented in one basis, find its representation relative to another basis. When expressed in blossom terms, a basis is represented as a set of *knots*, and a polynomial's representation relative to this basis is the evaluation of the blossom at consecutive knots.

For a degree n polynomial with a one dimensional domain, its knot vector is $\{a_n, \ldots, a_1, b_1, \ldots, b_n\}$, and the polynomial is specified by the blossom values

$$f(a_n \ldots a_1),\ f(a_{n-1} \ldots a_1 b_1),\ \ldots,\ f(b_1 \ldots b_n). \qquad (1)$$

If we want to convert from a basis $\{a_n, \ldots, a_1, b_1, \ldots, b_n\}$ to a basis $\{a'_n, \ldots, a'_1, b'_1, \ldots, b'_n\}$, we need to compute the blossom values

$$f(a'_n \ldots a'_1),\ f(a'_{n-1} \ldots a'_1 b'_1),\ \ldots,\ f(b'_1 \ldots b'_n).$$

from the values given in Equation (1). The obvious algorithm is to explicitly perform these evaluations. If we use the de Casteljau algorithm, this would require $(n+1)n(n-1)/2$ affine combinations, as illustrated for cubic polynomials in Figure 2. In the next three sections, I will examine (from blossoming and algorithmic viewpoints) the improvements to this algorithm described by Barry and Goldman [1]. Note for basis conversion, there is little or no concern about evaluating at a blossom value at multiple permutations of its arguments.

§3. Sablonnière's Algorithm

Sablonnière devised a more efficient basis conversion algorithm [9]. From a blossoming viewpoint, the way to understand this algorithm is to reorder the de Casteljau evaluations for the control points for our new basis so that these evaluations have common intermediate values.

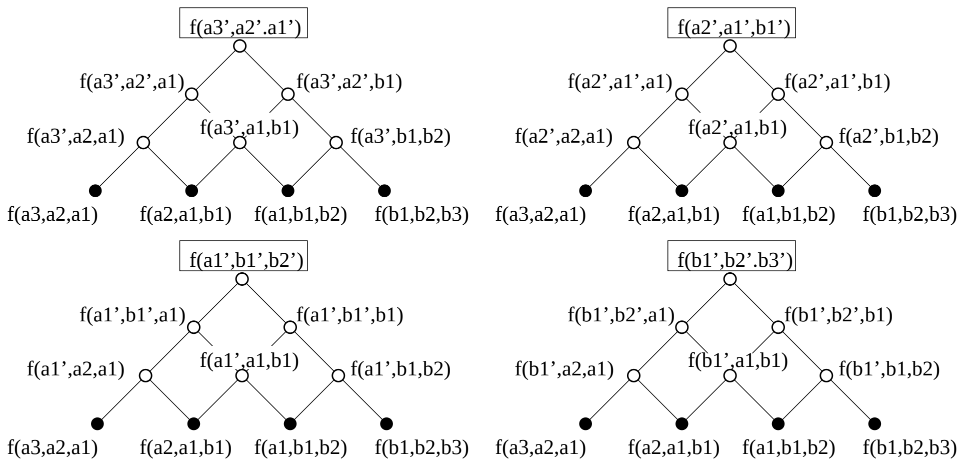

Fig. 2. Basis Conversion by full evaluation.

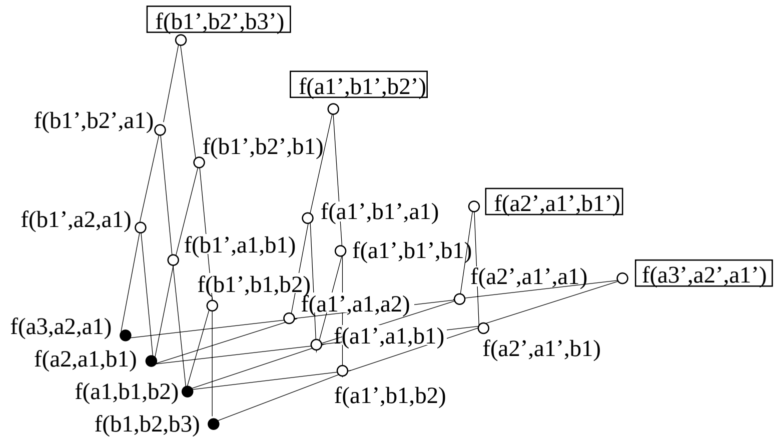

Fig. 3. Sablonnière's Basis Conversion Algorithm.

For example, consider what happens when we compute $f(a'_n \dots a'_1)$ and $f(a'_{n-1} \dots a'_1, b'_1)$. If we compute the latter point first, then at the next to last step of de Casteljau's algorithm we will have computed $f(a'_{n-1} \dots a'_1 a_1)$ and $f(a'_{n-1} \dots a'_1 b_1)$. From these two points we can compute $f(a'_{n-1} \dots a'_1 b'_1)$ with a single affine combination and $f(a'_n \dots a'_1)$ with one additional affine combination. Thus, we have reduced the cost of computing these two points by almost a factor of 2. The computation of the remaining blossom values can also be reduced, although by a smaller factor.

The complete Sablonnière computation for cubics is illustrated in Figure 3. From this figure, we see that the algorithm chooses knots from starting with the first knot in a group rather than the last knot. This allows for a high amount of reuse of the intermediate blossom values.

For a degree n curve, Sablonnière's algorithm requires $\binom{n+3}{n} - (n+1) = (n+1)n(n-1)/6 - (n+1)$ affine combinations, or roughly a factor of 3 fewer affine combinations than required by fully evaluating each blossom value.

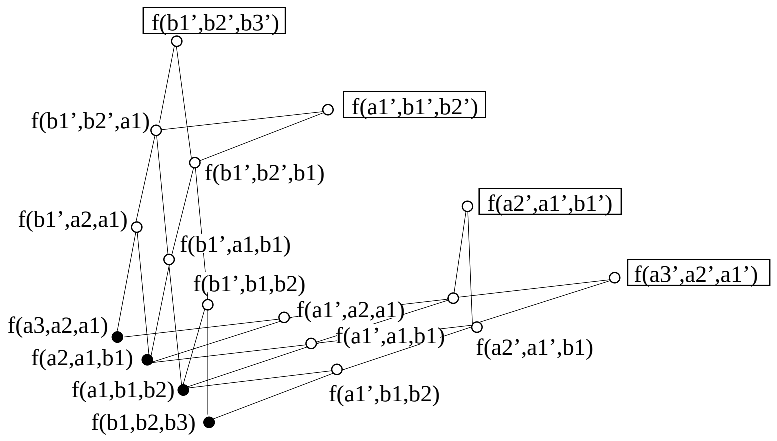

Fig. 4. Recursive Sablonnière's Algorithm.

This algorithm generalizes to domains of arbitrary dimension in the obvious manner.

§4. Recursive Sablonnière's Algorithm

In the example in Figure 3, we see that the blossom value $f(a_1' b_1' b_2')$ can be computed with a single affine combination instead of three (Figure 4). Barry and Goldman developed this idea for all curves of degree one less than a power of 2. For these degrees, their algorithm makes optimal reuse of intermediate values.

The number of affine combinations required by Recursive Sablonnière's Algorithm can be expressed as a recurrence whose closed form solution is unknown, but its runtime clearly falls between that of Sablonnière's Algorithm and Goldman's algorithm.

We can generalize this algorithm to work for one-dimensional domains of arbitrary degrees without many difficulties. The generalization to functions with domains of arbitrary dimensions is more complex. A discussion of these generalizations can be found in Appendix A.

§5. Goldman's Algorithm

The preceding basis conversion algorithms are using the final points computed by de Casteljau's algorithm. Goldman's observation is that sometimes we can use the intermediate points computed by de Casteljau's algorithm. Goldman proceeded by performing a complete de Casteljau evaluation to compute one of the desired blossom values. Then, to compute the remaining points, he starts a second de Casteljau evaluation *starting from the edge* of the previous de Casteljau triangle. The desired control points then lie along one of the *edges* of this second de Casteljau triangle. Figure 5 illustrates this algorithm for cubics. The first evaluation and use of control points along its edge is effectively a change of basis.

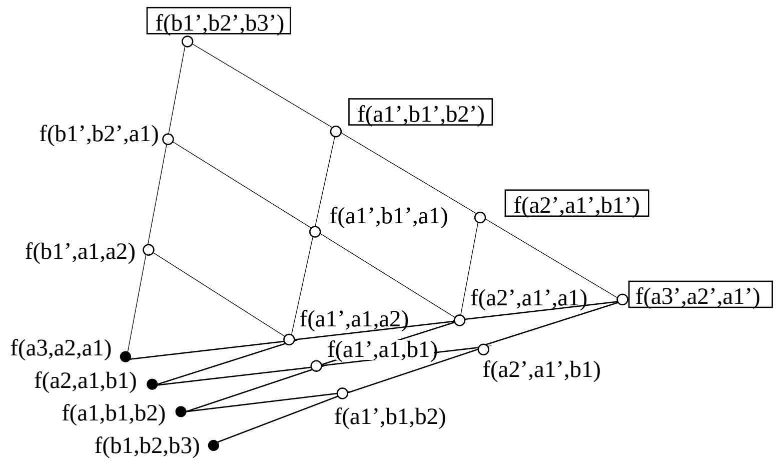

Fig. 5. Goldman's Algorithm.

Goldman's algorithm requires $n(n-1)$ affine combinations to basis convert a degree n curve, or roughly a factor of $(n+1)/3$ savings over Sablonnière's algorithm. Note that this algorithm generalizes to domains of arbitrary dimensions, where for a domain of dimension k, we will need to perform $k+1$ complete de Casteljau evaluations.

§6. Polynomial Composition

The univariate polynomial composition problem is the following:

Given. *Affine spaces $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ (of dimensions 1, 1, and K_Z respectively), control points $\{G_i\}$, $i = 0 \ldots \ell$, defining a Bézier curve $G : \mathcal{X} \to \mathcal{Y}$ of degree ℓ relative to a domain simplex $\Delta_\mathcal{X} \subset \mathcal{X}$, and control points $\{F_j\}$, $j = 0 \ldots m$ defining a degree m Bézier curve $F : \mathcal{Y} \to \mathcal{Z}$ relative to a domain simplex $\Delta_\mathcal{Y} \subset \mathcal{Y}$.*

Find. *The control points $\{H_k\}$, $k = 0 \ldots m\ell$ of the degree $m\ell$ Bézier curve $H = F \circ G$ relative to $\Delta_\mathcal{X}$.*

Solution. *If f denotes the blossom of F, then*

$$H_k = \sum_{\substack{I \in \mathcal{Z}_\ell^m, \\ |I|=k}} \mathcal{C}(I) f(G_I), \tag{2}$$

where G_I with $I = (i_1, ..., i_m)$ is an abbreviation for $(G_{i_1} \ldots G_{i_m})$, and $\mathcal{C}(I)$ is a combinatorial function. Here, the I are known as hyper-indices.

Note that when computing all the H_k, we will need to evaluate f at all combinations of G's control points.

A proof of this result can be found in the paper by DeRose et al. [2], along with a reasonably efficient algorithm, and a discussion applications of

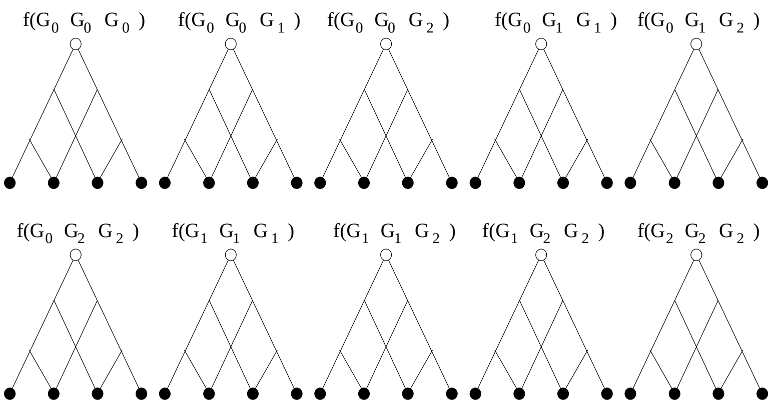

Fig. 6. Polynomial Composition Algorithm.

polynomial composition. Mann and Liu later developed two improved algorithms [7]. In this section, I will discuss all three algorithms, and analyze the blossoming techniques used.

As a first comment, note that for each blossom value $f(a_1 \ldots a_n)$, Equation (2) sums the blossom evaluated at all permutations of these arguments. Since all permutations of any particular hyper-index appear in the equation for a single H_k, we can evaluate f at just one permutation of a hyper-index, and weight it by the number of permutations of its arguments. Figure 6 shows the 60 affine combinations used when composing a cubic curve with a quadratic curve when we evaluate at only one permutation of each set of blossom arguments.

Note that all the polynomial composition algorithms discussed in this paper generalize to arbitrary degrees and dimensions.

§7. 1993 Algorithm

DeRose et al. made the same observation about polynomial composition that was made for Sablonnière's algorithm: When computing the required blossom values using de Casteljau's algorithm, many of the intermediate values are the same. To reuse these partial evaluations, they imposed an ordering on hyper-indices. They then evaluated the blossom in the order imposed on the hyper-indices by evaluating one argument at a time as follows:

(1) For each i,
   (a) Partially evaluate f at G_i, giving f'.
   (b) Recursively evaluate f' in the same fashion, but only at G_j where $j \geq i$
(2) For each complete evaluation of f, add the appropriate weighted contribution to the corresponding control point of H.

Note that this algorithm automatically evaluates f at only one permutation of each required argument set. Figure 7 shows the 31 affine combinations used when composing a cubic curve with a quadratic curve. Details on this algorithm can be found in several papers [2,5,7].

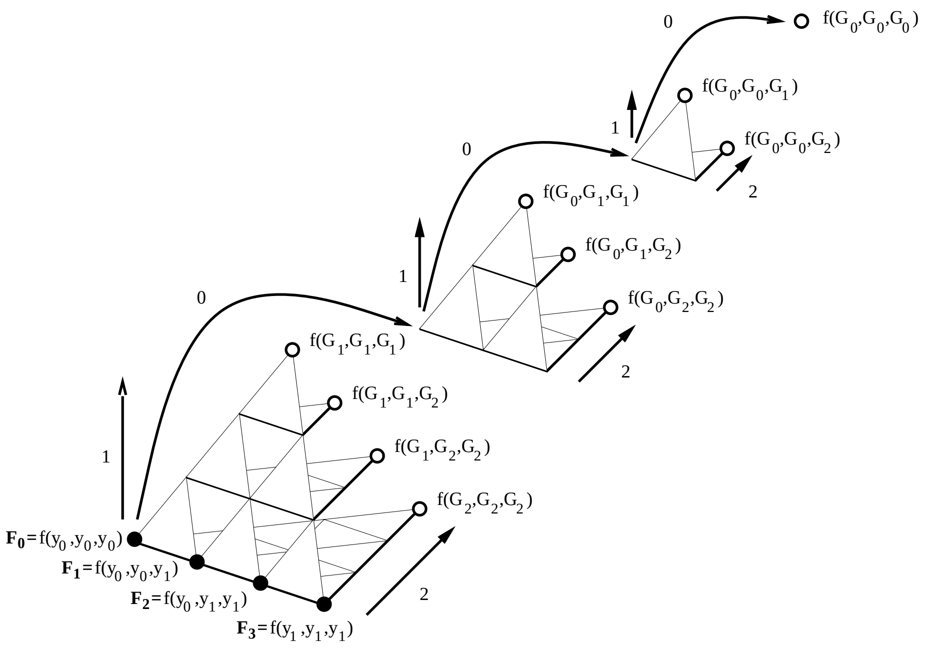

Fig. 7. 1993 Polynomial Composition Algorithm.

§8. Recursive Algorithm

Just as we can improve Sablonnière's algorithm, we can improve the 1993 algorithm. The 1993 algorithm orders the indices within a hyper-index in increasing order, and uses a lexicographical ordering of the hyper-indices. We can make optimal reuse of our evaluations by using a different ordering on both the indices within a hyper-index and of the hyper-indices.

The observation to exploit is the following: In the steps of the de Casteljau algorithm, the early levels of evaluation are more expensive than later ones. Thus, to make the best reuse of partial evaluations, we want to minimize the number of early level evaluations. We can achieve this minimization by ordering the indices within a hyper-index from most to least repetitions. I.e., our hyper-indices will be

$$I = (i_1^{r_1}, \ldots, i_k^{r_k})$$

where $r_j \leq r_{j+1}$. To make our hyper-indices unique up to permutations, we add the additional condition that $i_j < i_{j+1}$ when $r_j = r_{j+1}$.

This ordering of the indices within the hyper-indices, and the subsequent ordering of the hyper-indices allows for more effective reuse of the partial evaluations of the blossom. Figure 8 shows the 27 affine combinations used when composing a cubic curve with a quadratic curve; the dotted lines show the evaluations that would have been used by the 1993 Algorithm. Details on this recursive algorithm can be found in the Mann-Liu technical report [7].

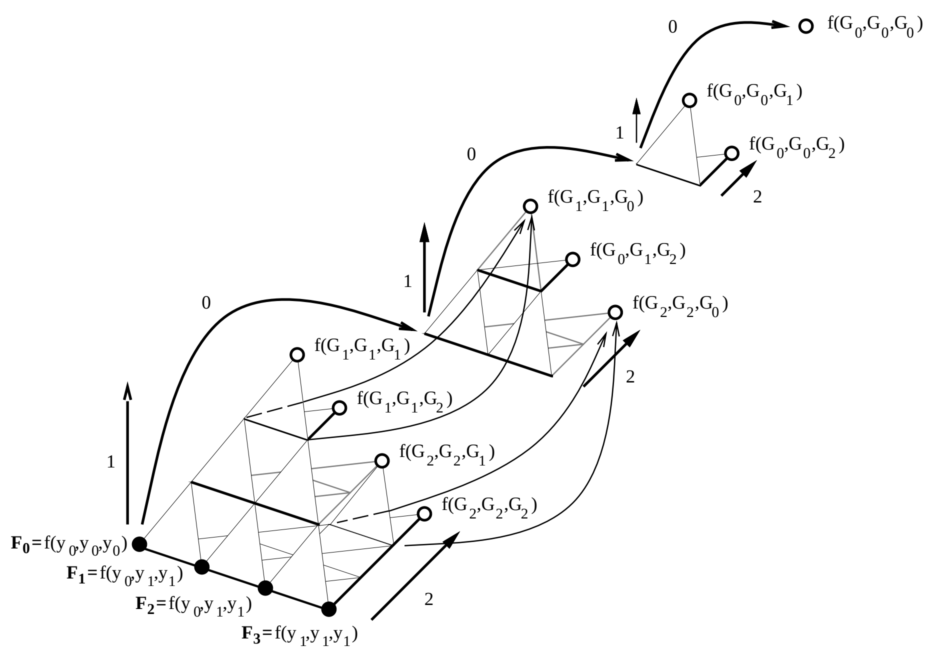

Fig. 8. Recursive Algorithm.

§9. Optimal Algorithm

Just as we can improve Sablonnière's algorithm by performing a change of basis, we can improve the 1993 algorithm by first performing a change of basis. However, for polynomial composition, we get a more remarkable result: If we change to a basis consisting of G's control points, and then apply the 1993 algorithm, then *every* intermediate value computed (after the basis conversion) is a blossom value used in Equation (2). Further, each of these values is computed exactly once.

Figure 9 shows the affine combinations used when composing a cubic curve with a quadratic curve. Details on this algorithm can be found in several papers [7][5].

§10. Knot Insertion, Partial Evaluation, and Knot Swapping

In the previous sections, we have seen illustrations of some techniques to make efficient algorithms from blossom equations. These algorithms involve evaluating the blossom and using reusing these partial evaluations. The operation of partial evaluation is similar to that of knot insertion. A third similar technique is *knot swapping*, where essentially we take the light gray points of Figure 1 (b) together with one of the $f(u_0u_1u_2)$ and $f(u_3u_4u_5)$. In this section, I will discuss these three techniques.

All three operations have the same computational cost. For curves, knot insertion is somewhat more natural, as it gives us exactly the set of knots we commonly want. However, knot insertion does not cleanly generalize to domains of higher dimension, while partial evaluation and knot swapping both

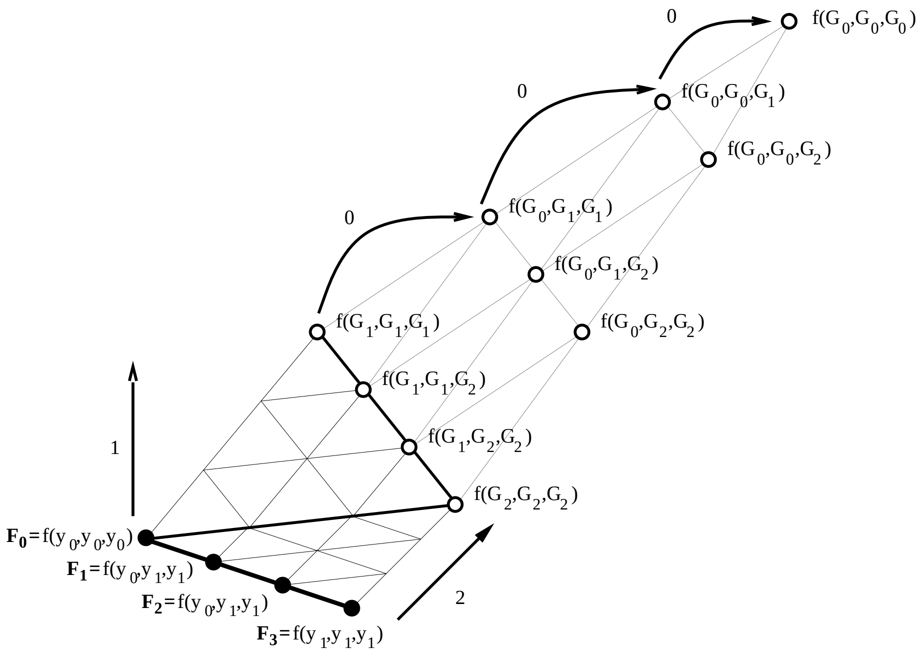

Fig. 9. Optimal Composition Algorithm. Solid lines show the basis conversion.

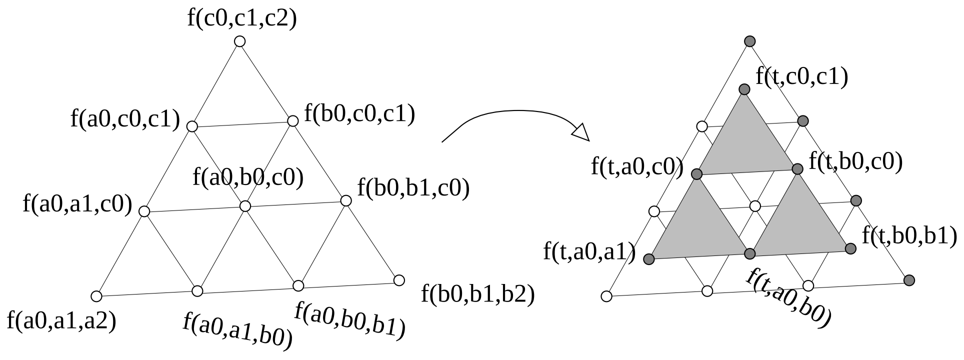

Fig. 10. Evaluation of a B-patch.

do. When we generalize to dimension k, a degree n patch has $k+1$ sets of n knots [10]. Consider the diagram in Figure 10. On the left are the control points of a B-patch with knots $\{a_0, a_1, a_2;\ b_0, b_1, b_2;\ c_0, c_1, c_2\}$. If we perform one level of the de Casteljau algorithm, we get the shaded panels on the right. In this right-hand figure, I have drawn the original control points underneath, but have only labeled the new points.

Looking at this diagram, it is clear that the shaded panels represent the partial evaluation of the original B-patch. If we take the shaded panels together with any edge of the original control net, we get the knot-swapped representation of the patch, where t has replaced one of a_2, b_2, or c_2 (in the figure, the shaded points are the control points for the patch with knot net

$\{t, a_0, a_1;\ b_0, b_1, b_2;\ c_0, c_1, c_2\}$). On the other hand, it is unclear what knot insertion even means in this context. Thus, except when working with curves, one should use either partial evaluation or knot swapping.

Additional tricks can be played when using knot swapping, and it may prove to be slightly more useful than knot swapping. See Liu's thesis [4] for a more complete discussion of all three techniques.

A C++ library implementing these ideas is discussed in [4][6]

§11. Conclusions

Blossoming is a useful technique for developing CAGD theory. Once we have developed the theory, however, we need to convert blossom formulas into program code. This primarily involves evaluating the blossom at argument sets specified by our blossom equations. In this paper, we have seen three techniques to write efficient code: evaluating the blossom only once for each permutation of its arguments; reuse of partial evaluations; and conversion to a basis where the computation requires fewer blossom evaluations.

A few notes are in order. Based on efficiency considerations, there is no point in using Sablonnière's algorithm for basis conversion or for using the 1993 algorithm for polynomial composition since more efficient algorithms are available. However, Sablonnière's algorithm and the 1993 algorithm algorithms have a couple of advantages over the more efficient algorithms: they are easier to code and (in comparison to Goldman's algorithm and the Optimal composition algorithm) they are numerically more stable.

Thus, when converting from blossom formulas to code, it is almost always advantageous to consider a straight-forward reuse of intermediate blossom values. This is readily found by looking for a natural ordering on the blossom indices we are evaluating at. Once this first improvement is made, we should check to see if we can get a further improvement by either using a better ordering of the blossom arguments, or by first converting to a better basis.

Finally, note that these ideas are just starting points for optimizing blossom equations. Sometimes further manipulation will be necessary.

§Appendix A. Generalized Recursive Composition Algorithm

Barry and Goldman developed an algorithm for basis conversion that works when the degree is one less than a power of 2. In this section, I will present a generalization of their algorithm to arbitrary degrees, and discuss the difficulties of generalizing to domains of arbitrary dimension.

I will give Perl code that iterates through the evaluation indices for both the Barry-Goldman algorithm and for my generalizations. These routines both make calls to the routine `PrintI` (Figure 11), which records and prints the indices. In actual basis conversion code, the calls to `PrintI` would be replaced with partial evaluations at the domain point with the corresponding index.

A Perl script that iterates through the multi-indices used by the Barry-Goldman algorithm is given in Figure 12. When working with a domain

```
# A fancy "print" routine.
$prevs = -1;
sub PrintI {
        local($i,$j,$s) = @_;

        if ( $s != $prevs+1 ) {
                local($i);
                for ($i=0; $i<$s; $i++) {
                        print "  .     ";
                }
        }
        $prevs = $s;
        printf "a$i,%-2d ",$j;
        $li[$s] = "a$i,$j";
        if ( $s == $gn -1 ) {
                $li[$s] = "a$i,$j";
                print " ::    @li ";
                print "\n";
        }
}
```

Fig. 11. A print routine.

of dimension 1, our knot net is $\{a_{0,0},\ldots,a_{0,n-1};\ a_{1,0},\ldots,a_{1,n-1}\}$. All of our blossom evaluations will be of the form $f(a_{0,0},\ldots,a_{0,k},\ a_{1,0},\ldots,a_{1,n-k})$ (although we will perform the evaluation in a different ordering of these knots).

The key to understanding why their algorithm works (and to understanding the generalization to arbitrary degree) is that in a complete evaluation of a blossom of a degree 2^N-1 polynomial, there will be more evaluations with indices from one set or the other (eg., either more $a_{0,j}$ or more $a_{1,j}$). Thus, we can evaluate at 2^{N-1} of one type of index, and then recusively compute the remaining indices at which to evaluate. This recursion can be performed by only knowing how many of each index we have already evaluated at (this is needed to determine the remaining subscripts).

The point is, in our recursion, we do not have to scan through the indices at which we have already evaluated. The remaining indices depend only on the value of N and how many of each type of index we have evaluated at. The reason Barry and Goldman restricted the algorithm to degrees of 2^N-1 is to ensure that the number of remaining indices is odd at each level of recursion. The problem is if the degree is even, then we have to avoid evaluating at both $f(a_{0,0},\ldots,a_{0,n/2},\ a_{1,0},\ldots,a_{1,n/2})$ and $f(a_{1,0},\ldots,a_{1,n/2},\ a_{0,0},\ldots,a_{0,n/2})$.

This problem (of avoiding evaluating twice) is easily avoided. If the degree is even at any level of the recursion, then we evaluate at one of our indices $n/2$ times before making the recursive call, and evaluate at the other indexs $n/2+1$ times before making the recursive call. Perl code for this generalized

```
&Improved($gn,0);     # $gn = 2**$N-1

@M = ();             # Keep track of how many so far
sub Improved {
        # n = 2^N-1 - the number of unevaluated arguments
        # m - number of indices selected so far
        local($n,$m)=@_;
        local($i,$j,$s);

        if ( $n == 0 ) { return; }

        $s = int(($n+1)/2);
        # Iterate over all dimensions
        for ( $i=0; $i<=1; $i++ ) {
                local($oldM);
                $oldM = $M[$i];
                for ( $j=0; $j<$s; $j++ ) {
                        &PrintI($i,$oldM+$j,$m+$j);
                }
                $M[$i] += $s;
                &Improved($n-$s,$m+$s);
                $M[$i] = $oldM;
        }
}
```

Fig. 12. Barry-Goldman recursive basis conversion algorithm.

algorithm is given in Figure 13. Note that the only change between the two algorithms is the additional `if` statement at lines 18–20.

Generalizing to domains of arbitrary dimension proves to be more troublesome. The generalized basis conversion problem converts a degree n polynomial with domain of dimension k represented in a basis defined by a knot net

$$A = \{a_{0,0}, \ldots, a_{0,n};\ a_{1,0}, \ldots, a_{1,n};\ \ldots;\ a_{k,0}, \ldots, a_{k,n}\}$$

to a representation in a basis defined by a knot net

$$B = \{b_{0,0}, \ldots, b_{0,n};\ b_{1,0}, \ldots, b_{1,n};\ \ldots;\ b_{k,0}, \ldots, b_{k,n}\}.$$

Define

$$a_{\vec{i}} = \prod_{\ell=0}^{d} \prod_{m=0}^{i_{\ell-1}} a_{\ell,m} = a_{0,0} \ldots a_{0,i_0} a_{1,0}, \ldots a_{d,0} \ldots a_{d,i_d},$$

where $\vec{i} = (i_0, \ldots, i_n)$. Then a representation of a polynomial relative to a knot net A (i.e., the set of control points) is given by evaluations of the blossom at $a_{\vec{i}}$ for all $\vec{i}$ with $|\vec{i}| = n$.

```
&Improved($gn,0,0);

@M = ();                # Keep track of how many so far
sub Improved {
        # n - the number of unevaluated arguments
        # m - number of indices selected so far
        local($n,$m)=@_;
        local($i,$j,$s);

        if ( $n == 0 ) { return; }

        $s = int(($n+1)/2);
        # Iterate over all dimensions
        for ( $i=0; $i<=1; $i++ ) {
                local($oldM);
                $oldM = $M[$i];
                if ( 2*$s == $n  &&  $i == 1 ) {
                        $s++;
                }
                for ( $j=0; $j<$s; $j++ ) {
                        &PrintI($i,$oldM+$j,$m+$j);
                }
                $M[$i] += $s;
                &Improved($n-$s,$m+$s);
                $M[$i] = $oldM;
        }
}
```

Fig. 13. Generalized recursive basis conversion algorithm (dimension = 1).

So our goal is to convert from the set of control points

$$f(a_{0,0}\dots a_{0,n-1}a_{0,n}),\ f(a_{0,0}\dots a_{0,n-1}a_{1,0}),\ \dots f(a_{0,0}\dots a_{0,n-1}a_{k,0})$$
$$f(a_{0,0}\dots a_{0,n-2}a_{1,n-1}a_{0,n}),\ f(a_{0,0}\dots a_{0,n-2}a_{1,n-1}a_{1,0}),\ \dots$$
$$f(a_{0,0}\dots a_{0,n-2}a_{1,n-1}a_{k,0})$$
$$\vdots$$

to the control points

$$f(b_{0,0}\dots b_{0,n-1}b_{0,n}),\ f(b_{0,0}\dots b_{0,n-1}b_{1,0}),\ \dots f(b_{0,0}\dots b_{0,n-1}b_{k,0})$$
$$f(b_{0,0}\dots b_{0,n-2}b_{1,n-1}b_{0,n}),\ f(b_{0,0}\dots b_{0,n-2}b_{1,n-1}b_{1,0}),\ \dots$$
$$f(b_{0,0}\dots b_{0,n-2}b_{1,n-1}b_{k,0})$$
$$\vdots$$

Since we will have to evaluate

$$f(b_{0,0}\dots b_{0,n-1}b_{0,n}),\ \dots f(b_{k,0}\dots b_{k,n-1}b_{k,n})$$

we will want to make maximal reuse of the intermediate values computed when we calculate these values. Although certain aspects of the "best" algorithm for computing these values are clear, such as Barry and Goldman's observation that we should work from the partial evaluations

$$f(b_{0,0}\dots b_{0,n/2}),\ f(b_{1,0}\dots b_{1,n/2}),\ \dots, f(b_{k,0}\dots b_{k,n/2}),$$

other aspects are less clear (eg., what's the optimal way to compute

$$f(b_{0,0}\dots b_{0,i_0}\dots b_{k,0}\dots b_{k,i_k})$$

when all i_j are less than $n/2$). In particular, we want to avoid computing any value twice, and we want the runtime cost to be dominated by the blossom evaluations rather than multi-index sorting, etc.

A first attempt at creating code to allow for arbitrary dimensions would be to directly generalize the code of Figure 13 by changing the bounds on the `for` loop of line 15 from 1 to k and by changing line 13 from `$s = int(($n+$k)/2)` to `$s = int(($n+1)/($k+1))`. Unfortunately, while the resulting code would iterate through all the desired hyper-indices, most hyper-indices would be generated multiple times.

Wayne Liu suggested using an idea similar to that of the Recursive Composition Algorithm. To use this idea, we must reorder the $\vec{i}$ so that the i_j's are in non-decreasing order, eg.,

$$a_{\vec{i}} = a_{j_0,0}\dots a_{j_0,i_{j_0}}\dots a_{j_d,0}\dots a_{d,i_{j_d}},$$

where $i_{j_p} \le i_{j_{p+1}}$, with $i_{j_p} = i_{j_{p+1}}$ only if $j_p < j_{p+1}$. We then partially evaluate the blossom at prefixes of these tensors.

A variation of the code in Figure 13 of [7] efficiently iterates through and evaluates the blossom at the appropriate knots. This code is given in Figure 14. The code has been compressed somewhat so that it fits on one page.

Although better than Sablonnière's algorithm, this code is not optimal in any sense of the word. In particular, it does not generalize the Barry-Goldman recursive basis conversion algorithm. In Figure 15, we see the sequences produced by both algorithms for polynomials of degree 7, dimension 1. In both of these sequences, I have just given the subscript (eg., I wrote 1,2 instead of $a_{1,2}$). On the left of the =, I have used " to indicate when an evaluation is used from the computation of the previous set of indices. To the right of the = the full index is given.

The problem with using the idea in the Recursive Composition Algorithm for basis conversion is that the Recursive Composition Algorithm evaluates at an index a_i some number of times, and recursively evaluates at the remaining

```
&Improved(0,$n,0,0);      # $k is the dimension

@M = ();                  # Mark the dimensions we use
sub Improved {
      local($m,$prevm,$previ,$ni)=@_;
      # m - number of indices selected so far
      # prevm - the number of previous indices selected
      # previ - the value of the previous index
      # ni - the number of indices selected
      local($i,$j,$nabove,$nbelow,$e);
      if ( $m >= $n ) { return; }

      $nbelow = 0;
      $nabove = $k-$ni;    # Really, this is k+1 -ni -1

      # Iterate over all dimensions
      for ( $i=0; $i<=$k; $i++ ) {
            # Skip over an already used dimension
            if ( $M[$i] ) { next; }
            $M[$i] = 1;
            # If the dimension less than the previous one
            # we can use at most one fewer of it's knots
            if ( $i < $previ ) { $e = 1;
            } else { $e = 0; }
            if ($m + ($prevm-$e)+ $nbelow*($prevm-$e-1)+
                $nabove*($prevm-$e) < $n ) {
                  $M[$i] = 0; $nbelow++; $nabove--;
                  next;
            }
            for ( $j=0; $j<$prevm-$e; $j++) {
                  if ( $m+$j >= $n ) { last; }
                  &PrintI($i,$j,$m+$j);
                  # We need a minimum number of
                  #  evaluations at this index.
                  if ( $nbelow*$j + $nabove*($j+1) +
                       $j+1 + $m < $n ) { next; }
                  &Improved($m+$j+1, $j+1, $i, $ni+1);
            }
            $M[$i] = 0; $nbelow++; $nabove--;
      }
}
```

Fig. 14. Recursive basis conversion algorithm generalized to arbitrary degree and dimension.

indices. However, the Barry-Goldman algorithm showed that we need to

Sequence of hyper-indices produced by the code of Figure 14:

```
0,0 0,1 0,2 0,3 1,0 1,1 1,2  =  0,0 0,1 0,2 0,3 1,0 1,1 1,2
 "   "   "   "  0,4 1,0 1,1  =  0,0 0,1 0,2 0,3 0,4 1,0 1,1
 "   "   "   "   "  0,5 1,0  =  0,0 0,1 0,2 0,3 0,4 0,5 1,0
 "   "   "   "   "   "  0,6  =  0,0 0,1 0,2 0,3 0,4 0,5 0,6
1,0 1,1 1,2 1,3 0,0 0,1 0,2  =  1,0 1,1 1,2 1,3 0,0 0,1 0,2
 "   "   "   "  1,4 0,0 0,1  =  1,0 1,1 1,2 1,3 1,4 0,0 0,1
 "   "   "   "   "  1,5 0,0  =  1,0 1,1 1,2 1,3 1,4 1,5 0,0
 "   "   "   "   "   "  1,6  =  1,0 1,1 1,2 1,3 1,4 1,5 1,6
```

Sequence of hyper-indices produced by the Barry-Goldman algorithm (Figure 12):

```
0,0 0,1 0,2 0,3 0,4 0,5 0,6   = 0,0 0,1 0,2 0,3 0,4 0,5 0,6
 "   "   "   "   "   "  1,0   = 0,0 0,1 0,2 0,3 0,4 0,5 1,0
 "   "   "   "  1,0 1,1 0,4   = 0,0 0,1 0,2 0,3 1,0 1,1 0,4
 "   "   "   "   "   "  1,2   = 0,0 0,1 0,2 0,3 1,0 1,1 1,2
1,0 1,1 1,2 1,3 0,0 0,1 0,2   = 1,0 1,1 1,2 1,3 0,0 0,1 0,2
 "   "   "   "   "   "  1,4   = 1,0 1,1 1,2 1,3 0,0 0,1 1,4
 "   "   "   "  1,4 1,5 0,0   = 1,0 1,1 1,2 1,3 1,4 1,5 0,0
 "   "   "   "   "   "  1,6   = 1,0 1,1 1,2 1,3 1,4 1,5 1,6
```

Fig. 15. Sequences of hyper-indices of dimension 1, degree 7.

repeat indices to get an optimal evaluation. The hard part is to control this repetition. The remainder of this appendix will discuss some issues and ideas related to this problem, giving a general algorithm that generalizes the Barry-Goldman algorithm, and is more efficient than the one in Figure 14. However, although it is unknown whether or not this generalization is optimal.

The Barry-Goldman algorithm evaluates the blossom at just over half the indices with one set of knots, and then recursively evaluates at the remaining arguments. The difficulty in generalizing this idea is that when our domain is of dimension greater than 1, we will need to evaluate the blossom at sets of arguments where each knot set comprises fewer than half the arguments. For example, with a domain of dimension 2, and degree 3, we need to compute $f(a_{0,0}, a_{1,0}, a_{2,0})$.

We can determine, however, the minimum number of times that some knot set will be evaluated at (eg., for dimension 2, degree 4, one of our knots set must be used as at least 2 of the arguments). Unfortunately, we cannot simply use this minimum to directly extend the Barry-Goldman algorithm. I.e., if we evaluate from one knot set the minimum number, and then recursively compute the rest of the evaluations, then we run the risk of computing the same blossom value twice. For example, with a domain of dimension 2, and degree 4, we need to evaluate at one knot set at least twice. If we evaluate f at $a_{0,0}$ and $a_{0,1}$, and then recursively compute the remaining arguments without remembering these initial evaluations, we'll evaluate at $f(a_{0,0}, a_{0,1}, a_{0,2}, a_{1,0})$

and $f(a_{0,0}, a_{0,1}, a_{1,0}, a_{0,2})$ (among a variety of repeated evaluations). If we try to avoid this duplication by only evaluating at each knot set once, then we produce the algorithm of Figure 14.

The approach I took was a brute force combination of the algorithms of Figures 13 and 14. When evaluating at a knot set, we consider two cases separately. If we evaluate over half the remaining arguments at one knot set, then we make a recursive call similar to the Barry-Goldman algorithm. If we evaluate at no more than half the remaining arguments, then we make a recursive call similar to that of the algorithm in Figure 14. I.e., in this second case, in the recursive call, we will make no more evaluations at the knot set with which we just evaluated at, and we limit the number of times at which we can use any other knot set to perform the remaining evaluations. The former case allows us to exploit the savings of Barry and Goldman, while the latter case avoids multiple evaluations. Some care must be taken in the latter case when determining the limit on the number of evaluations.

Code for this algorithm appears in Figure 16. Note that the code for `ComputeMin` is missing. This min is tricky to compute, as it changes based on the current knot set we are working with. Instead, I computed a simple min (Figure 17), and used the test on lines 29 and 31 to adjust this minimum. However, potentially this min will be calculated too low, and cause the algorithm to perform unnecessary partial evaluations.

A few notes on the algorithm of Figure 16. While it performs fewer evaluations than the one in Figure 14, this improvement first appears for domains of dimension 2. For such domains, the lowest degree at which this algorithm is better occurs at degree 7. The savings acrued is small. Further, it is unclear if the algorithm of Figure 16 is optimal. Potentially, another ordering of the hyper-indices will require even fewer evaluations (assuming that we require an algorithm that has only n layers of evaluations between the initial blossom values and the computed blossom values; otherwise, Goldman's algorithm, which has $(k+1)n$ layers but requires only $k+1$ complete de Casteljau computations, is more efficient).

Acknowledgments. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Barry, P. J. and R. N. Goldman, Knot insertion algorithms, in *Knot Insertion and Deletion Algorithms for B-spline Modeling*, R. N. Goldman and T. Lyche (eds), SIAM, Philadelphia, 89–133.
2. DeRose, T, R. Goldman, H. Hagen, and S. Mann, Functional composition algorithms via blossoming, ACM Trans. on Graphics **12** (1993), 113–135.
3. Farin, G., *Curves and Surfaces for Computer Aided Geometric Design*, Third Edition, Academic Press, NY, 1992.
4. Liu, W, Programming support for blossoming: The Blossom Classes, dissertation, Univ. Waterloo, Waterloo, 1996.

```
&Improved(0,$n,-1,0);
@M = ();      # Mark the dimensions we use
@NM =();      # Count how many of each dimension we use
sub Improved {
  local($m,$max,$previ,$ni)=@_;
  # m - number of indices selected so far
  # max - the max number of indices we can select
  # previ - the value of the previous index
  # ni - the number of indices eliminated
  local($i,$j,$min,$nabove,$nbelow);
  if ( $m >= $n ) { return; }
  $min = &ComputeMin($k-$ni,$n-$m,$previ,$max);
  $nbelow = 0;
  $nabove = $k-$ni;  # Really, this is k+1 -ni -1
  for ( $i=0; $i<=$k; $i++ ) {
    local($e);
    if ( $M[$i] ) { next; }
    if ( $i < $previ ) { $e = 1;
    } else { $e = 0; }
    if ($m + ($max-$e) +
        $nbelow*($max-$e-1) + $nabove*($max-$e) < $n ) {
      $M[$i] = 0; $nbelow++; $nabove--; next; }
    for ( $j=0; $j<$min-1; $j++ ) {
      &PrintI($i,$NM[$i]+$j,$m+$j); }
    for ( $j=$min-1; $j<$max-$e; $j++) {
      if ( $m+$j >= $n ) { last; }
      &PrintI($i,$NM[$i]+$j,$m+$j);
      # The following test is needed if min too small.
      if ( $nbelow*$j + $nabove*($j+1) + $j+1 + $m < $n){
        next; }
      if ( $j+1 > int(($n-$m)/2)  &&  $j+1 != $max) {
        $NM[$i] += $j+1;
        &Improved($m+$j+1, $j+1, $i, $ni);
        $NM[$i] -= $j+1;
        last;
      } else {
        $M[$i] = 1; $NM[$i] += $j+1;
        &Improved($m+$j+1, $j+1, $i, $ni+1);
        $NM[$i] -= $j+1; $M[$i] = 0;
      }
    }
    $nbelow++; $nabove--;
  }
}
```

Fig. 16. Generalized recursive basis conversion algorithm.

```
sub ComputeMin {
    local($nind, $deg, $previ, $max)=@_;
    local($min);
    # Compute an approximation to the min for now...
    $min = &Ceil($deg/($nind+1));
    if ( $min < 1 ) { $min = 1;}
    return $min;
}
```

Fig. 17. Generalized recursive basis conversion algorithm.

5. Liu, W., Mann, S., An optimal algorithm for expanding the composition of polynomials, submitted for publication, 1996, (12 pages).
6. Liu, W., Mann, S., Programming support for blossoming, in Proceedings of Graphics Interface '96, (Toronto), 1996, 95–106.
7. Mann, S. and W. Liu, An analysis of polynomial composition algorithms, University of Waterloo, Computer Science Dept. Report CS-95-24, 1995.
8. Ramshaw, L., Blossoming: a connect-the-dots approach to splines, Techn. Rep., Digital Systems Research Center, Palo Alto, 1987.
9. Sablonnière, P., Spline and Bézier polygons associated with a polynomial spline curve, Computer-Aided Design **10(4)** (1978), 257–261.
10. Seidel, H.-P., Symmetric recursive algorithms for surfaces: B-patches and the de Boor algorithm for polynomials, Constructive Approximation **7** (1991), 259–279.

Computer Science Department
University of Waterloo
200 University Ave W
Waterloo, Ontario N2L 3G1
Canada
`smann@cgl.uwaterloo.ca`