New Physics: Sae Mulli,
Vol. 71, No. 8, August 2021, pp. 711∼718
http://dx.doi.org/10.3938/NPSM.71.711

Index Notation and Simplification of Tensor Expressions

Dal Ho Park*
Department of Computer and Electronic Physics Sangji University, Wonju 220-702, Korea

(Received 03 May 2021 : revised 17 June 2021 : accepted 28 June 2021)

We discuss the pattern matching and the canonicalization as simplification strategies of tensor expressions in index notation and explain that the time required for simplification increases as the symmetry of the indices increases. The performance of simplification by using pattern matching indicates that it is not appropriate for use in practice, but the canonicalization based on the 'Butler-Portugal' algorithm is very efficient and so is widely used in general relativity. In this paper, in order to improve the existing simplification strategy, we introduce the concepts of index 'kind' to handle various indices and 'metric state' to solve the difficulties due to non-covariant operators. As a result, tensor expressions containing various indices and non-covariant operators are correctly simplified. In addition, we convert the implementation of the 'Butler-Portugal' algorithm from the GNU C language to the standard C++17 language, and as a result, we found that the asymptotic complexity of the canonicalization was greatly improved in the Windows operating system.

Keywords: Index notation, Symbolic tensor algebra, Index canonicalization

인덱스 표기법과 텐서 표현의 단순화

박달호*
상지대학교 컴퓨터전자물리학과, 원주 220-702, 대한민국

(2021년 5월 03일 받음, 2021년 6월 17일 수정본 받음, 2021년 6월 28일 게재 확정)

인덱스를 갖는 텐서 표현의 단순화 전략으로 패턴 매칭 방법과 정규화를 이용한 방법을 논의하였고, 인덱스의 대칭이 증가함에 따라 단순화 소요 시간이 증가할 수 밖에 없음을 설명하였다. 패턴 매칭 방법의 단순화 성능은 실제 사용이 곤란할 정도로 좋지 않지만, 'Butler-Portugal' 알고리즘에 기반한 정규화 방법은 매우 효율적이어서 일반 상대론 분야에서 많이 사용되고 있다. 본 논문에서는 기존 단순화 방법을 개선하기 위해 다양한 인덱스를 다루기 위한 인덱스 '계열' 개념을 도입하였고, 비공변 연산자의 작용에 따른 난점을 해결하기 위한 자료구조인 '메트릭상태'도 도입하였다. 그 결과 여러 가지 인덱스와 비공변 연산자가 포함된 텐서 표현도 올바르게 단순화할 수 있음을 설명하였다. 또한 'Butler-Portugal' 알고리즘 구현을 GNU C 언어로부터 2017년 표준 C++ 언어로 변환하였고, 결과적으로 정규화 소요 시간의 점근적 복잡도가 윈도우 운영체제에서 크게 향상됨을 확인하였다.

Keywords: 인덱스 표기법, 텐서 대수, 인덱스 정규화

*E-mail: dhpark@sangji.ac.kr

I. 서 론

일반 상대론 분야에서 (수치 연산이 아닌 기호 연산으로서의) 텐서 연산 자동화는 오래된 연구 과제이다. [1] 초창기 텐서 연산의 자동화는 좌표계와 계량 텐서가 주어졌을 때 곡률 텐서를 포함하는 텐서 성분들을 얼마나 효율적으로 계산하느냐에 주안점을 두었었다. [2] 그러나 최근의 컴퓨팅 환경에서 텐서 성분 연산은 그리 어려운 작업이 아니게 되었다. 더군다나 그러한 연산 결과의 단순화(simplification)는 텐서 표현과는 무관한 단지 기호 연산 패키지의 단순화 명령, 예를 들면 Mathematica에서의 Simplify 또는 Maple에서의 simplify를 사용하면 된다. 따라서 텐서 연산에서 소위 ‘ctensor’ [3]라고 불리는 텐서 성분에 대한 연산은 잘 알려진 기호 연산 패키지를 사용하는 것으로 쉽게 해결된다.

한편, ‘itensor’ [3,4]라고 불리는 인덱스를 갖는 텐서 연산은 인덱스 기호를 그대로 사용하므로 연산 자체보다는 연산 결과를 단순화하는 작업에 더욱 어려움이 있다. 그 이유는 연산 결과 자체가 인덱스를 갖는 텐서 표현이므로 그 텐서 표현이 갖는 모든 성질을 사용해서 단순화해야 하기 때문이다. 그래서 인덱스를 갖는 텐서 표현을 단순화하는 전략은 대략 2000년까지는 수작업의 흉내내기 방법, 즉 패턴 매칭과 규칙 기반 방법을 사용하였지만 많은 어려움이 있었다. [5–11] 결정적인 문제는 텐서 자체의 대칭이나 더미 인덱스로 인한 대칭이 커짐에 따라 단순화에 소요되는 시간이 지수적으로 증가하여 단순한 표현이 아니면 실질적으로 사용하기 곤란하다는 것이었다. 그런데 순열 대칭에 대한 군론을 사용한 ‘Butler-Portugal’ 알고리즘으로 텐서 표현을 효율적으로 정규화(canonicalization)하는 방법이 2002년 개발되었다. [12–14] 특히 연산 효율을 극대화하기 위해 ‘Butler-Portugal’ 알고리즘을 C언어로 구현한 xPerm [15]은 최근에도 많은 텐서 연산 패키지와 관련 연구에서 사용되고 있다. [16–20]

본 논문에서는 텐서 표현의 기존 단순화 방법을 개선하여 mTensor [21]에 구현한 결과를 소개한다. 이를 위하여 II절에서는 텐서 표현에서 사용되는 인덱스를 분류하고, 인덱스 ‘계열’ 개념을 도입할 것이다. 또한, 비공변 연산자가 작용한 텐서를 단순화하기 위한 자료구조인 ‘메트릭상태’에 대하여도 설명할 것이다. III절에서는 텐서 표현 단순화에 필요한 준비 과정으로 내부 표현으로의 변환, 연산자에 의한 인덱스 대칭의 확장, 텐서곱에서 동일한 이름의 텐서들에 의한 교환 대칭, 그리고 텐서곱을 확장된 대칭을 갖는 한 개의 텐서로 병합하는 과정 [8]을 설명할 것이다.

IV절에서는 패턴 매칭을 이용한 텐서 표현의 단순화 방법을 설명하면서 텐서 대칭이 커질수록 단순화의 소요시간이 길어질 수밖에 없는 이유를 밝힐 것이다. 그리고 V절에서는 ‘Butler-Portugal’ 알고리즘을 사용하는 정규화에 의한 단순화 방법을 설명할 것이다. xPerm을 기반으로한 텐서 연산 패키지인 xTensor [16]에서는 비공변 연산자를 제대로 다루지 못할뿐더러 비공변 연산자가 포함된 텐서 표현의 정규화에 오류도 발생시킨다. IV절과 V절에서 이러한 오류들은 ‘메트릭상태’를 이용하여 해결할 수 있음을 설명할 것이다.

mTensor [21]에서는 V절에서 설명하는 정규화를 구현하기 위해 xPerm의 C코드를 사용하지 않는다. 사실 xPerm은 C언어로 구현하였음에도 특정 프로그램인 GNU 컴파일러 고유의 문법을 사용했기 때문에 운영체제에 따라 연산 효율이 달라진다. 예를 들면 유닉스 계열의 운영체제에 비해 윈도우 운영체제에서 상당히 비효율적으로 동작한다. 특히 VI절에서 보듯이 xPerm 코드 체계의 메모리 관리 문제가 윈도우 운영체제에서 명확하게 드러난다. 본 논문에서는 이러한 문제를 개선하기 위해 xPerm의 C 언어 코드를 2017년 표준안 [22]을 따르는 C++ 언어로 변환하여 특정 컴파일러에 대한 종속성을 제거하였고, 정규화 과정의 일관성을 확인하는 절차에 이진 검색 알고리즘을 사용하였다.

VI절에서는 xPerm [15]과 mTensor [21] 사이의 텐서 표현에 대한 정규화 성능을 리눅스 운영체제와 윈도우 운영체제에서 각각 비교할 것이고, VII절에서 결론 및 토론을 전개할 것이다.

II. 텐서 인덱스

일반 상대론에서 텐서의 인덱스는 크게 두 가지 의미로 사용된다: 추상 인덱스 표기법(abstract index notation) [23–25]에서 인덱스는 $(0,2)$ 타입의 텐서 T_{ab}처럼 타입을 결정하는 표식자이다. 반면에 성분 인덱스는 $T_{ab} = T_{\mu\nu} dx^\mu_a dx^\nu_b$에서의 $T_{\mu\nu}$처럼 주어진 기준계에서의 성분 텐서를 나타낸다. 편미분 연산자 ∂_μ는 성분 텐서에 작용하는 것으로 보통 해석한다: $\partial_\mu T_\nu{}^\rho = \partial T_\nu{}^\rho / \partial x^\mu$. 그러나 ∂_a를 추상 인덱스에 작용하는 특별한 미분 연산자로 해석할 수도 있다: $\partial_a T_b{}^c$. [25] 본 논문에서의 인덱스는 두 가지 의미 모두에서 사용할 수 있는 것으로 간주한다. 사실 IV절과 V절에서 논의하는 단순화 방법은 텐서의 인덱스뿐만 아니라 인덱스를 갖는 미분 형식(differential form)에도 사용 가능하므로 용어상으로는 ‘텐서’보다는 ‘인덱스를 갖는 객체(indexed object)’로 부르는 것이 좀 더 정확한 표현이다.

인덱스를 구분할 때 다른 인덱스와 축약(contraction)되지 않은 인덱스를 프리(free) 인덱스, 축약된 인덱스를 더미(dummy) 인덱스로 부른다. 그리고 위 첨자(contravariant) 인덱스와 아래 첨자(covariant) 인덱스로도 구분한다. 인

덱스가 a, b와 같은 기호이면 정규 인덱스, 숫자이면 성분 인덱스로 부른다.

일반 상대론에서 보통 사용하는 인덱스들을 나열해 보자. 추상 인덱스는 라틴 소문자 $a, b, c, \ldots$를 사용하고, 고차원 공간에서의 추상 인덱스나 스피너 인덱스는 라틴 대문자 $A, B, C, \ldots$, 시공간의 성분 인덱스는 그리스 소문자 $\mu, \nu, \rho, \ldots$를 주로 사용한다. 또한 스피너 인덱스를 위해 $\alpha, \dot{\alpha}, \beta, \dot{\beta}, \ldots$, 시공간에서 정규 직교(orthonormal) 기준계의 성분 인덱스로 $\hat{\mu}, \hat{\nu}, \hat{\rho}, \ldots$, 공간에서 정규 직교 기준계의 성분 인덱스로 $\hat{\imath}, \hat{\jmath}, \hat{k}, \ldots$ 등을 사용한다. 따라서 인덱스의 사용 목적에 따라 구분되는 개념인 인덱스 '계열(kind)'을 도입할 필요가 있다. [21] 추상 인덱스로 사용하는 인덱스 계열은 수학적으로는 벡터 번들 개념으로 볼 수도 있다. 예를 들면, 게이지 장 $F_{\mu\nu}{}^{a}$의 표현에서 $\mu, \nu, \ldots$는 시공간의 탄젠트 번들에 대한 인덱스이고, $a, b, \ldots$는 내부 공간의 벡터 번들에 대한 인덱스이다. 한편 성분 텐서의 경우에는 동일한 공간 내에서 사용할지라도 기준계에 따라 다른 계열의 인덱스를 사용한다. 따라서 인덱스 계열을 사용하려면 그 계열을 어떤 목적으로 사용할 지 명시적으로 먼저 결정해야 한다. 특히 인덱스 계열에 따라 다른 계량 텐서를 사용해야 한다. 예를 들면, 시공간의 계량 텐서는 $g_{\mu\nu}$이고, 내부 공간의 계량 텐서는 ϕ_{ab}, 정규 직교 기준계에서의 계량 텐서는 $\eta_{\hat{\mu}\hat{\nu}}$ 등인 경우이다. mTensor [21]에서 인덱스 계열은 DefKind 명령으로 정의한다.

텐서의 계열은 사용하는 인덱스의 계열로 결정된다. 예를 들면, $g_{\mu\nu}$는 Greek 계열, T_{ab}는 Latin 계열, $F_{\mu\nu}{}^{a}$는 {Greek, Greek, Latin} 계열이다. 한편, $\nabla_a T^{\nu}_{\mu}$ 처럼 연산자의 인덱스 계열은 Latin 계열이고 텐서의 계열은 Greek이면 연산자 입장에서 T^{ν}_{μ}는 단지 스칼라로 간주할 것이다. 또한 연산자 ∇_a는 Latin 계열의 인덱스에 대해서는 공변 연산자이지만 Greek 계열의 인덱스에 대해서는 비공변 연산자이다. 이러한 이유 때문에 텐서에 작용하는 연산자의 계열도 고려해야 한다. mTensor [21]에서 텐서는 DefTensor, 계량 텐서는 DefMetric, 미분 연산자는 DefDerivativeOperator 명령으로 각각 정의하고, SetMetricConnection 명령으로 특정 계량 텐서에 대한 공변 도함수를 설정한다.

xPerm을 기반으로한 텐서 연산 패키지인 xTensor [16]에서는 비공변 연산자가 포함된 텐서 표현의 정규화에 오류가 발생한다. 예를 들면, 대칭 텐서 S_{ab}와 반대칭 텐서 A^{ab}로 구성된 다음 표현은 0이 되어여 한다: $S_{ab}\partial_\mu A^{ab} \to 0$. 그러나 $S_{a}{}^{b}\partial_\mu A^{a}{}_{b}$는 ∂_μ가 인덱스 a, b가 속한 계열의 계량 텐서에 대해 공변 연산자가 아니면 일반적으로는 0이 되지 않음에도 xTensor [16]의 정규화 명령을 사용하면 0이 출력된다. 이러한 치명적 오류를 발생시키지 않으려면 텐서 표현의 모든 인덱스에 대해 '위 첨자(+1) 또는 아래 첨자(-1)', '해당 인덱스 계열의 계량 텐서', 그리고 '작용한 연산자의 공변 연산자 여부'를 저장하고 필요에 따라 갱신할 수 있는 자료 구조를 도입해야 한다. 예를 들면, 텐서 $\partial_\mu v^a$의 인덱스 μ의 경우 그 자료구조는 {-1, g, +1}이 되고, 인덱스 a의 경우 비공변 연산자가 작용하므로 그 자료구조는 {+1, ϕ, 0}이 된다. 앞으로는 이 자료구조를 '메트릭상태'로 부를 것이다.

III. 텐서 표현의 단순화를 위한 준비

텐서 표현을 단순화하려면 그 표현을 구성하는 각각의 텐서를 여러 가지 상태 정보를 갖는 내부 표현으로 먼저 변환시켜야 한다. 이때 중요한 상태 정보로는 텐서의 이름, 프리 인덱스와 더미 인덱스, 그리고 '메트릭상태'가 있다. 텐서에 연산자가 포함되면 그 연산자의 이름도 텐서의 이름에 포함시켜야 한다. 예를 들면, 텐서 $\nabla_a T_b{}^a$의 이름은 ∇T이고, 프리 인덱스는 $\{b\}$, 더미 인덱스는 $\{a\}$가 된다.

각 인덱스의 '메트릭상태' 정보가 중요한 이유는 계량 텐서로 인덱스의 첨자 위치를 바꿀 수 있기 때문이다: $T_b{}^c = g^{cd}T_{bd}$. 작용한 연산자가 공변 연산자라면 계량 텐서를 그 연산자 밖으로 이동시킬 수 있으므로 인덱스 b, c의 '메트릭상태'는 변경되지 않는다:

$$\begin{aligned}\nabla_a T_b{}^c \equiv (\nabla T)_{ab}{}^{c} &= \nabla_a (g^{cd} T_{bd}) \\ &= g^{cd} \nabla_a T_{bd} \equiv g^{cd} (\nabla T)_{abd}.\end{aligned} \tag{1}$$

그러나 비공변 연산자의 작용은 텐서 인덱스의 '메트릭상태'를 변경시킨다; 비공변 연산자가 작용한 텐서 표현

$$\begin{aligned}\partial_a T_b{}^c \equiv (\partial T)_{ab}{}^{c} &= \partial_a (g^{cd} T_{bd}) \\ &\neq g^{cd} \partial_a T_{bd} \equiv g^{cd} (\partial T)_{abd}\end{aligned} \tag{2}$$

에서 계량 텐서는 연산자 밖으로 이동할 수 없으므로 인덱스 b, c의 첨자 위치를 계량 텐서가 변경시킬 수 없다.

텐서 표현을 단순화할 때 인덱스의 모든 대칭을 고려해야 한다. 그런데 텐서 표현의 대칭에는 텐서 자체가 갖고 있는 대칭뿐만 아니라 연산자의 연속적인 작용에 의해 확장된 대칭도 있다. 먼저, 토션(torsion)이 없을 때 공변 도함수가 스칼라 표현에 두 번 연속해서 작용하면 그 연산자의 인덱스들은 교환된다: $\nabla_a \nabla^b f = \nabla^b \nabla_a f$. 또한, 아래 첨자 인덱스를 갖는 편미분 연산자가 임의의 텐서에 두 번 이상 연속해서 작용할 때에도 그 연산자의 인덱스들은 서로 교환된다: $\partial_a \partial_b T_c{}^d = \partial_b \partial_a T_c{}^d$. 물론 위 첨자 인덱스는 ∂_a가 공변 연산자가 아니므로 교환되지 않는다: $\partial_a \partial^b T_c{}^d \neq \partial^b \partial_a T_c{}^d$.

텐서곱 표현에 동일한 이름의 텐서가 두 개 이상 존재할 때에도 그 텐서곱 표현의 인덱스 대칭이 확장된다: 예를

들면, $T_a{}^c T_{bc} = T_{bc} T_a{}^c = T_b{}^c T_{ac}$. 따라서 텐서곱 표현에서 두 개 이상의 텐서가 교환 가능한가를 확인해야 하는데 이때에도 ‘메트릭상태’ 정보가 필요하다. 왜냐하면, 예를 들어 $(\nabla_a T_{cd})(\nabla_b T^{cd}) = (\nabla_b T_{cd})(\nabla_a T^{cd})$에서 인덱스 a와 b는 교환되지만, $(\partial_a T_{cd})(\partial_b T^{cd}) \neq (\partial_b T_{cd})(\partial_a T^{cd})$처럼 비공변 연산자가 작용하면 인덱스 a와 b는 교환되지 않기 때문이다.

텐서곱의 인덱스 대칭을 모두 구하려면 먼저 각각의 텐서에 대해서 그 자체의 대칭과 연산자의 작용에 의한 확장 대칭을 병합한다. 그 후, 동일한 이름의 교환 가능한 텐서가 두 개 이상 있을 때의 대칭을 최종적으로 병합한다. 여기까지 진행되면 단순화 과정에서의 텐서곱 표현을 확대된 대칭을 갖는 한 개의 텐서로 취급할 수 있다. [8]

IV. 패턴 매칭과 단순화

임의의 텐서 표현은 텐서곱의 합으로 전개할 수 있다. 그런데 텐서곱은 III절에서 설명하였듯이 확대된 대칭을 갖는 하나의 텐서로 병합할 수 있으므로, 결국 임의의 텐서 표현은 텐서들의 합으로 볼 수 있다. 따라서 텐서 표현의 단순화는 원리적으로는 간단하다: 먼저 텐서합을 구성하는 각각의 텐서 중에서 대칭에 의해 0이 되지 않는 것들만 남겨 둔다. 이 때 n개의 텐서가 남아있으면 i 번째 텐서와 j 번째 텐서를 비교하여 동일하면 합친다. 여기서 i는 $1 \le i \le n-1$번 반복하고, j는 각각의 i에 대해서 $i < j \le n$번을 반복한다.

텐서가 대칭에 의해 0이 되는 경우는 본질적으로 더미 인덱스 때문이다. 예를 들면, 반대칭 텐서 A_{ab}의 인덱스가 축약되는 경우가 있다: $A_a{}^a \to 0$. 또한 반대칭 텐서 A_{ab}와 대칭 텐서 S_{ab}가 축약되는 경우도 있다: $A_{ab}\partial_c S^{ab} \to 0$. 일반적으로는, 텐서의 더미 인덱스들을 (가능하면) 모두 아래 첨자로 만든 후, 그 더미 인덱스들을 교환하였을 때의 대칭이 반대칭이면 그 텐서 표현은 0이어야 한다. (이 때 더미 인덱스의 이름 a, b는 중요하지 않으므로 Mathematica의 패턴 변수 $a_, b_$를 사용한다.) 이러한 작업을 텐서 대칭이 허용하는 모든 가능한 인덱스 표현에 대해 0이 될 때까지 반복한다. 그래도 0이 안되면 그 텐서는 남겨 둔다.

더미 인덱스를 항상 아래 첨자로 변경할 수는 없다. 일반적으로 더미 인덱스를 아래 첨자로 변경할 수 있는가를 판단하려면 인덱스의 ‘메트릭상태’를 근거로 해야 한다. 예를 들면, $A_{ab}\partial_c S^{ab}$에서처럼 비공변 연산자가 작용한 경우에는 $A_{a_b_}\partial_c S_{a_b_}$로 만들 수는 없다. 동일한 이유 때문에 $A_a{}^b \partial_c S^a{}_b$도 아래 첨자로 만들 수 없다. 결과적으로 패턴 매칭에서 $A_{a_b_}\partial_c S^{a_b_}$와 $A_{b_a_}\partial_c S^{b_a_}$는 일치하고 A와 S가 각각 반대칭과 대칭 텐서이기 때문에 $A_{ab}\partial_c S^{ab}$는 0이 된다. 그런데 $A_{a\_}{}^{b\_}\partial_c S^{a\_}{}_{b\_}$ 와 $A^{b_}{}_{a_}\partial_c S_{b_}{}^{a_}$는 패턴 매칭에서 일치하지 않으므로 $A_a{}^b \partial_c S^a{}_b$는 0이 아니다.

이제 두 텐서를 비교하는 일만 남았다. 먼저 두 텐서의 이름이 동일한지 확인하고, 다르면 더 이상의 비교를 중지한다. 이름이 동일하면 두 텐서의 인덱스만 패턴 매칭으로 비교한다. 이 때 두 텐서의 프리 인덱스가 서로 다르면 적합한 텐서합이 아니므로 적절한 오류를 출력하며 단순화 과정을 중지한다.

프리 인덱스가 동일하면 i 번째 텐서의 더미 인덱스들을 (가능하면) 아래 첨자로 만들고, j 번째 텐서의 더미 인덱스들은 패턴 변수로 만든 후 Mathematica의 MatchQ 함수를 이용하여 두 인덱스들을 비교한다. 만일 일치하면 두 텐서를 합치고, 일치하지 않으면 i 번째 텐서에 대칭을 적용하여 얻어진 새로운 순서의 인덱스 표현으로 다시 j 번째 텐서와 패턴 매칭으로 비교한다. 이러한 과정을 i 번째 텐서에 대칭을 적용하여 얻을 수 있는 모든 순서의 인덱스 표현에 대해서 반복하여, 일치하면 j 번째 텐서와 합치고 일치하지 않으면 그대로 둔다. 최종적으로, 남아있는 텐서들을 다시 원래의 텐서곱의 합으로 변환하면 단순화 과정이 끝난다.

패턴 매칭을 이용한 텐서 표현의 단순화는 수작업을 모델로 하였지만 대칭이 큰 표현은 매우 비효율적으로 동작한다. 예를 들면, 리만 텐서로 구성된 표현

$$R_{ab}{}^{cd} R_{efcd} R^{abef} + R_{abcd} R_{ef}{}^{cd} R^{abef} \to 2 R_{abcd} R_{ef}{}^{cd} R^{abef}$$

의 단순화에 (동작 속도 3.4GHz의 표준 PC에서) 대략 15초 이상 소요된다. V절에서 묘사한 정규화를 이용한 단순화 방법을 사용하면 무시할만한 시간이 소요된다는 것과 명백히 비교된다. 이러한 비효율성의 가장 큰 원인은 대칭을 적용하여 얻을 수 있는 모든 인덱스에 대하여 매번 반복해서 패턴 매칭으로 비교하였기 때문이다. [11]

MathTensor [6]에서는 패턴 매칭 방법의 비효율성을 피하기 위해 리만 텐서에 대한 수 많은 규칙을 사용하였다. 그러나 규칙 기반의 패턴 매칭 방법도 사용하는 규칙이 많아질수록 연산 효율이 떨어진다. 더군다나 리만 텐서가 아닌 다른 텐서에 대한 규칙은 따로이 도입해야 하는 어려움이 있다.

결국 텐서 표현의 단순화에서 발생하는 어려움은 텐서 자체의 대칭, 연산자의 연속 작용에 의한 대칭, 텐서곱에서 두 개 이상의 동일한 텐서에 의한 교환 대칭, 더미 인덱스에 의해 유도된 대칭 등으로 가능한 인덱스 표현의 가짓수가 급격히 증가하여 컴퓨터의 연산 시간과 필요한 메모리 용량을 폭발적으로 증가시키기 때문이다.[2] 현재까지 이러한 문제를 해결하는 최선의 방법은 다음 절에서 논의할 유한 순열군 이론을 적용한 정규화 방법뿐이다.

[2] 패턴 매칭 단순화에서는 더미 인덱스와 관련한 작업을 패턴 매칭으로 처리하기 때문에 더미 인덱스가 새로운 대칭을 만들지는 않는다.

V. 정규화를 이용한 단순화

패턴 매칭을 이용한 단순화는 n 개의 텐서곱으로 구성된 텐서 표현에서 두 개를 선택하여 비교하는 작업을 반복하기 때문에 $O(n^2)$의 복잡도를 갖지만, 정규화를 이용한 단순화는 각각의 텐서곱을 정규화하는 것만으로 충분하기 때문에 $O(n)$의 복잡도를 갖는다. 한편, 패턴 매칭 방법에서는 순열 대칭으로 변환 가능한 모든 인덱스를 고려하였지만, 정규화 방법에서는 대칭 군의 coset representative를 사용하여 고려해야 할 인덱스의 가짓수를 현저히 감소시킨다. [12–14] 이러한 이유 때문에 정규화를 이용한 단순화의 실행 효율이 패턴 매칭 방법에 비해서 상대적으로 매우 높다.

텐서 인덱스의 대칭은 크게 두 가지 종류로 분류할 수 있다. 한 가지는 텐서 자체의 대칭, 연산자의 연속적인 작용에 의한 대칭, 동일한 텐서 이름에 의한 교환 대칭을 병합한 순열 대칭이다. 예를 들면, 리만 텐서의 순열 대칭은 $R_{abcd} = -R_{bacd} = -R_{abdc} = R_{cdab}$를 만족한다. 다른 한 가지는 더미 인덱스에 의해 생겨난 대칭이다. 예를 들면, 텐서 $T_{abc}{}^{b}$에서 더미 인덱스인 아래 첨자 b와 위 첨자 b는 서로 교환된다: $T_{abc}{}^{b} = T_{a}{}^{b}{}_{cb}$.

더미 인덱스가 항상 새로운 교환 대칭을 만들지는 않는다. 예를 들면,

$$\begin{aligned} v_b \nabla_a v^b &= v_b \nabla_a (g^{bc} v_c) = v^b \nabla_a v_b \\ v_b \partial_a v^b &= v_b \partial_a (g^{bc} v_c) \neq v^b \partial_a v_b, \end{aligned} \tag{3}$$

즉 비공변 연산자가 존재하면 더미 인덱스로 인한 교환 대칭이 생겨나지 않을 수 있다. 따라서 인덱스의 '메트릭상태'를 고려하여 교환 대칭을 만드는 더미 인덱스와 그렇지 못하는 더미 인덱스를 구별해야 한다. xTensor [16]에서는 이 과정을 놓쳐서 $A_a{}^b \partial_c S^a{}_b$를 0으로 만드는 오류를 범하였다.

순열 대칭과 더미 인덱스에 의한 대칭이 동시에 존재할 때 텐서 인덱스를 정규화하는 'Butler-Portugal' 알고리즘 [12–14]은 VI절에서 보듯이 특별한 경우를 제외한 대부분의 경우 다항식의 복잡성을 갖는다. [14]

정규화 과정은 III절에서 설명하였듯이 텐서곱을 확대된 대칭을 갖는 하나의 텐서로 병합할 수 있으므로, 결국 한 개의 텐서를 주어진 대칭으로 어떻게 정규화하느냐만 묘사하면 된다. [14] 그 과정을 간략히 요약하면 다음과 같다:

1. 텐서의 인덱스를 특정한 순서로 정렬한다. [16,21] 이 때 '메트릭상태' 정보를 이용하여 교환 대칭을 만들지 못하는 더미 인덱스는 그 위치를 고정시킨다. [21]

2. 정렬된 인덱스를 원래의 인덱스로 변환하는 대칭 군의 원소를 구한 후, 'Butler-Portugal' 알고리즘을 구현한 'CanonicalPerm' 함수를 호출한다. [15,21]

3. 절차 2에서 구한 coset representative로 정렬된 인덱스를 변환하여 정규화된 인덱스를 얻는다.

4. 정규화된 인덱스를 갖는 병합된 텐서를 원래의 텐서곱으로 변환한다.

절차 1에서 '메트릭상태' 정보를 이용하는 이유는 xTensor [16]에서 범한 오류를 방지하기 위한 것이다. 그리고 절차 3과 4는 단순한 과정이고 실행 시간도 거의 무시할 수 있다. 실제로 정규화 과정에서 가장 많은 시간을 소요하는 과정은 절차 2에서 'CanonicalPerm' 함수가 실행되는 과정이다. 따라서 xPerm [15]에서는 실행 효율을 극대화하기 위해 'CanonicalPerm' 함수를 C 언어로 구현하였다.

mTensor [21]의 'CanonicalPerm' 함수는 xPerm [15]과 세 가지 측면에서 차이가 있다. 첫째, 운영체제에 따른 효율성의 차이를 줄이기 위해 2017년 표준안 [22]을 따르는 C++ 언어로 구현하였다. 그 결과 윈도우 운영체제에서 실행 효율이 증가하였음을 VI절에서 볼 것이다. 둘째, 정규화에 필요한 strong generating set을 얻기 위한 방법인 'Schreier-Sims' 알고리즘 [26] 구현에 Mathematica 내장 함수를 사용하였다. xPerm에서 구현한 알고리즘은 평균적으로 평범한 실행 효율을 갖지만 Mathematica 내장 함수인 GroupStabilizerChain 함수는 매우 빠른 속도로 strong generating set을 구해 준다. 끝으로, 정규화 과정의 일관성을 확인하는 절차에 이진 검색 알고리즘을 사용하였다. 결과적으로 VI절에서 보듯이 정규화의 실행 효율이 전반적으로 향상되었다.

VI. 성능 비교

IV절의 패턴매칭을 이용한 텐서 표현의 단순화 방법은 리만 텐서 세 개의 곱으로 된 표현을 단순화하는 간단한 작업조차 상당한 시간이 소요되어 실질적인 사용이 곤란할 정도이다. 따라서 이 절에서는 정규화 방법만을 사용한 단순화 성능을 xPerm [15]과 mTensor [21] 사이에서 비교할 것이다. 또한, 윈도우 운영체제와 리눅스 운영체제에 따른 차이도 알아볼 것이다. 비교를 위하여 동일한 표준 PC (동작 속도 3.4GHz, 메모리 16G)를 사용하였고, 운영체제는 Windows 10과 Mint 20.1을 사용하였다.

일반 상대론 분야에서 가장 중요한 텐서는 리만 텐서이므로 먼저 리만 텐서의 곱으로 구성된 표현의 정규화 소요 시간을 비교한다. [14,15] 이 때 인덱스는 무작위로 선택되고, 최악의 경우를 한정하기 위해 모든 인덱스가 축약된 상황만을 다룬다. (프리 인덱스가 많을수록 더미 인덱스가 만드는 대칭이 줄어 들어 정규화 소요 시간이 적어진다.)

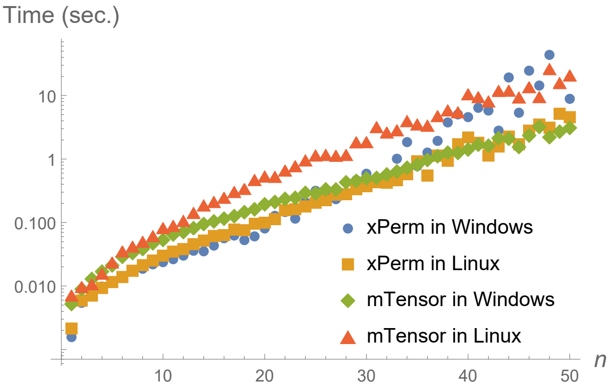

Fig. 1. (Color online) Average timings in a logarithmic axis for canonicalization of completely contracted products of n Riemann tensors with random indices.

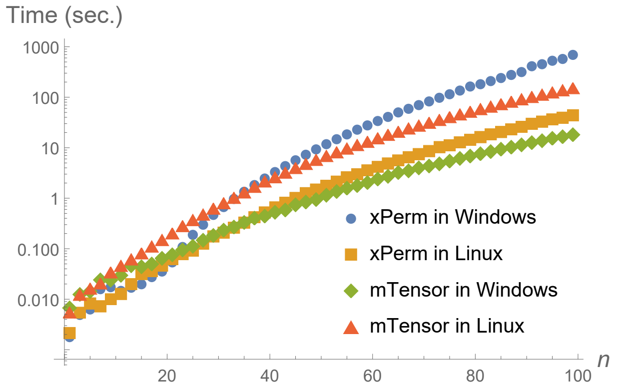

Fig. 2. (Color online) Timings in a logarithmic axis for canonicalization of completely contracted products (4) of *odd* n antisymmetric tensors.

Fig. 1에는 모든 인덱스가 축약된 최대 50개의 리만 텐서 곱에 대한 정규화 소요 시간이 묘사되었다. 각각의 리만 텐서 곱에 인덱스를 무작위로 선택하여 정규화 소요 시간을 측정하는 과정을 20번 반복한 후 기하 평균을 구한 것이다.

Fig. 1을 보면 윈도우에서의 mTensor와 리눅스에서의 xPerm이 유사한 점근 형태를 가지며 좋은 성능을 보이고, mTensor는 리눅스에서 그리고 xPerm은 윈도우에서 상대적으로 비효율적임을 볼 수 있다. 각각의 경우에 해당하는 점근적 적합 곡선을 보면

$$\text{xPermWin} \sim O(n^{6.75}),\ \text{xPermLinux} \sim O(n^{4.57})$$
$$\text{mTensorWin} \sim O(n^{3.52}),\ \text{mTensorLinux} \sim O(n^{4.33})$$

으로 다항식 복잡도를 갖는다. 윈도우에서의 xPerm은 비교적 차수가 높은 다항식 표현이고, 실제로 n이 증가함에 따라 소요 시간이 급격히 증가함을 볼 수 있다.

두 번째 예로 반대칭 텐서 A_{ab}의 곱으로 구성된, 모든 인덱스가 축약된 표현

$$A_a{}^b A_b{}^c A_c{}^d A_d{}^e \cdots A_p{}^q A_q{}^a \tag{4}$$

의 정규화 소요 시간을 비교한다. [15] 이 경우는 인덱스의 무작위성이 없고, 텐서 A_{ab}의 개수 n이 홀수이면 0이 된다. Fig. 2에는 모든 인덱스가 축약된 최대 99개의 반대칭 텐서 A_{ab}의 (홀수 개의) 곱으로 구성된 표현에 대한 정규화 소요 시간이 묘사되었다. 전체적인 결과는 Fig. 1과 유사하다. 즉 윈도우에서의 mTensor와 리눅스에서의 xPerm이 상대적으로 좋은 성능을 보인다. 특히 윈도우에서 xPerm은 n이 증가함에 따라 소요 시간도 급격히 증가한다. 각각의 경우에 해당하는 점근적 적합 곡선을 보면

$$\text{xPermWin} \sim O(n^{6.14}),\ \text{xPermLinux} \sim O(n^{4.87})$$
$$\text{mTensorWin} \sim O(n^{4.17}),\ \text{mTensorLinux} \sim O(n^{4.72})$$

으로 다항식 복잡도를 갖는다. 결과적으로 리눅스에서 mTensor와 xPerm이 유사한 성능을 보이고, 윈도우에서는 mTensor가 xPerm 보다 월등히 좋은 성능을 보인다.

세 번째 예로 반대칭 텐서 A_{ab}와 대칭 텐서 S_{ab}의 곱이 반복되면서 모든 인덱스가 축약된 표현을 고려하자: [27]

$$A_{ab} S^{bc} A_{cd} S^{de} \cdots A_{pq} S^{qa}. \tag{5}$$

이 경우는 운영체제와 상관없이 xPerm에서는 텐서의 개수가 $n = 12$일때부터, 그리고 mTensor에서는 $n = 14$일때부터 소요 시간이 지수적으로 증가한다. 따라서 텐서 표현 (5)의 정규화 문제는 'Butler-Portugal' 알고리즘의 최악의 경우에 해당한다. 그러나 이렇게 특수한 경우는 휴리스틱 방법을 쓸 수 있다. 즉 텐서 표현 (5)의 패턴을 보면 새로운 텐서 AS_{ab}를 도입하고 규칙 $A_{ab}S^{bc} \to AS_a{}^c$를 사용할 수 있음을 알 수 있다. 그러면 텐서 표현 (5)는 텐서 표현 (4)와 동일한 표현이 되고, 정규화도 다항식 복잡도를 갖게 된다.

VII. 결론 및 토론

인덱스를 갖는 텐서 표현의 단순화 전략으로 패턴 매칭 방법과 정규화를 이용한 방법을 논의하면서, 인덱스의 '계열'과 비공변 연산자로 인한 자료구조인 '메트릭상태'를 도입하였다. 그 결과 다양한 인덱스를 가지면서 공변 연산자뿐만 아니라 비공변 연산자가 포함된 텐서 표현도 오류없이 단순화할 수 있음을 설명하였다.

패턴 매칭을 이용한 단순화 방법 [6,11]은 실제로 사용하기 곤란한 정도의 시간이 소요되므로 순열 대칭에 대한 군론을 이용한 'Butler-Portugal' 알고리즘으로 정규화하는 방법 [12–14]을 주로 논의하였다.

정규화 과정의 연산 효율을 극대화하기 위해 'Butler-Portugal' 알고리즘을 C언어로 구현한 xPerm [15]은 관련된 많은 연구에서 사용되고 있지만 [16–20] 여러 가지 단점도 가지고 있다. 특히 윈도우 운영체제에서 인덱스 대칭이 커짐에 따라 정규화 성능이 급격히 떨어진다. 이러한 문제점을 개선하여 mTensor [21]에 구현하였고, xPerm과 정규화 성능을 비교하였다.

본문에서는 설명하지 않았지만 게이지장 $F_{\mu\nu}{}^{a}$ 처럼 여러 가지 인덱스 '계열'이 혼합된 텐서 표현이나, 스피너 인덱스처럼 반대칭 계량 텐서를 갖는 텐서 표현도 동일한 방법으로 정규화한다. 또한, $F_{1}{}^{1}$ 처럼 숫자로 표시되는 성분 인덱스를 갖는 텐서 표현도 마찬가지 방법으로 정규화한다.

'Butler-Portugal' 알고리즘은 최악의 경우 연산 시간이 지수적으로 증가한다. VI절에서 보듯이 규칙 기반의 휴리스틱 방법으로 특별한 경우는 다룰 수 있지만 일반적인 경우에는 아직 한계가 있다. [27] 또한, 단일항인 텐서곱의 정규화만 논의하였지만 비앙키 항등식과 같은 다중항으로 표현되는 텐서 대칭도 고려해야 한다. [28]

Mathematica에는 'Butler-Portugal' 알고리즘을 구현한 내장 명령이 있어서 xPerm이나 mTensor처럼 외부에서 작성한 코드를 연동해야 하는 불편함을 피할 수는 있다. 그러나 이 내장 명령은 인덱스 없는 추상 텐서 표기법을 사용하였고, 부가적인 자료구조를 도입하지 않는 한 계량 텐서가 대칭인 경우만 다룰 수 있다. 따라서 내장 명령만을 사용하여 본 논문에서 논의한 인덱스 표현의 단순화 기능을 구현하는 것도 하나의 도전 과제가 될 수 있다. 또는, 반대쪽 방향으로, Mathematica의 내장 명령으로 구현된 부분을 모두 C++ 코드로 바꾸어 특정 소프트웨어에 대한 종속성을 제거하는 것도 의미가 있을 것이다.

감사의 글

이 논문은 2019년도 상지대학교 교내 연구비 지원에 의한 것입니다.

REFERENCES

[1] M. A. H. MacCallum, Living Rev. Rel. **21**, 6 (2018).
[2] K. Lake, in *Proceedings of the 15th International Conference on General Relativity and Gravitation*, edited by N. Dadhich and J. Narlikar (1997), p. 421.
[3] V. Toth, *Tensor manipulation in GPL Maxima*, https://arxiv.org/abs/cs/0503073
[4] R. A. Bogen and R. Pavelle, Lett. Math. Phys. **2**, 55 (1977).
[5] S. A. Fulling, R. C. King, B. G. Wybourne, and C. J. Cummins, Class. Quantum Grav. **9**, 1151 (1992).
[6] L. Parker and S. M. Christensen, *MathTensor: A system for doing tensor analysis by computer* (Addison-Wesley, 1994).
[7] V. A. Ilyin and A. P. Kryukov, Comput. Phys. Commun. **96**, 36 (1996).
[8] R. Portugal, Comput. Phys. Commun. **115**, 215 (1998).
[9] A. Balfagón and X. Jaén, *Simplifying Tensor Polynomials with Indices*, https://arxiv.org/abs/gr-qc/9809022v2
[10] R. Portugal, J. Phys. A: Math. Gen. **32**, 7779 (1999).
[11] D. H. Park, *Indicial Tensor Package Using Notebook Interface*, https://library.wolfram.com/infocenter/MathSource/755/
[12] G. Butler, in *Proceedings of Computational Group Theory*, edited by M. D. Atkinson (Academic Press, 1984), p. 283.
[13] A. Ya. Rodionov and A. Yu. Taranov, in *EUROCAL '87, Proceedings of the European Conference on Computer Algebra*, edited by J. H. Davenport (1989), p. 192.
[14] L. R. U. Manssur, R. Portugal, and B. F. Svaiter, Int. J. Mod. Phys. C, **13**, 859 (2002).
[15] J. M. Martin-Garcia, Comput. Phys. Commun. **179**, 597 (2008).
[16] J. M. Martin-Garcia, *xTensor*, http://www.xact.es/xTensor/index.html
[17] A. García-Parrado, J. M. Martín-García, Comput. Phys. Commun. **183**, 2214 (2012).
[18] K. Peeters, J. Open Source Software, **3**, 1118 (2018).
[19] C. Cheung and M. P. Solon, Phys. Rev. Lett. **125**, 191601 (2020).
[20] J. Noller, L. Santoni, E. Trincherini, and L. G. Trombetta, J. Cosmol. Astropart. **01**, 045 (2021).
[21] D. H. Park, *mTensor*, https://github.com/dhpark00/mTensor,
[22] *ISO/IEC 14882:2017 Programming languages — C++*, https://www.iso.org/standard/68564.html, (accessed Mar. 10, 2021).

[23] R. Penrose and W. Rindler, *Spinors and space-time I: two-spinor calculus and relativistic fields*. (Cambridge University Press, Cambridge, 1984), chap. 2.

[24] A. Ashtekar, G. T. Horowitz and A. Magnon-Ashtekar, Gen. Rel. Grav. **14**, 411 (1982).

[25] R. M. Wald, *General Relativity* (The University of Chicago Press, Chicago and London, 1984), chap. 2 and chap. 3.

[26] G. Butler, *Fundamental Algorithms for Permutation Groups*, (Springer-Verlag, Berlin, 1991), chap. 13.

[27] B. E. Niehoff, Comput. Phys. Commun. **228**, 123 (2018).

[28] Hongbo Li, Zhang Li, and Yang Li, in *ISSAC '17: Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, edited by M. Burr, (2017), p. 269.