Rule-Based Simpli¯cation in Vector-Product Spaces

Songxin Liang, David J. Je®rey

Department of Applied Mathematics, University of Western Ontario,
London, Ontario, Canada

Abstract. A vector-product space is a component-free representation of the common three-dimensional Cartesian vector space. The components of the vectors are invisible and formally inaccessible, although expressions for the components could be constructed. Expressions that have been built from the scalar and vector products can be simpli¯ed using a rule-based system. In order to develop and specify the system, an axiomatic system for a vector-product space is given. In addition, a brief description is given of an implementation in Aldor. The present work provides simpli¯cation functionality which overcomes di±culties encountered in earlier packages.

1 Introduction

We start with a de¯nition from J. W. Gibbs [7], which appeared in a privately printed pamphlet that was dated 1881.

> An algebra or analytical method in which a single letter or other expression is used to specify a vector may be called a vector algebra or vector analysis.

Vector analysis has found many applications throughout engineering and science. It often simpli¯es the derivation of mathematical theorems and the statements of physical laws, while vector notation can often clearly convey geometric or physical interpretations that greatly facilitate understanding.

Almost all well known computer algebra systems provide basic vector datatypes and vector operations. The data types, however, consist of lists of components, and there is no provision for a letter or other symbol of the system to have vector attributes such as was intended by Gibbs. This is a great pity, as a perusal of any advanced textbook in physics will reveal vector expressions being formed and worked with in a component-free way. In addition to the operations that are built-in, meaning that they are distributed with the base system, the major systems o®er vector-analysis packages, for example, the *VectorAnalysis* package for Mathematica [15], the *Vector33* package for Reduce [8] and the *VectorCalculus* package in Maple. There are a number of vector analysis packages build on the well known systems, for example, the *VecCalc* package using Maple [2], the *Vect* package using Macsyma [14] and the *OrthoVec* package using Reduce [5]. All these packages, however, rely on component representation. The following shows some features of the *VectorCalculus* package in Maple.

```
> with(VectorCalculus)
> CrossProduct(A, B)
      Error, (in VectorCalculus:-CrossProduct) the first argument
      must either be the differential operator Del, or a three
      dimensional VectorField, got A
> DotProduct(A, B)
      A . B
```

It is curious that Maple checks the arguments of *CrossProduct* and requires an explicit set of components, but *DotProduct* does not. In any event, it is clear that the package does not expect abstract vectors. Once explicit components are given, the package is able to perform common tasks.

```
> A := <a, b, c>
      A := a e_x  + b e_y  + c e_z
> B := <d, e, f>
      B := d e_x  + e e_y  + f e_z
> CrossProduct(A, B)
      (b f - c e) e_x  + (c d - a f) e_y  + (a e - b d) e_z
> DotProduct(A, B)
      a d + b e + c f
```

Like the *VectorCalculus* package, almost all existing packages can only perform component-dependent operations. They cannot perform component-free operations. This means that before one can do any vector analysis, one must set the components for all vectors involved. This is inconvenient when one wants to deal with problems containing many vectors and only wants to know the relationship among them, because the expansion of vector expressions into specific components is usually tedious and error prone. With a component-free system, one is free from the distracting details of individual components and can concentrate on the meaningful results of the problems.

For more detailed commentary, we introduce some notation. Vectors are denoted by bold-face letters; for the vector product of $\mathbf{a}$ and $\mathbf{b}$, we use the notation of Chapman [3], namely $\mathbf{a} \wedge \mathbf{b}$. The scalar product is $\mathbf{a} \cdot \mathbf{b}$, and $(\mathbf{abc})$ or $(\mathbf{a}; \mathbf{b}; \mathbf{c})$ stands for the scalar triple product $(\mathbf{a} \wedge \mathbf{b}) \cdot \mathbf{c}$ of three vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$.

Compared with component-dependent systems, which have a systematic method for deriving mathematical statements, component-free systems are more difficult and challenging because vector algebra has a strange and intriguing structure [12].

- Vectors are not closed under the scalar product operation. If $\mathbf{p}$, $\mathbf{q}$ are vectors, then $\mathbf{p} \cdot \mathbf{q}$ is a scalar.
- The scalar product is commutative while the vector product is anticommutative. If $\mathbf{p}$, $\mathbf{q}$ are vectors, then $\mathbf{p} \cdot \mathbf{q} = \mathbf{q} \cdot \mathbf{p}$ while $\mathbf{p} \wedge \mathbf{q} = -\mathbf{q} \wedge \mathbf{p}$.
- Neither is associative. If $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ are vectors, then $\mathbf{p} \wedge (\mathbf{q} \wedge \mathbf{r}) \neq (\mathbf{p} \wedge \mathbf{q}) \wedge \mathbf{r}$, whereas $\mathbf{p} \cdot (\mathbf{q} \cdot \mathbf{r})$ and $(\mathbf{p} \cdot \mathbf{q}) \cdot \mathbf{r}$ are invalid.

{ Neither has a multiplicative unit. There does not exist a ¯xed vector u such that for any vector p, u ^ p = p or p ^ u = p or u ² p = p or p ² u = p.
{ Both admit zero divisors. For any vector p, p ^ p = 0; if q is a vector perpendicular to p, then p ² q = 0.
{ The two operations are connected through the operation of scalar-vector multiplication ¤ by the strange side relation p ^ (q ^ r) = (p ² r) ¤ q ¡ (p ² q) ¤ r.

In contrast to the number of component-dependent vector analysis packages, to our knowledge, only the packages [6], [10] and [12] address component-free vector operations. However, the emphasis of these packages is still on component-dependent operations, and only the package by Stoutemyer [12] provides non-trivial simpli¯cation examples. Even in Stoutemyer's package, however, simpli¯cation problems remained unsolved. For example, when he tried to simplify the vector expression

(a ^ b) ^ (b ^ c) ² (c ^ a) ¡ (a ² (b ^ c))^2

which should be simpli¯ed to zero, he only got

¡ a ² c ^ (a ² b ^ c ¤ b ¡ a ² b ^ (b ^ c)) ¡ (a ² b ^ c)^2:

When he returned this expression to his simpli¯cation engine, instead of getting the desired result 0, he could only get

a ² (a ² b ^ c ¤ b) ^ c ¡ (a ² b ^ c)^2:

He explained that the scalar factor a²b^c could be factored out, clearly revealing that the expression is zero, but the built-in scalar-factoring-out mechanism could not recognize that a ² b ^ c is a scalar. Therefore, it is necessary and meaningful to develop a new component-free vector analysis package.

As a ¯nal comment on the background, we note that the vector product has not been generalized beyond 3 dimensions except in a limited way. A generalization of the vector product in 3 dimensional space to 7 dimensional case has been proposed, but the important property p ^ (q ^ r) = (p ² r) ¤ q ¡ (p ² q) ¤ r is no longer valid [11]. So we con¯ne ourselves here to the 3 dimensional case.

2 Axiomatic Theory and Transformation Rules

In order to provide a uni¯ed picture of component-free vector algebra and component-dependent vector algebra, we de¯ne an axiomatic theory *T* for a vector-product space. The language of *T* is the set of vector operations *f*+; ¤; ²; ^*g*, where +; ¤, and ² correspond to the addition, scalar multiplication and scalar product of R^3 considered as an inner produc[illegible]Tf p0ade 360(an)-362(g75(aectt5r.10TDJ/F9190TD[(c)]TJ/F99

A0: (a + b) ^ c = a ^ c + b ^ c:
A1: a ^ b = ¡ b ^ a:
A2: a ^ (b ^ c) = (a ² c) ¤ b ¡ (a ² b) ¤ c:
A3: (abc) = (bca); where (abc) denotes (a ^ b) ² c:
A4: (abc) = 0 implies that a; b; c are linearly dependent.

Based on the axioms of *T*

the left hand side of T6.

For theorem T7, we can use a similar method as above so we omit the details. For theorem T8, taking scalar product of both sides of T6 with **h** and using axiom A3, the desired result follows immediately. For theorem T9, we can use a similar method as theorem T6, so we omit the details again. Consequently, all theorems are proved.

Now we de¯ne another axiomatic theory T' for component-dependent vectors in R^3. The de¯nitions of the vector operations in T' are just the standard ones. For $\mathbf{a}; \mathbf{b}$ *2* R^3 with $\mathbf{a} = (a_1; a_2; a_3)$ and $\mathbf{b} = (b_1; b_2; b_3)$, and *k 2* R, de¯ne

$$\begin{aligned} k ¤ \mathbf{a} &= (ka_1; ka_2; ka_3) : \\ \mathbf{a} + \mathbf{b} &= (a_1 + b_1; a_2 + b_2; a_3 + b_3) : \\ \mathbf{a} ² \mathbf{b} &= a_1b_1 + a_2b_2 + a_3b_3 : \\ \mathbf{a} \wedge \mathbf{b} &= (a_2b_3 \ ¡ \ a_3b_2; a_3b_1 \ ¡ \ a_1b_3; a_1b_2 \ ¡ \ a_2b_1) : \end{aligned}$$

It is routine to check that the vector operations de¯ned above satisfy all the axioms of T. Therefore, we have showed that T' is a model of T. In this way, we provide a uni¯ed picture of the axiomatic speci¯cation of vector operations, the computation of abstract (component-free) vectors, and the computation of concrete vectors (component-dependent).

Now we turn to transformation rules. In order to achieve a normal form for an input vector expression, or to simplify a vector expression to its shortest normal form, we need transformation rules. The transformation rules consist of the 4 axioms (A0)-(A3) and 9 theorems (T1)-(T9) of T, together with the *inverses* of theorems (T6)-(T9). By the *inverse* of a formula (theorem) $A = B$, we mean $B = A$.

Roughly speaking, the transformation rules can be divided into two types: expansion type and combination type. An *expansion type rule* expands a single term into a sum of di®erent terms, or transfers a single term to another single term. For example, $(\mathbf{abc}) ¤ \mathbf{d} = (\mathbf{d} ² \mathbf{a}) ¤ (\mathbf{b} \wedge \mathbf{c}) + (\mathbf{d} ² \mathbf{b}) ¤ (\mathbf{c} \wedge \mathbf{a}) + (\mathbf{d} ² \mathbf{c}) ¤ (\mathbf{a} \wedge \mathbf{b})$ is an expansion type rule. A *combination type rule* combines a sum of di®erent terms into a single term. For example, $(\mathbf{d} ² \mathbf{a})(\mathbf{hbc}) + (\mathbf{d} ² \mathbf{b})(\mathbf{ahc}) + (\mathbf{d} ² \mathbf{c})(\mathbf{abh}) = (\mathbf{d} ² \mathbf{h})(\mathbf{abc})$ is a combination type rule.

3 Structure and Description of the Package

Our package is developed using Aldor. Aldor [1] is a programming language with a two-level object model of categories and domains. Types and functions are ¯rst class entities allowing them to be constructed and manipulated within Aldor programs just like any other values. Aldor is an ideal tool for symbolic mathematical computations.

For the package, we ¯rst de¯ne a vector space category *VectorSpcCategory* as follows.

```
Join(ExpressionType,VectorAlgCategory R)== add
{
  ...
},
```

where the part within the braces after *add* is the implementation details of the domain which composes the rest of this paper.

3.1 Normal Form

A common strategy for simplifying expressions in computer algebra is to de¯ne a normal form. We de¯ne a normal form for vector expressions as follows.

De¯nition 1 (Normal Form). *A normal form for a vector expression is a sum of terms. Each term may consist of a real coe±cient, a scalar part and a vector part. The vector part consists of either a single letter or a vector product, or there is no vector part if the term is not really a vector. If there is a scalar part for a term, then the factors of the scalar part may consist of single letters, scalar products and scalar triple products. The terms are in ascending order which is determined by the* termOrder *de¯ned in section 3.2.*

In our vector algebra package, all input vector expressions will be transferred automatically into their normal forms. A vector expression can be presented in di®erent normal forms. For example, according to theorem T6 in section 2, we can present the same expression in two di®erent normal forms: (abc) ¤ d and (c ² d) ¤ (a ^ b) ¡ (b ² d) ¤ (a ^ c) + (a ² d) ¤ (b ^ c). However, by using the simpli¯cation functionality (see section 4.2), we can get the shortest normal form for a given vector expression.

3.2 Data Representations and Term Order

There are two components for a normal form of a vector expression: data representation and term order. In our package, a vector expression is presented internally as:

Rep==List Term, and

Term==Record (coe:R, sca:List List String, vec:List String),

where *coe*, *sca* and *vec* are respectively the real coe±cient, the scalar part and the vector part of a term. For example, the term ¡2(a ² b)(abc) ¤ (c ^ d) can be expressed as [¡2; [[\a"; \b"]; [\a"; \b"; \c"]]; [\c"; \d"]]. Inspired by Stoutemyer's unsolved problem, we adopt such data representation because we want to separate the scalar part of a term from the vector part of the term.

In the scalar part *sca*, there are three di®erent kinds of lists of strings. List of length 1 stands for a scalar of single letter, list of length 2 stands for a scalar product, and list of length 3 stands for a scalar triple product.

Similarly, for the vector part *vec*, list of length 0 means that the term is not really a vector, list of length 1 stands for a vector of single letter, and list of length 2 stands for a vector product.

On the other hand, in order to implement the transformation rules in section 2, we also need another minor data structure.

mixTerm == Record(scal:scaList, vec2:List String), where

scaList == List scaTerm, and

scaTerm == Record(coe2:R, sca2:List List String).

This representation is necessary when one wants to simplify a vector expression which contains terms with the same vector parts. For example, for vector expression (d ^{2} a)(hbc) ¤ c + (d ^{2} b)(ahc) ¤ c + (d ^{2} c)(abh) ¤ c, we can combine the scalar parts of the three terms because they have same vector parts c. Then, [(d ^{2} a)(hbc) + (d ^{2} b)(ahc) + (d ^{2} c)(abh)] ¤ c is a *mixTerm*, the *scal* part is (d ^{2} a)(hbc) + (d ^{2} b)(ahc) + (d ^{2} c)(abh), the *vec2* part is c, and there are three *scaTerms*: (d ^{2} a)(hbc), (d ^{2} b)(ahc) and (d ^{2} c)(abh). Using the transformation rules from section 2, we can simplify the expression to (d ^{2} h)(abc) ¤ c.

As mentioned above, the second component for a normal form is term order. Unlike polynomials which have a natural way to de¯ne term order, we have to choose a term order for our purpose. The order *termOrder* is de¯ned step by step as follows.

Firstly, we use the natural order for strings.

Secondly, we de¯ne the order for lists of strings. Given two lists of strings L_1 and L_2, then $L_1 < L_2$ *( )* $\#L_1 < \#L_2$, or $\#L_1 = \#L_2$ and there exists a natural number i such that for all $0 < j < i$, $L_1(j) = L_2(j)$ and $L_1(i) < L_2(i)$, where $\#L$ denotes the length of list L.

Thirdly, we de¯ne the order for lists of lists of strings in the same way as above for lists of strings.

Finally, we de¯ne the order of terms for vector expressions. Given two terms $T_1 = [coe_1; sca_1; vec_1]$ and $T_2 = [coe_2; sca_2; vec_2]$, then $T_1 < T_2$ *( )* $vec_1 < vec_2$, or $vec_1 = vec_2$ and $sca_1 < sca_2$, or $vec_1 = vec_2$ and $sca_1 = sca_2$ and $coe_1 < coe_2$ (if R is an ordered arithmetic type).

For example, given a vector expression 5(abc)(a ^{2} d) ¤ (b ^ c) + 2(b ^{2} c) ¤ (a ^ c) + (acd)(a ^{2} b) ¤ c, after being sorted by *termOrder*, the expression will become (a ^{2} b)(acd) ¤ c + 2(b ^{2} c) ¤ (a ^ c) + 5(a ^{2} d)(abc) ¤ (b ^ c).

4 Implementation of the Package

This section addresses the implementation details of the package. These include the implementation of transformation rules, the implementation of the simpli¯cation function and the implementation of vector algebra operations.

4.1 Implementations of Transformation Rules

Now we turn to the implementations of transformation rules. The main skill we use is pattern matching [13]. Since the implementations are detail-involved, we only give brief descriptions of the algorithms.

The implementations of expansion type rules are not complicated. For example, the following algorithm is to implement the transformation rule (abc) ¤ d = (d ^{2} a) ¤ (b ^ c) + (d ^{2} b) ¤ (c ^ a) + (d ^{2} c) ¤ (a ^ b):

```
Algorithm Rule01
Input: a Rep xx.
Output: a Rep yy.

  { Let tx range over the terms of xx:
  { If there are lists in tx which match the pattern (abc) ¤ d, replace them with
    the lists representing (d ² a) ¤ (b ^ c) + (d ² b) ¤ (c ^ a) + (d ² c) ¤ (a ^ b):
  { Output the resulting Rep.
```

Compared with the implementations of expansion type rules, the implementations of combination type rules are much more complicated. For example, in order to implement the transformation rule (d ^{2} a)(hbc) + (d ^{2} b)(ahc) + (d ^{2} c)(abh) = (d ^{2} h)(abc), ¯rst we have to implement a subalgorithm *Match?* to check if any given three scaTerms match the pattern of this rule, then we have to implement another subalgorithm *Comb* which combines the matching scaTerms into a single Term using this rule.

Then the algorithm for implementing the transformation rule (d ^{2} a)(hbc) + (d ^{2} b)(ahc) + (d ^{2} c)(abh) = (d ^{2} h)(abc) can be described as follows.

```
Algorithm Rule02
Input: a Rep xx.
Output: a Rep yy.

  { Change the data representation of xx from Rep to List mixTerm.
  { Let tx range over xx, and let L be the scal part of tx:
  { If #L < 3, then leave it for output. Otherwise, again let L denote the
    collection of scaTerms in L in which there are lists of length 2 and length 3
    in their sca2 part, and leave other scaTerms for output.
  { In a while-loop, check if any given three scaTerms match the rule using the
    subalgorithm Match?.
  { If there is no such three scaTerms or #L < 3, then break the loop. Otherwise,
    combine the three scaTerms using the subalgorithm Comb and update L.
  { output the remaining L:
```

4.2 Simpli¯cation

There are two levels of simpli¯cation. *Simplify0* is the basic level which, as part of the de¯nitions of vector algebra operations, transfers an input vector expression into its normal form. *Simplify* is the advanced level which, based on *Simplify0*, transfers a vector expression into its shortest normal form.

The basic level can be described as follows.

```
Algorithm Simplify0
Input: an element x of VectorAlg.
Output: an element of VectorAlg.
```

{ Let *tx* range over the data representation *xx* of *x:*
{ Make the scalar part and vector part of *tx* in ascending order, and use the transformation rules to check if *tx* is zero.
{ Collect non-zero terms *tx*, and denote is *yy:*
{ Make *yy* in ascending order and combine like terms.
{ Output the domain element *y* of *yy:*

The advanced simpli¯cation can be described as follows.

Algorithm Simplify
Input: an element *x* of VectorAlg.
Output: an element of VectorAlg.

{ Perform basic simpli¯cation on *x*.
{ Apply transformation rules to the resulting expression one by one within a while-loop and then perform basic simpli¯cation.
{ If the resulting expression is shorter than the original one then we replace the original one with the resulting expression.
{ If at any time the numbumbnum

At this point, we are ready to test some examples using our package. All computations have been performed on a Pentium IV PC with 3.2 GHz CPU and 1 GB RAM.

Example 1.

We ¯rst test Stoutemyer's unsolved problem [12].

```
%14 >> ((a^b)^(b^c)).(c^a)-(a.(b^c))*(a.(b^c))
0 @ VectorAlg(MachineInteger)
                          Comp: 0 msec, Interp: 40 msec
```

From the example above, we can see that our package is quite e±cient. It only takes 40 milliseconds to get the result.

Example 2. Now, we test some more examples also from Stoutemyer:

{ (a ¡ d)^(b ¡ c)+(b ¡ d)^(c ¡ a)+(c ¡ d)^(a ¡ b) ¡ 2¤(a^b+b^c+c^a) = 0.

{ (b ^ c) ^ (a ^ d) + (c ^ a) ^ (b ^ d) + (a ^ b) ^ (c ^ d) + 2(abc) ¤ d = 0.

```
%15 >> (a-d)^(b-c)+(b-d)^(c-a)+(c-d)^(a-b)-2*(a^b+b^c+c^a)
0 @ VectorAlg(MachineInteger)
                          Comp: 0 msec, Interp: 40 msec
%16 >> v1:=(b^c)^(a^d)+(c^a)^(b^d)+(a^b)^(c^d)+2*s3p(a,b,c)*d
(bcd)*a-(acd)*b+(abd)*c-(abc)*d @ VectorAlg(MachineInteger)
                          Comp: 10 msec, Interp: 30 msec
%17 >> Simplify(v1)
O @ VectorAlg(MachineInteger)
                          Comp: 10 msec, Interp: 30 msec
```

Example 3 The following examples come from Cunningham [4].

{ a ^ (b ^ c) + b ^ (c ^ a) + c ^ (a ^ b) = 0.

{

6 Summary

In this paper, we have presented a rule-based component-free vector algebra package developed using the computer programming language Aldor. All the examples we encountered in the literature are simpli¯ed to their shortest normal forms. The key idea is to choose an appropriate data structure and a suitable set of transformation rules. In the future, we will add more functions to the package, for example, solving vector equations and systems of vector equations.

Acknowledgement. The authors would like to thank Dr. Marc Moreno Maza for his helpful suggestions and the referees for their valuable comments on improving this paper.

References

1. Aldor.org, 2002. Aldor User Guide. http://www.aldor.org/docs/aldorug.pdf/.
2. Belmonte, A. and Yasskin, P.B., 2003. A vector calculus package for Maple. http://calclab.tamu.edu/maple/veccalc/.
3. Chapman S., Cowling, T.G., 1939. The Mathematical Theory of Non-uniform Gases. Cambridge University Press.
4. Cunningham, J., 1969. Vectors. Heinemann Educational Books Ltd, London.
5. Eastwood, J.W., 1991. OrthoVec: version 2 of the Reduce program for 3-d vector analysis in orthogonal curvilinear coordinates. Comput. Phys. Commun. 64, 121-122.
6. Fiedler, B., 1997. Vectan 1.1. Manual Math. Inst., Univ. Leipzig.
7. Gibbs, J.W., 1961. Elements of vector analysis. In: The Scienti¯c Papers of J. Willard Gibbs, Dover.
8. Harper, D., 1989. Vector33: A Reduce program for vector algebra and calculus in orthogonal curvilinear coordinates. Comput. Phys. Commun. 54, 295-305.
9. Patterson, E.M., 1968. Solving Problems in Vector Algebra. Oliver & Boyd Ltd, Edinburgh-London.
10. Qin, H., Tang, W.M., Rewoldt G., 1999. Symbolic vector analysis in plasma physics. Comput. Phys. Commun. 116, 107-120.
11. Silagadze, Z.K., 2002. Multi-dimensional vector product. arXiv: math.ra, 0204357.
12. Stoutemyer, D.R., 1979. Symbolic computer vector analysis. Computers & Mathematics with Applications, 5, 1-9.
13. Tanimoto, S.L., 1990. The Elements of Arti¯cial Intelligence Using Common Lisp. Computer Science Press, New York.
14. The Mathlab Group, 1983. Macsyma Reference Manual (Vol. II), MIT.
15.