Efficient algorithms for factorization and join of blades

Fontijne, D.; Dorst, L.

DOI
10.1007/978-1-84996-108-0_21
Publication date
2010
Document Version
Submitted manuscript
Published in
Geometric algebra computing

Efficient Algorithms for Factorization and Join of Blades

Daniel Fontijne

Abstract Subspaces are powerful tools for modeling geometry. In geometric algebra, they are represented using blades and constructed using the outer product. To produce the actual geometrical intersection (Meet) and union (Join) of subspaces, rather than the simplified linearizations often used in Grassmann-Cayley algebra, requires efficient algorithms when blades are represented as a sum of basis blades. We present an efficient blade factorization algorithm, and use it to produce implementations of the Join which are are 5 to 10 times faster than earlier algorithms.

1 Introduction

In geometric algebra implementations, blades are commonly represented as weighted sums of orthogonal basis blades, which we call the *additive representation* [5]. This allows for straightforward and efficient implementation of many linear operations and products, but complicates implementing true subspace union (join) and intersection (meet), which are non-linear products [4]. If a factored (or *multiplicative*) blade representation were used, standard linear algebra techniques like Gram-Schmidt or the SVD an be used to implement the join efficiently and with ease [5]. But to implement the join of blades in additive representation, factorization of one of the input blades into vectors seems to be required.

In this paper, we present a new blade factorization algorithm and new algorithms for computing the join. The algorithms are improvements of previous work [1, 4, 7]. Our main contributions are the FastFactorization algorithm which factors blades by simply rearranging coordinates, and the StableFastJoin algorithm which computes the join in a numerically stable way at little additional cost compared to the FastJoin algorithm.

Daniel Fontijne
University of Amsterdam, Kruislaan 403, 1098 SJ, the Netherlands,
e-mail: fontijne@science.uva.nl

We do not consider computing the meet (either directly or simultaneously along with the join) as suggested by [7]. Experimentation [6] showed that adjusting our FastJoin algorithm to compute the meet directly is somewhat slower than computing it from the join using $\mathbf{A} \cap \mathbf{B} = (\mathbf{B} \rfloor (\mathbf{A} \cup \mathbf{B})^{-1}) \rfloor \mathbf{A}$ [4], and also leads to more generated code. We assume a Euclidean metric in all computations, as both factorization and join are metric independent operations [2]. Throughout the paper, n is the dimension of the vector space V^n and k is used to denote the grade of the blade in the current context.

2 New Algorithm for Blade Factorization

The problem of factorizing a blade is the following. Given a k-blade $\mathbf{B}$ (in the additive representation), find k vectors $\mathbf{b}_i$ such that $\mathbf{B} = \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \ldots \wedge \mathbf{b}_k$, where for reasons of numerical stability we prefer the factors $\mathbf{b}_i$ to be 'sufficiently non-parallel'.

To find factors, one may project 'probing vectors' $\mathbf{p}_i$ onto the blade. In geometric algebra, this is done by the projection operator $\mathbf{q}_i = (\mathbf{p}_i \rfloor \mathbf{B}^{-1}) \rfloor \mathbf{B}$ [3, 4]. If $\mathbf{q}_i$ is not zero, it is a factor of $\mathbf{B}$. By finding k linearly independent vectors $\mathbf{q}_i$ a factorization of $\mathbf{B}$ has been found (up to scale). A straightforward choice for the probing vectors $\mathbf{p}_i$ are basis vectors $\mathbf{e}_i$. To obtain the factorization one selects a total of k independent projected vectors $\mathbf{q}_i$. This is the essence of the outer factorization algorithm presented in detail in [4], based on ideas in [1], which we improve upon below.

If we are only looking for factors, we are more interested in the fact that the final result $\mathbf{q}_i$ is in $\mathbf{B}$ than in it being the precise projection of the original probing vector $\mathbf{p}_i$. But this 'being in $\mathbf{B}$' is already guaranteed by the second contraction in the projection equation. In our fast outer factorization algorithm, we save on the first contraction by using the orthogonal complement of the probing vector $\mathbf{p}_i$ with respect to the largest basis blade in $\mathbf{B}$. The whole operation then becomes merely the selection of appropriate coordinates. We turn this idea into an algorithm as follows:

Algorithm FastFactorization($\mathbf{B}$):

Let $\mathbf{B}$ be a k-blade, with $1 < k < n$ (to exclude trivial cases). The algorithm computes a factorization $\mathbf{B} = \beta\, \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \ldots \wedge \mathbf{b}_k$, where β is a scalar:

1. Find the basis blade $\mathbf{F} = \mathbf{f}_1 \wedge \mathbf{f}_2 \wedge \ldots \wedge \mathbf{f}_k$ to which the absolute largest coordinate of $\mathbf{B}$ refers. The $\mathbf{f}_i$ are basis vectors. Let β be the coordinate that refers to $\mathbf{F}$.
2. Compute $\mathbf{B}_s = \mathbf{B}/\beta$.
3. For each $\mathbf{f}_i$ compute: $\mathbf{b}_i = (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{B}_s$.

The independency of the vectors $\mathbf{b}_i$ guarantees that they form a factorization of $\mathbf{B}_s$.
Theorem: The factors $\mathbf{b}_i$ as computed by the FastFactorization algorithm are linearly independent.

Proof: $\mathbf{B}_s$ is represented as sum of grade k basis blades $\mathbf{E}_j$: $\mathbf{B}_s = \sum_{j=1}^{\binom{n}{k}} \frac{\beta_j}{\beta} \mathbf{E}_j$.

Through distributivity, each $\mathbf{b}_i = \sum_{j=1}^{\binom{n}{k}} \frac{\beta_j}{\beta} (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{E}_j$. Let us analyze the contribution of each $\mathbf{E}_j$ to each $\mathbf{b}_i$:

- If $\mathbf{E}_j$ is equal to $\mathbf{F}$, then $(\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{E}_j = \mathbf{f}_i$.
- Else if $\mathbf{E}_j$ is orthogonal to $\mathbf{F}$ we find $(\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{E}_j = 0$.
- Else $\mathbf{E}_j = \sigma_j (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \wedge \mathbf{e}_j$, for some basis vector $\mathbf{e}_j$, where the sign $\sigma_j = \pm 1$ depends on the order of basis vectors in $\mathbf{E}_j$. This $\mathbf{e}_j$ cannot be equal to $\mathbf{f}_i$, for then $\mathbf{E}_j$ is equal to $\mathbf{F}$, and also $\mathbf{e}_j$ can also not be any of the other basis vectors in $\mathbf{F}$, for then $\mathbf{E}_j$ would be 0 as it would contain the same basis vector twice. Thus in this case

$$(\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{E}_j = (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor (\sigma_j (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \wedge \mathbf{e}_j) = (\mathbf{f}_i \rfloor \mathbf{F}^{-1})^2 \, \sigma_j \, \mathbf{e}_j = \pm \sigma_j \mathbf{e}_j,$$

  where $\mathbf{e}_j$ is not a factor of $\mathbf{F}$.

Thus each $\mathbf{b}_i$ equals

$$\mathbf{b}_i = \mathbf{f}_i + \sum_{\mathbf{e}_j \wedge \mathbf{F} \neq 0} (\pm \sigma_j) \frac{\beta_j}{\beta} \mathbf{e}_j,$$

so that it does not contain any other factor of $\mathbf{F}$ than $\mathbf{f}_i$ itself. Since the $\mathbf{f}_i$ are linearly independent, so are the $\mathbf{b}_i$. □

Even though the factors are linearly independent, they are not orthogonal in general. For an estimation of how non-orthogonal the factors can be in the worst case, let us assume that the $\mathbf{B}$ is placed so skewly relative to the basis that it has an equal weight for each basis blade (it is not guaranteed that such an element $\mathbf{B}$ is indeed a blade). For a unit blade $\mathbf{B}$, this gives as worst case:

$$\mathbf{B} = \frac{1}{\sqrt{\binom{n}{k}}} \sum_{j=1}^{\binom{n}{k}} \pm \mathbf{E}_j, \quad \text{so} \quad \mathbf{B}_s = \sum_{j=1}^{\binom{n}{k}} \pm \mathbf{E}_j.$$

For such an element $\mathbf{B}_s$, the largest possible absolute value of an inner product between a pair of factors is the number of nonzero coordinates:

$$\|\mathbf{b}_i \cdot \mathbf{b}_j\| \leq n - k,$$

and the largest absolute value of an inner product between a pair of normalized factors is

$$\|\mathrm{unit}(\mathbf{b}_i) \cdot \mathrm{unit}(\mathbf{b}_j)\| \leq \frac{n-k}{n-k+1}.$$

This last equation shows that the factorization method could be less usable in for example 10-D space (e.g., if $k = 5, n = 10$, then $\|\mathrm{unit}(\mathbf{b}_i) \cdot \mathrm{unit}(\mathbf{b}_j)\| \leq \frac{5}{6}$) than it is in for example 3-D space, because the factors are potentially less and less orthogonal as the dimension of space n increases. But in such spaces a factorized blade representation becomes more attractive than the additive representation [5] and issues of efficient factorization of blades in the additive representation are then less crucial.

3 Algorithms for Computing the Join of Blades

Given the FastFactorization algorithm of the previous section, it is straightforward to formulate fast algorithms for the join based on [4, 7]. Given two blades $\mathbf{A}$ and $\mathbf{B}$, the following algorithm computes the join $\mathbf{J} = \mathbf{A} \cup \mathbf{B}$. A small constant threshold value ε is required.

Algorithm FastJoin$(\mathbf{A}, \mathbf{B}, \varepsilon)$:

1. Filter out trivial cases:
   - If $\mathbf{A}$ and/or $\mathbf{B}$ is zero, return $\mathbf{J} = 0$.
   - Else if $\mathbf{A}$ is of grade 0, return $\mathbf{J} = \text{unit}(\mathbf{B})$.
   - Else if $\mathbf{B}$ is of grade 0, return $\mathbf{J} = \text{unit}(\mathbf{A})$.
   - Else if $\mathbf{A}$ and/or $\mathbf{B}$ is of grade n, return $\mathbf{J} = \mathbf{I}_n$.
   - Otherwise continue with step 2.
2. If required, swap $\mathbf{A}$ and $\mathbf{B}$ such that $\text{grade}(\mathbf{A}) \geq \text{grade}(\mathbf{B})$. If the swap is applied, remember the 'swapping-sign' $\sigma = (-1)^{\text{grade}(\mathbf{A})\,\text{grade}(\mathbf{B})}$, otherwise set $\sigma = 1$. The swap is for reasons of efficiency.
3. Set $\mathbf{J} \leftarrow \text{unit}(\mathbf{A})$.
4. Find the largest basis blade term $\mathbf{F}$ in $\mathbf{B}$.
5. While $\text{grade}(\mathbf{J}) \neq n$ and not all basis vectors $\mathbf{f}_i$ in $\mathbf{F}$ have been tried:
   a. Take any basis vector $\mathbf{f}_i$ in $\mathbf{F}$ which has not been tried yet.
   b. Compute $\mathbf{b}_i = (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{B}$.
   c. Compute $\mathbf{H} = \mathbf{J} \wedge \text{unit}(\mathbf{b}_i)$.
   d. If $(\|\mathbf{H}\| \geq \varepsilon)$ set $\mathbf{J} \leftarrow \text{unit}(\mathbf{H})$.
6. Return $\sigma\,\mathbf{J}$.

The implementation of this algorithm can be made very efficient by generating optimized code for each combination of arguments in steps 5b and 5c, see Section 4.

3.1 Grade Stability and Numerical Stability

The main loop of the algorithm just tries to increase the grade of the current $\mathbf{J}$ by taking the outer product with factors of $\mathbf{B}$ in step 5c, and accepts the result if its norm is above some threshold value ε. This leads to a problem in stability of the grade of result: the factorization of $\mathbf{B}$ is dependent on the basis, and hence so is the grade of $\mathbf{J}$ as computed by the algorithm. In some application this may be acceptable, but in general one would prefer the grade of the computed join to be independent of the choice of basis. If we could somehow compute the required grade of the join $\mathbf{J}$ in advance, we could use this knowledge to guide the FastJoin algorithm. For this purpose, we use the following equation [4]:

$$\text{grade}(\mathbf{A} \cup \mathbf{B}) = \frac{\text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}) + \text{grade}(\mathbf{A} \Delta \mathbf{B})}{2}, \tag{1}$$

The delta product Δ (geometric symmetric difference) [2] can be computed as the highest grade part of the geometric product $\mathbf{A}\,\mathbf{B}$ which is nonzero. It can be determined efficiently through lazy evaluation [6]. Let us call the function which computes the grade of the delta product FastDeltaGrade$(\mathbf{A}, \mathbf{B}, \delta)$ where δ is a again a small threshold. We use it to improve our FastJoin algorithm as follows:

Algorithm StableFastJoin$(\mathbf{A}, \mathbf{B}, \varepsilon, \delta)$:

Start with steps 1-5 of FastJoin$(\mathbf{A}, \mathbf{B}, \varepsilon)$.

6. If $(\text{grade}(\mathbf{J}) = n)$ or $(\text{grade}(\mathbf{J}) = \text{grade}(\mathbf{A}) + \text{grade}(\mathbf{B}))$, return $\sigma\mathbf{J}$. (σ was computed in step 2 of the FastJoin algorithm.) Otherwise:
7. Compute grade$(\mathbf{A} \cup \mathbf{B})$ using Equation (1).
8. While $(\text{grade}(\mathbf{J}) < \text{grade}(\mathbf{A} \cup \mathbf{B}))$
   a. For all valid i, compute $\mathbf{b}_i = (\mathbf{f}_i \rfloor \mathbf{F}^{-1}) \rfloor \mathbf{B}$.
      Set $\mathbf{b}_m$ to that $\mathbf{b}_i$ which leads to the largest $\|\mathbf{J} \wedge \mathbf{b}_i\|$.
   b. Update $\mathbf{J} \leftarrow \mathbf{J} \wedge \mathbf{b}_m$.
9. Return $\sigma\mathbf{J}$.

Note that one should set $\varepsilon \geq \delta$, or else in step 7 blade $\mathbf{J}$ may already have a grade which is larger than the grade required by FastDeltaGrade$(\mathbf{A}, \mathbf{B}, \delta)$. Also note that in step 8a it makes no sense to try $\mathbf{b}_i$ which were already accepted in step 5d or in an earlier iteration of step 8a.

Another issue in the FastJoin algorithm is numerical stability due to the use floating point values which causes round off errors. The main thing we can do to ensure numerical stability of the StableFastJoin algorithm is to choose 'good' factors $\mathbf{b}_i$ in step 5c of the algorithms. Each factor $\mathbf{b}_i$ potentially extending the current iteration value of the join $\mathbf{J}$ can be written as a sum $\mathbf{b}_i = \mathbf{b}_i^{\parallel} + \mathbf{b}_i^{\perp}$ where $\mathbf{b}_i^{\parallel}$ is parallel to the current $\mathbf{J}$, and $\mathbf{b}_i^{\perp}$ is orthogonal to it. If $\|\mathbf{b}_i^{\parallel}\| \gg \|\mathbf{b}_i^{\perp}\|$ it is likely that the floating point precision of the outer product $\mathbf{J} \wedge \mathbf{b}_i$ is low. Hence we would prefer to select those $\mathbf{b}_i$ which result in the largest $\|\mathbf{J} \wedge \mathbf{b}_i\|$ because that $\mathbf{b}_i$ is most orthogonal to the current $\mathbf{J}$.

One solution is trying every remaining $\mathbf{b}_i$ in every loop and use the one which results in the largest norm, but this is inefficient. Fortunately, it is trivial to adjust the StableFastJoin algorithm to be more precise, with only a minimal performance impact by running step 1 through 5 (its FastJoin part) using a relatively large threshold like $\varepsilon = 10^{-2}$. This means that in step 5d,we only accept factors which are reasonably orthogonal to $\mathbf{J}$. When the input blades are in a non-degenerate configuration, the computed $\mathbf{J}$ will have the required grade (as verified by step 6 or 7). Otherwise, we find the best factors in step 8. This step is more expensive, but it rarely needs to execute.

4 Implementation

To implement the FastFactorization and the FastJoin algorithms we have written a code generator on top of the `Gaigen 2` code generation framework [5]. The code generator generates a `C++` implementation of the algorithms for a specific n and for a specific order and orientation of the basis elements. We used a code generator because our implementation approach leads to relatively large amounts of code which is rather tedious and error prone to write by hand.

Implementation of the Fast Factorization Algorithm

The essential optimization of our implementation of the FastFactorization algorithm is to generate a function for each possible largest basis blade $\mathbf{F}$ of the input blade $\mathbf{B}$. These functions implement the actual factorization and they are called via a lookup table. Figure 1 shows an example of such a function. It is clearly visible that the function just copies (and possibly negates) coordinates of the input blade to the coordinates of a factor. In the order of $\sum_k \binom{n}{k} = 2^n$ of these functions are generated, each with code size proportional to $k\,n$, for a total code size of $O(\sum_k nk\binom{n}{k}) = O(n^2\,2^{n-1})$.

In total, the implementation of the FastFactorization algorithm amounts to filtering out special cases, finding largest basis blade $\mathbf{F}$, and jumping to the corresponding factorization function via the lookup table.

```
void factorE234grade3(const float *B, float **b) {
  b[2][0] = B[0]; b[1][0] = -B[1]; b[0][0] = B[2];
  b[0][1] = b[1][2] = b[2][3] = B[3];
  b[2][4] = B[6]; b[1][4] = -B[8]; b[0][4] = B[9];
  b[0][2] = b[0][3] = b[1][1] = b[1][3] = b[2][1] = b[2][2] = 0.0f;
}
```

Fig. 1 *Example of a generated factorization function which computes the factors* $\mathbf{b}_i$ *of a normalized blade* $\mathbf{B}$. *This function implements the core of the* FastFactorization *algorithm for* $n = 5$, $k = 3$ *and* $\mathbf{F} = \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4$.

Implementation of the Join Algorithms

The generated implementation of the join algorithms closely follows their description in Section 3. The most significant optimization is the main loop of the algorithms: the implementation combines the factorization of step 5b with the outer product of step 5c. This allows it to take advantage of the zero coordinates which are in the factors due to the FastFactorization algorithm. One factor-and-outer-product function is generated for each valid combination of $\mathbf{f}_i \rfloor \mathbf{F}$ and grade($\mathbf{J}$) in steps 5b,c of the algorithm. These functions are again called via a lookup table. There are in the order of $O(\sum_j \sum_k \binom{n}{k}) = O(n\,2^n)$ of these functions, each with code size proportional to $n\binom{n}{j}$, for a total code size of $O(\sum_j \sum_k \binom{n}{k} n\binom{n}{j}) = O(n2^{2n})$.

4.1 Benchmarks

We performed our benchmarks on a 1.83 GHz Core2 Duo notebook, using a single thread (i.e., one CPU). The programs were compiled with Visual C++ 2005, using standard optimization settings. 32 bit floating point arithmetic was used.

We ran benchmarks for $3 \leq n \leq 6$. Above 6-D, our particular implementation starts to make less sense because the amount of generated code becomes too large (one may then switch to a conventional hand-written implementation which will be somewhat less efficient, see the discussion in Section 5). Besides giving absolute values, such as the number of factorizations that can be performed per second, we also list benchmarks relative to outer product of vector and 2-blade. This gives a fair impression of how expensive a factorization is relative to a straightforward bilinear product.

Factorization Benchmarks

To benchmark the FastFactorization$(\mathbf{A},\mathbf{B})$ algorithm we generated a number of random blades of random grades. The grades of the blades were uniformly distributed over the range [0, n]. A random k-blade was generated by computing the outer product of k random vectors. A random vector was generated by setting the n coordinates of the vector to random values, uniformly distributed in the range $[-1,1]$. The following table shows the results relative to the outer product and in absolute values (M = million).

n	3	4	5	6
relative to outer product	5.1×	5.1×	3.4×	3.8×
factorizations per second	15M	9.2M	5.2M	2.8M

Join Benchmarks

To benchmark the FastJoin and StableFastJoin algorithms we generated pairs of random blades $\mathbf{A}$ and $\mathbf{B}$ using the same method as described for the factorization benchmark. The grades of the blades $\mathbf{A}$ and $\mathbf{B}$ were uniformly distributed over the range [0, n]. However, the pairs of random blades were generated such that they shared a common factor. The grade of the common factor was uniformly distributed over the range [0, $\min\{\text{grade}(\mathbf{A}),\text{grade}(\mathbf{B})\}$].

We benchmarked both the FastJoin algorithm ($\varepsilon = 10^{-6}$) and the StableFastJoin algorithm ($\varepsilon = 10^{-2}$, $\delta = 10^{-6}$). The table below shows both absolute and relative (to the outer product) benchmarks. We also show relative figures for computing the Gram-Schmidt orthogonalization of the factors of each pair of random blades. For this we retain the factors that generated the blades, perform a standard Gram-Schmidt orthogonalization, and discard the dependent factors (using the same ε threshold as for the join). This algorithm was implemented using the same principles (i.e., optimizing and unrolling the inner loop of the algorithm) as the FastJoin algorithm. Hence the last row should give an impression of how expensive the FastJoin algorithms are compared to a classical linear algebra approach for computing a minimal basis set which spans a subspace union.

n	3	4	5	6
FastJoin (relative)	9.8×	8.7×	5.8×	6.4×
FastJoin (absolute)	7.4M	5.4M	3.1M	1.8M
StableFastJoin (relative)	9.8×	9.1×	7.0×	6.8×
StableFastJoin (absolute)	7.4M	5.2M	2.6M	1.6M
Gram-Schmidt (relative)	12×	12×	7.9×	8.0×

Code Size

The table below lists the size of the generated code for our factorization and join implementation. The code size grows in approximate agreement with the theoretical complexities of $O(n^2 2^{n-1})$ and $O(n2^{2n})$, respectively. We state 'approximate' because for low dimensional spaces the constant code size of the algorithm (which is included in the figures) can be relatively large compared to the amount of generated code, especially for the factorization algorithm.

n	3	4	5	6	7
FastFactorization	3.74kB	5.85kB	11.4kB	25.8kB	62.1kB
FastJoin	14.7kB	26.4kB	75.9kB	321kB	1.47MB

5 Discussion

FastFactorization Algorithm

Our benchmarks show that the FastFactorization algorithm is in the order of 5 times slower than a regular outer product in the same space. The time complexity of the FastFactorization algorithm is $O(\binom{n}{n/2}) = O(n^{-1/2}2^n)$ (using the Stirling approximation of factorials) due to the step which finds the largest coordinate of the input blade. The fact that this step uses conditional statements makes it extra expensive on modern pipelined processors. The outer product of a vector and a 2-blade relative to which we presented the benchmarks has a time complexity of $O(n^3)$, and uses no conditional statements (an outer product of arbitrary blades has a time complexity around $O(2^n)$). The benchmarks suggest that the FastFactorization algorithm becomes less expensive compared to the outer product as the n becomes larger, but if one plots $\binom{n}{n/2}/n^3$ for $1 \leq n \leq 20$ it becomes clear that $n = 6$ is in fact the turning point beyond which the FastFactorization should become exceedingly expensive relative to the outer product. So our figure of five times slower is only valid for the limited range of n for which we benchmarked.

It is rather remarkable (but understandable) that in general the FastFactorization algorithm does not use all coordinates of the input blade once it has found which coordinate is the largest one: the k factors of a k-blade have kn coordinates, which in many cases is less than the $\binom{n}{k}$ coordinates of the blade in additive representation.

The code size of the generated implementation is acceptable (less than 100kB) up to 7-D, but extrapolation of the figures suggests that a 10-D implementation would about 1MB in size. This is confirmed by the theoretical figure that code size should

be in the order of $O(n^2\,2^{n-1})$. Thus in high-dimensional spaces we recommend using a more conventional implementation approach (our initial implementation of the FastFactorization algorithm was implemented without using code generation and was about two times slower than the generated implementation).

The FastFactorization algorithm is a useful building block for other algorithms. In this paper we used it for computing the join. Another useful application may be a fast 'blade manifold projection' function which projects a non-blade onto the blade manifold in Grassmann space (the elements satisfying the Plücker relations). This may be implemented by naively 'factoring' the non-blade, and using the factors thus obtained to compute a valid blade as their outer product.

Fast Join Algorithm

Our benchmarks show that our implementation of the FastJoin algorithms is slightly faster than an implementation of Gram-Schmidt orthogonalization applied to the factors of the input blades. This is quite remarkable, as it means that – even if the only geometry you need is computing the join – you may be better off using the basis-of-blades representation rather than a factorized representation in terms of basis sets (at least for such low-dimensional spaces).

To make sure the grade of the join which is computed by our FastJoin algorithm is independent of the (arbitrary) basis, use of the delta product is required, invoking some additional computational cost. However, the delta product needs to be invoked only when the algorithm cannot determine that it has computed a join of the right grade. As a result, the cost of the StableFastJoin (which uses the delta product) is only about 10% higher than that of the straightforward FastJoin algorithm.

The time complexity of the FastJoin algorithms is $O(n^2\binom{n}{n/2}) = O(n^{3/2}2^n)$, as we need to compute in the order of n outer product of vectors with blades (in step 5c), and each of these outer products has a time complexity of order $n\binom{n}{n/2}$. This means that the cost FastJoin algorithm relative to a vector-2-blade outer product should increase right from $n = 3$. The fact that the benchmarks do not entirely agree with this is likely due to the decreasing relative cost of the overhead (filtering out special cases, and such) as n increases.

We implemented our join algorithms using code generation. Starting around 7-D, this no longer tractable. The generated code for 6-D is 0.32MB, while the code for 7-D is 1.47MB in size; generating and compiling the 7-D code took several minutes. The size of the code is in the order of $n2^{2n}$. Hence for $n \geq 7$ we recommend using a more conventional implementation which does not explicitly spell out the functions used in the inner loop for all possible arguments.

6 Conclusion

We have shown that the outer factorization of a k-blade in V^n that is represented as a sum of basis k-blades is (computationally) a trivial operation. It amounts to copying

and possibly negating selected coordinates of the input blade into the appropriate elements of the factors. Implemented as such and using code generation, factorization is only about five times slower than an outer product in the same algebra in the low-dimensional spaces. The $O(n^{-1/2}2^n)$ time complexity of the factorization algorithm is determined by the number of coordinates of the input k-blade, which becomes exceedingly large in high-dimensional spaces.

The join and meet of blades are relatively expensive products, due to their non-linearity. However, when efficiently implemented through our FastJoin algorithm, the cost of the join and meet is only in the order of 10 times that of an outer product in the same algebra, compared to 100 times in previous research [4]. Again, these figures are valid only for low dimensional spaces. The $O(n^{3/2}2^n)$ time complexity makes clear that in high-dimensional spaces, one should use a multiplicative presentation of blades [5] and use classic linear algebra algorithms like the SVD (which has $O(n^3)$ time complexity) to implement the join. Our StableFastJoin algorithm, which takes grade stability and numerical stability into account, is just 10% slower than the FastJoin algorithm.

These speeds are obtained at the expense of generating efficient code that spells out the operations for certain combinations of the basis blades and grades in the arguments. While efficient, this is only truly possible for rather low-dimensional spaces, since the amount of code scales as $O(n^2 2^{n-1})$ for FastFactorization and $O(n2^{2n})$ for the FastJoin algorithm.

References

1. Bouma, T: Projection and Factorization in Geometric Algebra. Unpublished paper. (2001)
2. Bouma, T and Dorst, L and Pijls, H: Geometric Algebra for Subspace Operations. Acta Mathematicae Applicandae. **73**, 285–300 (2002)
3. Dorst, L.: The Inner Products of Geometric Algebra. In: Dorst, L. , Doran, C., Lasenby, J. (eds.) Applications of Geometric Algebra in Computer Science and Engineering, pp. 35-46. Birkhäuser, Boston (2002)
4. Dorst, L. and Fontijne, D. and Mann, S.: Geometric Algebra for Computer Science: An Object Oriented Approach to Geometry. Morgan Kaufmann, San Francisco (2007)
5. Fontijne, D. Efficient Implementation of Geometric Algebra (PhD. thesis). University of Amsterdam (2007).
6. Fontijne, D and Dorst, L.: Efficient Algorithm for Meet and Join of Subspaces. Submitted to International Journal of Computational Geometry and Applications (2007).
7. Bell, I: private communication. 2004-2005.