Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Another Multivector Inverse Algorithm

David Arthur Jones

dajonesy@gmail.com

Research Article

Keywords:

Posted Date: November 12th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11536816/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Another Multivector Inverse Algorithm

David Arthur Jones

Abstract. A numerical algorithm is described for computing the inverse of a non-singular multivector across dimensions. The method wraps a reversion-like product around a Faddeev–LeVerrier–Souriau (FLS) polynomial core. While FLS provides dimensional extensibility, the wrapper reduces the polynomial order of the FLS method, and introduces a sparse multivector which can be further exploited. The construction and essential properties of the initial involution are explained.

For dimensions 4–7, the optimized inverse requires fewer real multiplications than a same-dimension multivector product. Dimensions 8–11, follow easily on common hardware with similar efficiency. For dimensions 12–15, higher-precision arithmetic demonstrates the generality of the algorithm when the arithmetic can support it. Performance on common hardware is reported. Directions are included for obtaining a Python package with examples for numerical computation in Clifford algebra, supporting arbitrary signatures including degenerate cases.

Mathematics Subject Classification (2010). Primary 15A66; Secondary 65F05.

Keywords. Clifford algebra, Faddeev–LeVerrier–Souriau, Geometric algebra, Involution, Multivector inverse.

Introduction

Clifford algebra practitioners operating within a few dimensions will be familiar with multivector inverse formulas written in terms of common involutions following graded patterns. In 2018 Acus and Dargys [3] extended such formulas to 6 dimensional multivectors. Determinative formulas like these are useful, but bound to a specific dimension. The following year, Hitzer and Sangwine introduced a method for extending any even dimensional inverse algorithm to the next odd dimension [6], thus supporting an inverse method for dimension 7. Lately, Abdulkhaev and Shirokov [1] [2] revealed an algorithm for dimension 7 that cleverly folds the compound data types described by Hitzer into the

involution construction. This innovation relieves programmers from formal data-type conversions, leaving them grounded in real arrays.

After some vigorous algebraic manipulation of a 6D Acus and Dargys involution formula into polynomial form, this author this year published another 6D multivector inverse algorithm [7]. One advantage of the polynomial form was two fewer multivector products required to reach the determinant. The algorithm possessed a beautiful symmetry and simplicity which turned out to have a reason: the coefficients are found within a polynomial numerical method called Faddeev–LeVerrier–Souriau (FLS). The author excitedly reached for an 8D inverse algorithm using the next level of FLS coefficients only to be disappointed. It turned out, that in order to support the FLS core, the initial involution (reversion), had to be modified.

Working through the details of converting the Abdulkhaev 7D algorithm to polynomial form, the odd step involution innovation that was needed began to show: negate the reversion for the 7D extension. In higher odd dimensions that formula became, more precisely: append to the block negation pattern its negative. Experimentation uncovered the pattern for the even steps as simply doubling the block size from the previous odd pattern.

With these insights revealed, a general algorithm across dimensions took form. In the sections to follow the algorithm is explained. What can be achieved with freely available open-source software, on inexpensive commercial hardware, is documented, and made available online.

On many occasions, we will reference the pygae (Python Geometric Algebra Enthusiasts) team and their Python package `clifford` [5]. We will apply the "pygae" label to results obtained from their software.

1. Algorithm Description

1.1. The Envelope

Most of the involution formulas for the inverse of multivector A given by Acus and Dargys begin by computing: $H = A\widetilde{A}$. It follows that $A^{-1} = \widetilde{A}H^{-1}$ provided H is invertible. An inverse for A therefore follows from an inverse for H.

1.2. Equivalent Expressions in Six Dimensions

Acus and Dargys give the determinant of a 6D multivector [4] as

$$\det(A) = \frac{1}{3}\left(HH\,\overline{HH} + 2H\,\overline{\overline{H}\,\overline{\overline{H}\,\overline{H}}}\right), \qquad \text{where } \overline{X} = 2\langle X\rangle_0 - X.$$

The bar involution has the effect of negating all but the scalar term in the multivector. 5 multivector products are required to realize this formula.

The polynomial form, written with a scalar-tweak operator, requires only 3 multivector products:

$$\det(A) = -\frac{1}{3}\big(((((H \circ 3)H) \circ 1)H) \circ \tfrac{1}{3}\big)H, \qquad \text{where } X \circ s = X - (1+s)\langle X\rangle_0.$$

Listing 1. Signs for the Tau Involution at Dimensions 4-11

```
invo_4   = [ 1]*16
invo_5   = [ 1]*32
invo_6   = [ 1]*64
invo_7   = [ 1]*64  + [-1]*64
invo_8   = [ 1]*128 + [-1]*128
invo_9   = [ 1]*128 + [-1]*128 + [-1]*128 + [ 1]*128
invo_10  = [ 1]*256 + [-1]*256 + [-1]*256 + [ 1]*256
invo_11  = ([ 1]*256 + [-1]*256 + [-1]*256 + [ 1]*256
          + [-1]*256 + [ 1]*256 + [ 1]*256 + [-1]*256)
```

This is a main result of the author's prior paper [7]. The scalar-tweak operator has the effect of multiplying the scalar term of the multivector by $-s$. Notice that $\overline{X} = -X \circ 1$.

Provided that the determinant is different than zero, either of these routes to the determinant offers an expression:

$$\det(A) = SH = HS, \text{ so that } H^{-1} = \frac{S}{\det(A)}$$

With H^{-1} we can complete $A^{-1} = \widetilde{A}H^{-1}$.

1.3. Modifying Reversion

Determining an alternative involution to reversion to expand FLS processing to higher dimensions is a main result of the present communication. The pattern described in Listing 1 is laid over reversion (by direct product) to determine which multivector components flip sign. This combination is being called the τ-involution, and is a function of dimension. For dimensions less than seven, reversion is unmodified. Each odd step, beginning at $d = 7$, appends the negative of the previous even step; each subsequent even step doubles the uniform block size of each block of the previous odd step.

It is important to realize that the order of components in this implementation is *ordinal*. In this order the basis-blades have a fixed location, so that the bits of an address correspond to vector-bases. For instance: the coefficient for element e_{356} will be found at `Reg[0b110100]` in a translation that treats e_1 as the first vector-basis. Without this statement, the specification of an involution by an array of signs is left ambiguous.

The block sign pattern is the Thue–Morse sequence: better known in the device of a *parity bit* appended to binary integers, only here converted to a sign under the map: $\{\oplus : 0, 1\} \mapsto \{\times : 1, -1\}$. This pattern is perhaps more appropriately described as the *grade involution* pattern which isolates the bivector (even) algebra. While grade involution is the direct product of reversion and Clifford conjugation, what it means to apply the grade involution only to the upper range of bits, in combination with reversion, is uncertain.

Table 1. Recursion order, native FLS against τ-FLS.

order	4D	5D	6D	7D	8D	9D	10D	11D
Native FLS	2^2 (4)	2^3 (8)	2^3 (8)	2^4 (16)	2^4 (16)	2^5 (32)	2^5 (32)	2^6 (64)
τ-FLS	2^1 (2)	2^1 (2)	2^2 (4)	2^2 (4)	2^3 (8)	2^3 (8)	2^4 (16)	2^4 (16)

There may be a better way to describe this involution: a subject for further investigation.

As a final remark about the τ-involution, it is strange to discover that it is **not** a graded involution, though its properties are similar to the familiar anti-involutions.

1.4. Faddeev–LeVerrier–Souriau and its Tweaks

A general method to reveal the eigen-structure of a multivector can be turned to the inverse problem, as described by Shirokov in theorem 4 of [13] and implemented in the publicly available Python package `clifford` [5]. This polynomial method, known as Faddeev–LeVerrier–Souriau (FLS), is beautifully simple but suffers from a rapid loss of precision as the order of the polynomial doubles every other additional dimension. For this reason, the pygae team makes the method available but does not systematically employ it.

The tweak coefficients of the FLS recursion at order 2^n are generally described by the formula:

$$FLS(n) = \left[\frac{2^n-1}{1},\ \frac{2^n-2}{2},\ \ldots,\ \frac{2^{n-1}}{2^{n-1}},\ \ldots,\ \frac{2}{2^n-2},\ \frac{1}{2^n-1}\right]$$

which for the first four orders gives

$$\begin{aligned}
FLS(1) &= [\ 1\ ]\\
FLS(2) &= [\ 3,\ 1,\ \tfrac{1}{3}\ ]\\
FLS(3) &= [\ 7,\ 3,\ \tfrac{5}{3},\ 1,\ \tfrac{3}{5},\ \tfrac{1}{3},\ \tfrac{1}{7}\ ]\\
FLS(4) &= [\ 15,\ 7,\ \tfrac{13}{3},\ 3,\ \tfrac{11}{5},\ \tfrac{5}{3},\ \tfrac{9}{7},\ 1,\ \tfrac{7}{9},\ \tfrac{3}{5},\ \tfrac{5}{11},\ \tfrac{1}{3},\ \tfrac{3}{13},\ \tfrac{1}{7},\ \tfrac{1}{15}\ ]
\end{aligned}$$

While FLS can be directly applied to the source multivector the advantage here is to the reduction in FLS order: half the order for even dimensions and one fourth the order for odd dimensions. The reduction in order extends the range in dimension before loss of precision foils the practical usefulness of the algorithm. See Table 1.

Notice that odd dimensions process at the same order as the **next** even dimension with native FLS, but at the same order as the **previous** even dimension with τ-FLS. Overall, substituting τ-FLS shifts the effort down by three dimensions.

The shift in FLS processing order is an immediate advantage, paid with the cost of two sparse multivector multiplies. With a little programming effort,

exploiting the repetitive sparse multiplications which remain in domain over an FLS sweep realizes a dramatic reduction in effort.

1.5. The τ-domain

Certain mathematical properties attributed to reversion are preserved in the τ-modification to reversion. These properties make the optimizations described later possible.

Let τ be a multivector involution $\tau(\tau(A)) = A$, described by component sign changes, which is also an anti-automorphism $\tau(AB) = \tau(B)\tau(A)$. Its +1 fixed set is called the τ-domain, and comprises about half the algebra. Although this domain is not closed under the geometric product, it has four properties that together confine the FLS recursion to it.

Entry. $C = A\,\tau(A)$ lies in the domain, since $\tau(C) = \tau(\tau(A))\,\tau(A) = A\,\tau(A) = C$.

Powers stay in domain. $\tau(X^n) = \tau(X^{n-1})\tau(X) = X^{n-1}X = X^n$.

Polynomials stay in domain. τ is linear, so real linear combinations of powers remain in the domain.

Commuting products stay in the domain. For X, Y in the τ-domain, $\tau(XY) = \tau(Y)\,\tau(X) = YX$. Hence XY lies in the τ-domain if and only if $XY = YX$.

Proof. Let $\mathcal{Z}$ denote the center of the algebra $\mathrm{Cl}(p, q)$, where $\mathcal{Z} = \mathbb{R}$ for even dimensions and $\mathcal{Z} = \mathbb{R} \oplus \mathbb{R}I$ for odd dimensions. Under the τ-construction, $\tau(Z) = Z$ for all $Z \in \mathcal{Z}$, placing the center entirely within the τ-domain. Each FLS iterate is formed by multiplying by C and shifting by a central element; hence, every iterate is a polynomial in C with coefficients in $\mathcal{Z}$.

Because central elements commute with all elements of the algebra and powers of C commute with each other ($C^iC^j = C^{i+j} = C^jC^i$), any two such polynomials $X = p(C)$ and $Y = q(C)$ commute: $XY = YX$. Consequently, the commutator product $X \times Y = \frac{1}{2}(XY - YX)$ vanishes, and the geometric product reduces to the symmetric product:

$$XY = \tfrac{1}{2}(XY + YX).$$

Since X and Y lie in the τ-domain and commute, $\tau(XY) = \tau(Y)\tau(X) = YX = XY$, proving that XY remains in the τ-domain. □

This property, guaranteeing that FLS iterations remain in domain is remarkable given that the τ-domain is not in general, closed under the geometric product. Since τ-domain products which stay in domain are commutative, we need record only one sign per qualifying τ-domain element pair; and those element pairs are easy to pick out: just those that commute.

1.6. The Model Algorithm in Python

To conclude the algorithm description, we attach Listing 2 modeling the algorithm in Python in terms of whole multivectors. Some explanations are necessary. The multivector `copy()` method instantiates an independent multivector of the same value. Otherwise, an aliasing action occurs with

Listing 2. Multivector Inverse Model (Python)

```
Involution[d-4]     # tau = reversion x Thue-Morse layer
Tweak[d-4]          # FLS coefficients, e.g. [3, 1, 1/3] at d = 6,7

def mv_inv(A):                      # 4 <= d <= 11
    d, odd = A.dimensions, A.dimensions & 1

    tA = Involution[d-4] * A        # tA = tau(A)
    C = A * tA                      # C = A tau(A)  -- lands in domain

    T = Tweak[d-4]                  # FLS sweep, all inside the domain
    D = C.copy()
    D.Reg[0] *= -T[0]
    if odd: D.Reg[-1] *= -T[0]
    for t in T[1:]:
        D *= C
        D.Reg[0] *= -t              # tweak the scalar ...
        if odd: D.Reg[-1] *= -t     # ... and the pseudoscalar
    E = D * C                       # collapses to the center

    if odd:                         # center is E = a + bI
        Ep = E.copy(); Ep.Reg[-1] *= -1
        rC = (1 / (Ep * E).Reg[0]) * Ep * D
    else:                           # center is a scalar only
        rC = (1 / E.Reg[0]) * D

    return tA * rC                   # 1/A = tau(A) / C
```

assignment `X=Y`. `Reg[]` is a multivector's coefficient array. `Reg[0]` selects the scalar coefficient, while `Reg[-1]` selects the last element of the array, the pseudoscalar coefficient.

2. Optimization

The product of two arbitrary d-dimensional multivectors generally requires 4^d real multiplies. However, no product in this algorithm actually requires that many, when structural particulars are exploited. Each optimization effectively avoids multiplying by zero, or producing zero when components are known to cancel.

2.1. Real-Product Counts

Let's grant that the number of real-products is an objective measure of effort for a numerical procedure. This gives us an objective comparison between the full-multivector unoptimized model shown in Listing 2, and an optimized version that reduces the number of real-products to just those required after

TABLE 2. Inputs to the real-product count formulas by dimension d

d	n	c	s	p
4	2	1	6	1
5	2	2	12	2
6	4	1	28	6
7	4	2	56	12
8	8	1	120	28
9	8	2	240	56
10	16	1	496	120
11	16	2	992	240
12	32	1	2,016	496
13	32	2	4,032	992
14	64	1	8,128	2,016
15	64	2	16,256	4,032

d = dimensions,
n = FLS order,
c = center size,
s = slots,
p = pairs.

eliminating multiplication by zero, or to a canceled product known in advance. It may be that sign-flipping gets translated to a real-product in execution, but we will not count these. Nor will we count the extra real-products introduced when substituting extended-precision products for double-precision products in dimensions 12-15.

First we describe the inputs to the real-product count formulas by dimension. Among these counts are the shapes of the data structures that govern FLS-processing. "Slots" is the size of the τ-domain, and "pairs" is the number of distinct τ-input combinations which produce each τ-component. The table of pairs `T[1:s][p]` is of the shape "slots by pairs". A sign table `S[1:s][p]` of the same shape indicates the sign associated with each pair. This sign table bears the effects of choice of signature. A non-degenerate signature generates a sign table containing {-1,1}. Zeros are introduced to the sign table for degenerate signatures. You can see the use of these tables in Python Listing 3.

We divide the algorithm into three sections: the step-down into the tau-domain, the FLS polynomial iterations, and the step-up from the tau-domain back to a full multivector. These are the formulas for the number of required real-products once the zeros in the sparse multivectors have been eliminated.

$$M_{\downarrow} = 2^d + (s-1)\,2^{d-1} = (s+1)\,2^{d-1} \tag{1}$$

$$M_{\tau} = (n-2)\big(c + cs + 2p(s-c)\big) + cs + 2(c-1) \tag{2}$$

$$M_{\uparrow} = s \cdot 2^d \tag{3}$$

2.2. The Envelope: Entering and Exiting the τ-domain

The *slots* are determined and listed in an array `I[]`. This is then the map for stepping into and out of the τ-domain. We are careful to ensure for odd

TABLE 3. Real multiplies per optimized inverse expressed in units of one arbitrary multivector product (4^d real multiplies), along side the model's multivector product count.

d	$M_\downarrow$	M_τ	$M_\uparrow$	total	model
4	0.22	0.02	0.38	0.62	3
5	0.20	0.03	0.38	0.60	5
6	0.23	0.18	0.44	0.84	5
7	0.22	0.18	0.44	0.84	7
8	0.24	0.62	0.47	1.33	9
9	0.24	0.62	0.47	1.33	11
10	0.24	1.59	0.48	2.32	17
11	0.24	1.59	0.48	2.32	19
12	0.25	3.58	0.49	4.32	33
13	0.25	3.58	0.49	4.32	35
14	0.25	7.57	0.50	8.31	65
15	0.25	7.57	0.50	8.31	67

dimensions that the formula `~I[n]==I[-1-n]` holds. Here the tilda represents the complement (trimmed to the given dimension); and negative indexing counts from the end of the array.

Because the components of the two multivectors in the product $C = A\tau(A)$ are redundant, the product of any two distinct components, a, b of A will either $= \pm 2ab$ or 0. Since multiplication by ± 2 does not require the full machinery of a real multiplication, we don't count it. We know that whenever the component result is not in domain the product is 0 so the products we require are easy to determine. This is the *step-down cost* described by equation 1.

The *step-up cost* in equation 3, is optimized simply by avoiding the zeros in the τ-domain C^{-1}, when computing $A^{-1} = \tau(A)C^{-1}$

2.3. The τFLS iteration

The τ-FLS iteration computes the polynomial which reduces to the center of the algebra. The multivector products in this section of processing are optimized to a table driven function, described in Python Listing 3. The table `T[][]` holds an array of `(a,b)` pairs of indecies into the compressed τ-multivector array which produce each non-central component of the product. The table `S[][]` holds one sign per pair (or possibly zero in the case of a degenerate signature) and has the same shape as `T[][]`. The construction of `S[][]` encodes the effects of the chosen signature of the algebra. The signs (or zero) associated with each square of the τ-components is tabled in `S0[]` for computing the scalar part of the τ-product. For odd dimensions a similar array of signs (or zero) associated with the complementary product is tabled in `SP[]`.

Listing 3. Tau Multiplication

```
# s rows of p index pairs tabled in T[]
# s by p sign table S[][]
# s signs in S0[] for tau-domain squares
def tau_mult( X, Y ): -> Z
    Z[0] = sum(S0[k] * X[k]*Y[k] for k in range(s))   # scalar
    if odd:
        Z[-1] = sum(SP[k] * X[k]*Y[-1-k] for k in range(s)) #
    pseudoscalar
    for r in range(1, s):                  # one row per tau-slot
        z = 0.0
        for k, (a, b) in enumerate(T[r]):
            z += S[r][k] * (X[a]*Y[b] + X[b]*Y[a])
        Z[r] = z                           # one sign, two terms
    return Z
```

A pair is recorded in the table only when $e_c = e_a e_b$ stays in the domain, which happens exactly when e_a and e_b commute. That is why the two terms of a pair share a single sign. The τ-product $Z = XY$ is then expressed mathematically for even dimensions:

$$Z_c = \sum_{(a,b)\in T_c} S_{c,ab}\big(X_a Y_b + X_b Y_a\big), \qquad Z_0 = \sum_{a\in I} S_{0,a} X_a Y_a.$$

For odd dimensions, the pseudoscalar component is computed:

$$Z_n = \sum_{a\in I} S_{n,a} X_a Y_{\bar{a}}.$$

Here n is the tau-index of the pseudoscalar and $\bar{a}$ is the tau-index of the blade complementary to a. The central components Z_0 and Z_n each take one product per slot, $|I|$ terms, whereas each non-central component Z_c takes two products per pair, $2|T_c|$ terms.

The last polynomial iteration is extracted from the loop because we want to special case that multivector product knowing that the result is a very sparse multivector containing only the *center*: a lone scalar when the dimension is even, or accompanied by the pseudoscalar when the dimension is odd.

Referring back to the model in Listing 2, we now describe optimization of the last FLS product, which is special cased because it collapses to the center: $E = DC$, from the model. So, the reciprocal we seek is just $1/C = D/E$ for even dimensions where E is just a scalar. When the center includes the pseudoscalar, a little extra processing is needed to reduce the center to a scalar we can divide by. For this we invent E' which is the two component multivector E with the pseudoscalar negated. Then we can compute the

TABLE 4. Stored even-dimension tables and the odd dimensions they derive.

stored table	$d = 4$	$d = 6$	$d = 8$	$d = 10$
slots	6	28	120	496
pairs per row	1	6	28	120
size	< 1 KB	1 KB	27 KB	481 KB
derives	$d = 5$	$d = 7$	$d = 9$	$d = 11$

scalar value: $E'E = E[0] * E[0] - I2 * E[-1] * E[-1]$ where $I^2 = \pm 1$ is the pseudoscalar squared. So, we now have for the odd case: $E'E = E'DC$, or $1/C = (E'D)/(E'E)$. Our real-product count for the last multivector product in the FLS sweep becomes: s for even dimensions, and $2+2s$ for odd dimensions (where s is the "slot count") . Or by formula: $cs + 2(c - 1)$. Where c is the size of the center.

Here is the full formula for the number of real-multiplications required in processing FLS in the τ-domain (reiterating equation 2):

$$M_\tau = (n - 2)(2p(s - c) + cs + c) + cs + 2(c - 1)$$

The formula in words: Each of the $n - 1$ τ-products except the last costs two real-products per pair ($2p$) for each of the non-central slots ($s - c$), whereas the central slots require one real-product per slot, finishing with one tweak per central slot. The last τ-product produces only the center, so it needs just the cs real-products landing in the center. Finally we include the 2 real-product cost to form $E'E$ when the center is composite.

2.4. Only Even Dimension Pair Tables are Stored

At odd d the pseudoscalar lies in the domain, so the domain factors:

$$(u_1 + w_1 I)(u_2 + w_2 I) = (u_1 u_2 + I^2 w_1 w_2) + (u_1 w_2 + w_1 u_2)I$$

with u and w in the even $(d - 1)$ domain. The signature of the top generator enters as a single bit, $I^2 = \pm 1$.

The whole family for $d = 4 \ldots 11$ occupies 509 KB.

No odd pair table need ever be built. However, the sign tables are another matter. Because of the vagaries of supporting degeneracy, the sign tables are expanded to the full dimensional requirement for degenerate signatures. Remember that there is a "complement" organizing device relating the second half of these sign tables to the first half, as expressed in the slot identifier array `I[]`.

3. Performance

The author developed and tested these algorithms in Python 3.12.13 using an Anaconda distribution on an Acer 515 Chromebook Plus. The AMD CPU features **4 physical cores with SMT enabled (8 logical threads)**, with a base frequency of 2.1 GHz and a dynamic boost frequency up to **3.7 GHz**. The processor supports **AVX2 and FMA3 instruction sets**, which are employed via

Numba's Just-In-Time (JIT) compiler to optimize high-dimensional multivector operations. The system uses **8GB of DDR4 RAM**, with **6.5 GiB** available to the virtualized Linux environment. The hardware supports a **64-byte cache line alignment** and **512 KB of L2 cache per core**.

As an interpreted language, Python's computing priorities emphasize interactive exploratory development at the expense of computational speed and efficiency. For numerical work, some of the efficiency of a compiled language can be recovered for numerical processes with declared data types (numpy), and a just-in-time compiler (numba). The benchmarks here must be understood in this context. For these reasons, and the intrusion of background processing, benchmarks listed here have to be understood as samplings from the range of expectations.

3.1. Benchmarks

Four inverse implementations are bench-marked for comparison. First the **model**, as described in abbreviated form in Listing 2, is timed. This represents the unoptimized τ-FLS algorithm coded in multivector operations. This method transports well to other platforms that support the geometric product. Next, is the **optimized τ-FLS** which eliminates induced zeros. The third method adds a **Newton-Schulz** refinement to as much as double the precision of the inverse when needed. This at least requires taking the product of the multivector and its computed inverse to assess the residual. The fourth method could be called **native FLS** which follows Shirokov's implementation, without first using the τ-involution.

Those four methods are supplemented with the independent pygae (Python Geometric Algebra Enthusiasts team) implementation. The inverse is computed from Hitzer's closed-form formulas for dimensions 4 and 5; and for dimensions without a closed-form formula, pygae relies upon the LAPACK linear algebra package [5] for an inverse.

Kingdon (v3.0.0) [12] was also considered. Its built-in inverse did not complete for a general multivector beyond $d = 4$, and FLS written with its product slows sharply from $d = 7$ onward, where Kingdon switches from generated code to direct computation. We omit Kingdon timings from the tables, concluding that dense high dimensional multivectors are outside its focus.

For these tests, Euclidean signature was chosen. Behavior under other signatures may vary. For instance, mixed or degenerate signatures may increase the possibility of ill-conditioning. We are considering these issues out of scope for this effort: though baseline regression testing does verify support for arbitrary signature, including degenerate algebras.

The benchmark used a "best of 5" methodology in attempt to avoid the intrusion of background processing. All inverses were pre-warmed to avoid including numba compile time.

benchmark package versions:

- Python version = 3.12.13
- numpy version = 2.4.6

TABLE 5. Time / Inverse

D	Model	τ-FLS	+ NS	FLS	pygae
(values in µs*)*					
4	21.17	7.135	24.21	21.27	21.52
5	42.12	8.867	29.07	53.06	56.97
6	72.40	15.47	38.86	84.15	142.9
7	212.1	36.94	77.18	384.3	561.2
(values in ms*)*					
8	0.7262	0.1480	0.2358	1.111	4.395
9	2.932	0.6636	1.478	7.818	26.52
10	16.89	6.192	8.843	30.39	181.2
11	82.98	28.14	39.08	266.8	1044

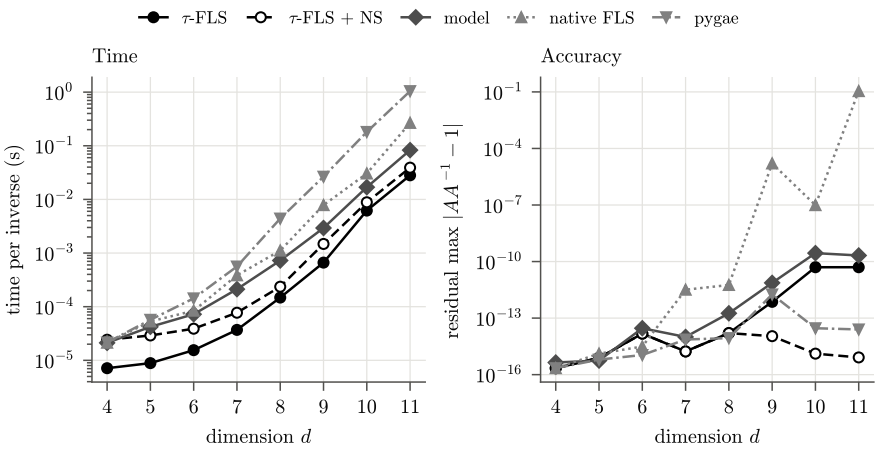


FIGURE 1. Benchmark Timings and Residuals by Method over Dimensions

TABLE 6. Residuals – $\max|AA^{-1} - 1|$

D	Model	τ-FLS	+ NS	FLS	pygae
4	4.4e-16	2.2e-16	2.2e-16	2.2e-16	2.2e-16
5	5.6e-16	7.8e-16	7.8e-16	1.3e-15	6.3e-16
6	2.9e-14	1.5e-14	1.5e-14	3.1e-15	1.1e-15
7	1.0e-14	1.7e-15	1.7e-15	3.2e-12	7.3e-15
8	1.8e-13	1.6e-14	1.6e-14	5.7e-12	8.7e-15
9	7.4e-12	7.3e-13	1.1e-14	1.6e-05	1.8e-12
10	2.8e-10	5.0e-11	1.3e-15	9.8e-08	2.9e-14
11	2.1e-10	5.0e-11	8.2e-16	1.1e-01	2.5e-14

- numba version = 0.65.1
- clif-ga version = 0.2.0

TABLE 7. optimized τ-FLS improvement factor over other methods

Method	4	5	6	7	8	9	10	11
τ-FLS / model	3.0	4.8	4.7	5.7	4.9	4.4	2.7	2.9
τ-FLS / native FLS	3.0	6.0	5.4	10.4	7.5	11.8	4.9	9.5
τ-FLS / pygae	3.0	6.4	9.2	15.2	29.7	40.0	29.3	37.1

- clifford version = 1.5.1

3.2. Reaching with Extended Precision and Parallel Processing

With standard double precision losing precision on the τ-FLS result at around 12D, a version of the inverse in double-double precision was created, and tested in the range 10D-15D. The results of that test are shown in Table 8. The primary purpose of this extended precision processing was to show that the algorithm is properly general in its mathematical form; and will produce a numerical inverse for a well conditioned multivector provided the arithmetic can support it. Secondarily, the exercise tests a practical extension to the range in dimension the multivector inverse reaches.

TABLE 8. Performance and residual comparison across dimensions d with FLS orders N.

d	N	Build (s)	f64 residual	dbl+dbl residual	f64 (ms)	dbl+dbl (ms)	Ratio	Peak (MB)
10	16	0.019	4.84×10^{-10}	8.53×10^{-15}	6.694	19.777	2.955	241
11	16	0.012	2.04×10^{-11}	7.61×10^{-15}	28.374	79.153	2.790	371
12	32	0.251	2.32×10^{-04}	1.75×10^{-15}	166.381	637.633	3.832	371
13	32	0.257	2.95×10^{-04}	7.18×10^{-15}	695.906	2551.612	3.667	371
14	64	6.648	–	8.42×10^{-08}	–	20837.411	–	589
15	64	7.523	–	4.39×10^{-07}	–	92922.446	–	590

The build times for constructing the required tables gets to be noticeable at higher dimensions, but is small in comparison to the wait times for extended precision inverse results (7 seconds vs 1.5 minutes at 15D). (Realize that build times are one time costs whereas inverse times are per-inverse costs.)

With long wait times for an extended precision result, the potential speed up using parallel processing becomes important. Numba was coached to use parallel processing for both the double and the double-double versions of FLS processing; and returned the timings reported in Table 9.

The just-in-time numba compiler provides an easy parallel processing option for today's multi-core processors (8 threads over 4 cores on the author's chromebook). The comparative results for parallel processing in the two numeric precisions is provided in Table 10.

Engaging numba's parallel processing saves enough time to warrant its use whether standard double precision or extended precision is employed. The current baseline software uses extended precision for dimensions 12D-15D,

TABLE 9. Performance and residual comparison after enabling numba parallel execution.

d	N	Build (s)	f64 residual	dbl+dbl residual	f64 (ms)	dbl+dbl (ms)	Ratio	Peak (MB)
10	16	0.019	4.84×10^{-10}	8.53×10^{-15}	1.910	4.800	2.513	252
11	16	0.016	2.04×10^{-11}	7.61×10^{-15}	7.789	19.250	2.471	379
12	32	0.253	2.32×10^{-04}	1.75×10^{-15}	52.567	157.498	2.996	379
13	32	0.261	2.95×10^{-04}	7.18×10^{-15}	216.595	690.291	3.187	379
14	64	6.722	–	8.42×10^{-08}	–	6605.000	–	598
15	64	6.554	–	4.39×10^{-07}	–	36246.962	–	598

TABLE 10. Speedup from enabling numba parallel execution (relative to non-parallel results).

d	f64 speedup	dbl+dbl speedup
10	3.50	4.12
11	3.64	4.11
12	3.17	4.05
13	3.21	3.70
14	–	3.15
15	–	2.56

standard double precision for dimensions 4D-11D, and parallel processing for the whole range of optimized inverse processing, 4D-15D. Please realize that earlier timings in this paper are for single-threaded processes only. If needed, parallel processing is easily switched off.

4. Conclusion

The author is pleased to have demonstrated a practical multivector inverse through 15D for use on common hardware with open-source software.

- 0D-3D: heuristic formulas
- 4D-7D: improving level of effort over previously known formulas
- 8D-11D: extending capability without resorting to matrices
- 12D-15D: practical processing with extended precision and parallel processing

Aside from the invention of another way to compute the inverse of a multivector, there is a certain satisfaction in uncovering the construction of the τ-involutions. The feeling that a deeper source of mathematical vitality has been struck comes through in the slots and pairs counts in Table 2. Notice that the number of pairs at any dimension equals the number of slots two dimensions back. $p[d+2] = s[d]$. This is an unintentional property that derives from an experimental construction. This observation suggests that we should

TABLE 11. Inputs to the real-product counts formulas for the pair-less dimensions, and the associated τ-involution

d	n	c	s	p	τ-involution
0	0	1	0	0	+
1	0	2	0	0	+ +
2	0	1	1	0	+ − − −
3	0	2	2	0	+ − − − − − − − +

d = dimensions,
n = FLS order,
c = center size,
s = slots,
p = pairs.

TABLE 12. As-shipped timing: one geometric product and one inverse, $\mathrm{Cl}(d, 0)$, random multivectors.

d	product	inverse	inverse / product	residual
(values in ns*)*				
0	1848	1792	0.97	0
1	1894	1966	1.04	1.1×10^{-16}
2	1920	2305	1.20	2.8×10^{-17}
3	2096	2368	1.13	5.7×10^{-17}
(values in µs*)*				
4	2.337	6.173	2.64	2.2×10^{-16}
5	3.615	8.054	2.23	7.8×10^{-16}
6	7.521	14.52	1.93	1.5×10^{-14}
7	20.23	35.81	1.77	1.7×10^{-15}
8	63.90	150.3	2.35	1.6×10^{-14}
9	226.9	664.4	2.93	7.3×10^{-13}
(values in ms*)*				
10	0.862	3.302	3.83	5.0×10^{-11}
11	3.458	14.77	4.27	5.0×10^{-11}
12	14.54	172.9	11.89	1.7×10^{-14}
13	148.9	1159	7.78	1.3×10^{-14}
14	239.5	6375	26.62	1.9×10^{-9}
15	1040	28524	27.42	1.0×10^{-6}

find τ-involutions at the lesser dimensions such that: $s[2] = 1$, and $s[3] = 2$; and, there **is** one: *Clifford conjugation* as described by Lounesto [11], pg.57. With conjugation as our τ-involution at these dimensions, $A\tau(A)$ is exactly the center. Since we routinely remove the center from the table of pairs, we conclude that there are no pairs at $d = 2, 3$. We conclude further that there are neither slots nor pairs at dimensions 0 or 1 where the multivector is already central. As shown in Table 11, the cases for dimension < 4 logically belong above Table 2. However, since these dimensions show no information useful to FLS processing this table annex was held back until now.

To conclude with an overall assessment of this package's performance, we offer the following benchmarks in Table 12.

The assumption that a real product count was a good measure of effort does not bear out with these timings. But, they are honest, and they represent a sampling of what you can expect from this [9] package if you choose to use it.

Statements and Declarations

Funding

The author received no financial support for the research, authorship, and/or publication of this article.

Competing Interests

The author declares no competing interests.

Data Availability

The author developed and tested these algorithms using an Anaconda distribution of Python and Jupyter Notebook. Original code and documentation are available from the public repository (see Ref. [10]). The reader will find browser friendly PDF versions of the Jupyter notebook files there.

The software package "clif-ga" can be imported from PyPI [9]. Documentation accompanying the software can be found at ReadTheDocs [8]. This is a capable package with special focus on the geometric product and its inverse. Aside from the inverse support described in this paper, the geometric product is implemented with virtual sign-tables culled from Hadamard matrices that leave memory for the application and don't needlessly limit the dimension. The author would entertain collaborators on this project.

References

[1] Abdulkhaev, K., Shirokov, D.: Explicit formula for inverse and determinant in geometric algebras over seven-dimensional vector spaces. Talk (ENGAGE 2025 workshop at Computer Graphics International 2025). URL `https://economics.hse.ru/en/gaa/news/1071598907.html`. Slides referenced via HSE University news post dated 15 July 2025

[2] Abdulkhaev, K., Shirokov, D.: Explicit formula for inverse and determinant in geometric algebras over seven-dimensional vector spaces. In: E. Hitzer, G. Papagiannakis, H. Hadfield (eds.) International Workshop on Empowering Novel Geometric Algebra for Graphics and Engineering, pp. 189–200. Springer Nature Switzerland (2026). DOI 10.1007/978-3-032-22539-9_14. URL `https://doi.org/10.1007/978-3-032-22539-9_14`

[3] Acus, A., Dargys, A.: The inverse of a multivector: Beyond the threshold p+q=5. Advances in Applied Clifford Algebras **28**, 1–20 (2018). URL `https://doi.org/10.1007/s00006-018-0885-4`

[4] Acus, A., Dargys, A.: The characteristic polynomial in calculation of exponential and elementary functions in clifford algebras. Mathematical Methods in the Applied Sciences **47**(14), 11353–11367 (2024). URL `https://doi.org/10.1002/mma.9524`

[5] Hadfield, H., Wieser, E., Arsenovic, A., Kern, R., The Pygae Team: pygae/clifford. URL https://doi.org/10.5281/zenodo.1453978

[6] Hitzer, E., Sangwine, S.J.: Construction of multivector inverse for clifford algebras over 2m+1-dimensional vector spaces from multivector inverse for clifford algebras over 2m-dimensional vector spaces. Advances in Applied Clifford Algebras **29**(2), 29 (2019). URL https://doi.org/10.1007/s00006-019-0942-7

[7] Jones, D.A.: Another 6d multivector inverse algorithm. Advances in Applied Clifford Algebras **36**(4), 43 (2026). DOI 10.1007/s00006-026-01455-5. URL https://doi.org/10.1007/s00006-026-01455-5

[8] Jones, D.A.: clif-ga documentation. Read the Docs (2026). URL https://clif-ga.readthedocs.io/en/latest/. Accessed: 2026-10-28

[9] Jones, D.A.: clif-ga (python package) (version 0.2.0). PyPI (2026). URL https://pypi.org/project/clif-ga/. Accessed: 2026-10-28

[10] Jones, D.A.: clif: High-dimensional clifford and geometric algebra engine (version 0.2.0). Zenodo (2026). DOI 10.5281/zenodo.23725284. URL https://doi.org/10.5281/zenodo.23725284

[11] Lounesto, P.: Clifford Algebras and Spinors, 2 edn. Cambridge University Press, Cambridge (2001)

[12] Roelfs, M.: The willing kingdon clifford algebra library (2025). URL https://arxiv.org/abs/2503.10451

[13] Shirokov, D.: On computing the determinant, other characteristic polynomial coefficients, and inverse in clifford algebras of arbitrary dimension. Computational and Applied Mathematics **40**(5), 173 (2021). URL https://doi.org/10.1007/s40314-021-01536-0

David Arthur Jones
Independent researcher
Springfield, VA, USA
e-mail: dajonesy@gmail.com
URL, ORCID: https://orcid.org/0009-0009-7693-3396