`libPerm`

Robert Adam

last edited: March 27, 2023

Contents

1. Motivation

In another project, we needed to deal with the permutational symmetries of indices of tensor elements that take the form of e.g.

$$T^{ab}_{cd} = -T^{ab}_{dc} = -T^{ba}_{cd} = T^{ba}_{dc} \tag{1}$$

In order to be able to perform any kind of automatic simplifications on expressions involving tensor elements with such symmetries, it is clear that a way to determine if two tensor elements are equivalent (in the sense that they can be written in the form $A = \pm B$) is necessary.

The best approach for dealing with symmetry-related questions like that, is to turn to group theory. Therefore, we needed a library to properly deal with permutation groups. Since the main project was in C++, so should that library. A search on the web produced [1] that seemed to fit the purpose perfectly. Unfortunately, this library has become quite outdated and does not even compile on somewhat modern systems. However, its author has written his thesis about the implementation[2] and in there were suggestions for how to treat permutation groups efficiently. Thus, we went ahead and created our own implementation.

2. Notation

Table 1 summarizes the symbols and concepts used throughout this work (unless noted otherwise). We assume the reader is generally familiar with group theory basics and thus will not include a full introduction here.

Furthermore, note that we start enumerations at zero rather than at one (unless noted otherwise).

Table 1: Common notation and concepts used throughout this document

Notation	**Concept**	**Def. / Ref.**
$\mathrm{Sym}(n)$	Symmetric group of n elements	[3]
P, G, H	General (permutation) group	
p, g, h	An element of the respective group	
$\alpha, \beta, \ldots$	Concrete integer number ($\alpha, \beta, \ldots \in \mathbb{Z}_0$)	
α^g	The image of α under the action of g	
$\lvert G\rvert$	The order (amount of elements contained) of the group G	
$gh = g \cdot h$	Product of g and h. Please see section 3 for what exactly this means.	
gH	The left coset of g with H. Note that (usually) $g \in G$ and $H \subseteq G$	$\{gh \lvert \forall h \in H\}$
Hg	The right coset of g with H.	$\{hg \lvert \forall h \in H\}$
id	The identity permutation	

3. Applying permutations and order of composition

When we "multiply" two permutations a and b, we create a new permutation $c = a \cdot b$. This new permutation is then the *composition* of its "factors". Composing two permutations effectively means first applying one of them and then applying the other. The only question remaining is this: given $c = a \cdot b$, do we first apply a or do we first apply b?

Someone coming from maths will immediately think of function composition $h = g \circ f$, where the composition is understood to mean $h(x) = g(f(x))$. That is, the order of applying the functions is

from right to left. However, when dealing with permutations, it is common to instead use a left-to-right interpretation of composing two permutations, such that $a \cdot b$ is understood as first applying a and then applying b. Thus, composing permutations works left-to-right.

However, this is only valid as long as the permutations are only applied to a single integer. This, by definition, gives the image of that number under the acting permutation α^a. There is also the possibility to let a permutation act on an entire sequence of elements, in which case the elements in the sequence shall be rearranged according to the permutation. Since the elements in that sequence don't have to be integers, it is clear that the permutation must somehow act on the indices of the elements instead of the elements itself. But still, there are two possible ways to do that:

1. The α-th element in the original sequence will be moved to position α^a in the rearranged sequence
2. The α-th element in the rearranged sequence will be the α^a-th element of the original sequence

We choose to use the second interpretation as this is the one in which a sequence of integers $[\overset{0}{0}, \overset{1}{1}, \overset{2}{2}, \overset{3}{\ldots}]$ will be mapped to $[\overset{0}{0^a}, \overset{1}{1^a}, \overset{2}{2^a}, \overset{3}{\ldots}]$ when acting on it with the permutation a.

If we call our sequence s and denote the α-th element of the sequence as $s[\alpha]$, we can formalize what it means for a permutation p to act on s:

$$s^p[\alpha] = s[\alpha^p] \tag{2}$$

If we instead apply two permutations to the sequence one after the other, say first a and then b, the situation would look as follows:

$$s^{ab}[\alpha] = (s^a)^b\,[\alpha] = s^a[\alpha^b] \overset{\beta=\alpha^b}{=} s^a[\beta] = s[\beta^a] = s[(\alpha^b)^a] = s[\alpha^{ba}] \tag{3}$$

Therefore, it can be seen that the meaning of the composite permutation ab is exactly reversed when acting on a single integer (α) vs. when acting on a sequence (s). Since permutations are by definition defined as a bijection of a set of numbers onto itself, the composition order of applying a permutation to an integer, is the "natural" composition order that we chose. Therefore, while composition of permutations acting on integers are ordered left-to-right, permutations acting on sequences act right-to-left.

An example where this comes into play is if you have two permutations a and b and apply them to your object individually, first a and then b, and then want to construct a composite permutation c that performs these two steps in a single action. If your object is an integer, then $c = ab$, but if your object is a sequence that you want to permute, $c = ba$.

Composition order

- When acting on integers, permutations compose left-to-right
- When acting on sequences, permutations compose right-to-left

4. Usage

Might come eventually. For now, see the examples directory in the source repository.

5. Implementation

The focus of our implementation is to arrive at a fast and efficient library primarily aimed at dealing with small permutation groups $P \subseteq \mathrm{Sym}(n), n \sim 10$. In order to do this, the library tries to avoid dynamic memory allocations wherever feasible (another big difference to [1]). The library is implemented according to the ISO C++17 standard.

C++ comes with the restriction that when working with objects through their base-class interface, it is (without tricks) impossible to e.g. create a copy of the object without resorting to dynamic memory allocations[I]. However, by assuming that we will never have to deal with implementations of our interfaces from outside our library, we can assume that we know all possible implementing classes of a given interface at compile-time. This special case of polymorphism is called *closed-set* polymorphism and it allows us to use a trick leveraging the `std::variant` class in the standard library. The details of this are not important here, but the effect is that this allows us to treat polymorphic objects as *value-types*[II]. For this purpose, we have implemented another library `polymorphic_variant`[4] that encapsulates all the unmentioned details of this trick.

In the following, if we refer to an interface being *unwrapped* into a new type, that corresponds to creating a `polymorphic_variant` wrapper for the closed-set polymorphism for the given interface (aka: collect all existing implementations of that interface into a single wrapper type).

The only thing to be aware of, is that you'll have to use the arrow operator (`->`) to access member functions for unwrapped types. E.g. `perm->isIdentity()` instead of `perm.isIdentity()`.

5.1. Representing permutations

The most fundamental object that a library dealing with permutation groups has to have, is a way to represent a single permutation. There are multiple possible ways of achieving this task, each having their distinct advantages and disadvantages.

Since this is a C++ library, permutations are represented as the bijection of a set S onto itself, where the points in S start at zero, rather than at one (as is common practice when working with permutations in e.g. a maths context).

5.1.1. `AbstractPermutation`

In order to be able to be able to test or switch out the used representation in `libPerm`, we created the `AbstractPermutation` interface. This interface covers all functionality that the library or any user of the library is expected to require in order to work with a given permutation. This covers basic operations like comparing for (in)equality, computing the image of a point under the represented permutation, checking for identity, inverting the permutation in-place and of course to multiply two permutations.

One consequence of choosing runtime polymorphism in order to allow for multiple implementations of permutation representations is that the general interface can't include any operation that itself returns a permutation object. The reason for this is that such an object would either have to be allocated dynamically (which we want to avoid) or the function signature in the interface would have to enforce one specific implementation as the return value (instead of returning the by interface)[III]. As a result,

[I] The reason for this is that the compiler needs to know the exact size of a given object, if it is to be created on the stack and in terms of polymorphic types, the size of the actual underlying object is only determined at runtime.

[II] Types that can be moved around, copied and stored (in containers)

[III] The underlying reason for this is again that the compiler can't deduce the actual size of the respective object

the general interface does not include a binary multiplication operator, that takes two permutations and creates a new one. Instead, only the multiply-assign (in-place multiplication) operator is provided, that multiplies two permutations `A` and `B` and then sets `A` to the result of the multiplication (thereby avoiding the need to create a new permutation object).

In contrast to the classical definition of permutations, the `AbstractPermutation` interface also allows permutations to carry a sign information. This property is completely orthogonal to the underlying permutation itself. However, it is transformed accordingly when e.g. multiplying or inverting permutation objects. The reason for adding this additional property is that dealing with signed permutations is required in the context of tensor symmetries and it is a lot easier to just integrate that into the main interface than injecting this property by other means.

5.1.2. `Permutation`

The `Permutation` "class" is simply the result of *unwrapping* the `AbstractPermutation` interface.

5.1.3. `ExplicitPermutation`

The `ExplicitPermutation` class is a concrete implementation of the `AbstractPermutation` interface. It stores a permutation as a list of points, where the ith point corresponds to the image of i under the represented permutation (`list[`α`] =` α^g, where g is the represented permutation). Note that this class will always explicitly store the images of the set $\{0, \ldots, n\}$, where n is the biggest element affected by the represented permutation. Always explicitly representing the images of (potentially unchanged) lower points avoids excessive branching in the remaining implementation.

The idea for using this representation is taken from Rehn [2] (they called this form "elementary permutations"). As noted there, this representation has the advantage of providing access to the image of a point α under the represented permutation g in $\mathcal{O}(1)$. Multiplying an `ExplicitPermutation` with another (arbitrary) `Permutation` can be performed by algorithm 1. If we assume that looking up the image of a point under the `Permutation` is $\mathcal{O}(1)$ (which is the case, if e.g. multiplying two `ExplicitPermutation` with one another), then this multiply operation can be performed in $\mathcal{O}(n)$[I].

Algorithm 1 Algorithm for multiplying an `ExplicitPermutation` with an arbitrary `Permutation`. Complications arising from the second permutation acting on a larger set of points that the first one are left out for brevity.

```
1: function MULTIPLY(ExplicitPermutation a, Permutation b)
2:     for all image points α in a do
3:         α ← α^b
4:     end for
5: end function
```

Inverting an `ExplicitPermutation` can be performed by algorithm 2, which also scales as $\mathcal{O}(n)$.

One big advantage of using a direct storage of the image points over e.g. "permutation words"[2] is that operating on the former can be expected to be cache-friendlier due to the avoidance of a linked-list-like structure[II].

[I] This is in contrast to what is stated in Rehn [2] who state that such multiplication can only be performed in $\mathcal{O}(n^2)$.
[II] At least for permutations acting on a reasonably small (probably < 100) set of points.

Algorithm 2 Algorithm for inverting an `ExplicitPermutation`.

```
1: function INVERT(ExplicitPermutation p)
2:     inverted: List for the inverted image points
3:     for α ∈ [0, length of inverted) ∧ α ∈ ℕ_0 do
4:         inverted[α^p] ← α
5:     end for
6:     Use inverted as new image points
7: end function
```

5.2. Representing permutation groups

A permutation *group* is just a collection of individual permutations, that together form a *group* (in the mathematical sense). The most trivial way of representing a permutation group is therefore to just represent it as a list of `Permutation` objects. However, this incurs some disadvantages when working with the group. One disadvantage is that the order of a permutation group increases very rapidly (the order of $\mathrm{Sym}(n)$ is $n!$) and therefore, explicit storage can quickly become costly and eventually impossible. Therefore, permutation groups are often represented in a compressed way by means of a *base* and a *strong generating set* (together: BSGS) as obtained by the so-called *Schreier-Sims algorithm* (for details see e.g. refs. [2, 5]).

5.2.1. `AbstractPermutationGroup`

Similarly to the `AbstractPermutation` interface, we also created a general interface for a general permutation group, which we called `AbstractPermutationGroup`. The interface tries to cover common operations one would like to perform on a permutation group, such as computing the *orbit* of a point α under the group P. Other supported operations include getting the group's order, performing membership tests[I], adding a (new) generator for the group, recomputing the group from a given set of generators, explicitly obtaining a list of all elements of the group and getting the *canonical coset representative* (see below).

Canonical coset representative Any element contained in a coset, can be chosen as a *coset representative*. However, it is beneficial if the act of choosing such a representative is defined in such a way that, given a coset, the chosen representative will always be the same (regardless of e.g. the order in which the coset's elements are in). If this is fulfilled, the chosen representative will be called a *canonical* coset representative.

5.2.2. `PrimitivePermutationGroup`

The `PrimitivePermutationGroup` is an implementation of the `AbstractPermutationGroup` interface, that essentially follows the trivial approach outlined before: it stores the elements of the group explicitly in a list. Additionally, the group maintains a separate list of generators. The latter turned out to be beneficial when e.g. trying to extend a group with a new generator.

The group is intended to be constructed from a set of permutations that *generate* the group. Since we want to represent the group by explicitly storing all of its elements, one fundamental task in this implementation deals with generating these elements from a set of generators. The most efficient way (known so-far) to accomplish this to use *Dimino's algorithm*.[5] A detailed "derivation" of this algorithm

[I] Answering the question "Does this group contain permutation p?"

can be found in Butler [5] and a short overview can be found in appendix A. This algorithm (by its very nature) can also be used to extend a given group with a new generator.

Computing the orbit of a point α under the represented group P is straight forward as we have access to all elements in the group (see algorithm 3). The complexity of this algorithm is $\mathcal{O}(|P| \cdot L)$, where L is the cost of performing the lookup to check whether α^p is already contained in `orbit`. Depending on the data structure used for `orbit`, this lookup could be anything between $\mathcal{O}(1)$ and $\mathcal{O}(N)$, where N is the size of `orbit`.

Algorithm 3 Computing the orbit of a point α under the action of a `PrimitivePermutationGroup` P

```
1: function ORBIT(PrimitivePermutationGroup P, point α)
2:     orbit: List of points in the orbit of α
3:     for all p ∈ P do                          ▷ Remember that id ∈ P
4:         if α^p ∉ orbit then
5:             Add α^p to orbit
6:         end if
7:     end for
8:     return orbit
9: end function
```

In our interface, we have chosen to represent orbits by means of a `std::vector`. A semantically better choice would have been to use a `std::set` or `std::unordered_set` as these would, by design, ensure that the set can't contain any duplicates and also provide fast checks for whether a given element is already contained. However, they require more memory accesses when inserting new elements (`std::set`) or in general more memory to represent the same set (`std::unordered_set`). The latter also necessarily stores the set elements in a less cache-friendly way. Since the orbits that we expect to encounter should only consist of a few elements, the linear search required for the membership test in `orbit` in algorithm 3 should not incur too much overhead and therefore it seemed beneficial to stick to `std::vector`. The additional cost might even be ameliorated by `std::vector` being very cache-friendly.

Getting the group's order and performing membership tests is also very straight forward, since we maintain an explicit element list. Thus, the order is just the size of that list and the membership test can be performed by a simple linear search. The latter could be improved upon, by keeping the elements in a sorted list instead of in random order. Then we could perform a binary search, which would scale as $\mathcal{O}(\log(N))$ instead of $\mathcal{O}(N)$ for the linear search. However, this would introduce additional overhead when constructing or modifying the list of elements as all operations must keep the list sorted. Since we don't yet have evidence that the linear search causes problems in applications of this class, we decided to stick with an unordered list and a linear search for the time being.

The approach for finding a canonical coset representative has been taken from Manssur *et al.* [6], but adapted to work with explicit element storage instead of representing the group as a BSGS. This approach requires to establish a total ordering relation $\prec$ between the elements in a coset.[6] This ordering will be defined relative to a list of numbers $\mathbf{b}$, which we simply chose to be $\mathbf{b} = [0, 1, \ldots]$. More details on how such ordering works can be found in appendix B. With this ordering relation at hand, the canonical coset representative is found by creating all elements in that coset and then finding the minimum element with respect to $\prec$. This minimum element is then the canonical representative.

A. Dimino's algorithm

An excellent step-by-step derivation of this algorithm can be found in Butler [5], but we'll sketch the main points here as well. Starting point for the Dimino algorithm is a set of generators S that generate

a group G, which we'll denote $G = \langle S \rangle$. The algorithm then tries to construct all $g \in G$ in an efficient way.

The fact that S generates G, means that every $g \in G$ can be decomposed into a product of generators (note: generators may appear multiple times in this decomposition)

$$g = \prod_i s_i \qquad\qquad s_i \in S \tag{4}$$

Thus, we can perform inductive steps during the generation of G.[5] Every element g that requires m generators in eq. (4) can be expressed as $g = hs$, where $s \in S$ and $h \in G$ is an element that has been generated by multiplying $m - 1$ generators.[5] This tells us, that when constructing the elements of g, it is sufficient to only consider gs as a potentially new group element, where g is an element that has already been found to belong to G and s is one of the generators.[5]

Dimino's algorithm ensures that the generators $S = \{s_0, s_1, \ldots\}$ are processed sequentially. That means that before considering the ith generator, it first ensures that all possible elements that can be generated by the s_i with $i \in \{0, 1, \ldots, i-1\}$ have been found and added to G.[5] In other words, we first form the complete subgroup that can be generated by the $i-1$ first generators, before generating the elements of the group arising from also considering the ith generator. The subgroup generated by the first k generators, will be denoted H_k

The algorithm makes use of the fact that a group is partitioned by the cosets of any of its subgroups.[5] This means that two cosets are either completely identical or they have no element in common (they are disjoint). A given coset can simply be generated by forming the coset of the subgroup in question with an arbitrary representative of the desired coset[I]. Therefore, if we have generated the subgroup H_{i-1} and want to extend it with the generator s_i, we simply have to check whether s_i is already contained in H_{i-1}. If it is, this generator will not yield any new elements and is therefore redundant. If it is not yet contained yet, we can immediately deduce that the entire coset $H_{i-1}s_i$ is not yet contained and can therefore add it to H_i. If we do add new elements in this way, we'll also have to check whether any of the previously processed generators create yet another coset from the coset that has just been added. If they do, we again add the entire coset as new elements. This has to be performed in a self-consistent until no new cosets are found. Then we'll have generated the entire subgroup H_i and we can move to the next generator s_{i+1} (if any).

In order to avoid having to compute $\{h \cdot s_i | \forall h \in Hs_n, \forall s_i \in S_i\}$, where S_i is the set of generators processed so far and s_n is the just added, new generator, just to find potential new coset representatives, we can instead make use of another coset property. If Hg is a coset of H in G and $s_i \in G$ then all elements of $(Hg)s_i$ lie in the coset of H with the coset representative $g \cdot s_i$.[5] Or in other words: during our search for new coset representatives, we only have to check the products of the old representative with all previously encountered generators, which will save a lot of membership tests.

From the concepts so far, we can assemble the basic version of Dimino's algorithm by additionally noting that eq. (4) implies that when only considering a single generator, then the group generated by that is simply given by that generator's powers.[5] The result is shown in algorithm 4.

Butler [5] mentions an additional optimization of this algorithm, that can be applied every time an encountered subgroup happens to be *normal*. With this, a few more operations can be saved, but at the expensive of having to implement and run a check whether a given subgroup is normal. This optimization has so far not been considered or implemented in `libPerm`.

In the actual implementation of this algorithm it is also possible to leverage the fact, that all cosets of some subgroup H all have the same size: $|H|$.

[I] This is because any specific element can only be contained in a single coset. And since a subgroup is itself a *group*, it contains the identity element and therefore the coset formed in this way will necessarily contain the element that was used to generate the coset.

Algorithm 4 Dimino's algorithm (adapted from Butler [5])

function GENERATEELEMENTS(Set of generators S)
    `elements`: Elements of the to-be-generated group G
    $s_0 \leftarrow$ first element in S
    for all unique powers of s_i (s_i^n) **do** ▷ Generate elements of $G = \langle s_0 \rangle$
        Add s_0^n to `elements`
    end for
    for $i \in [1, \text{size of } S) \wedge i \in \mathbb{N}$ **do** ▷ Extend G step-by-step
        $S_{i-1} \leftarrow$ first $i-1$ entries in S
        $s_i \leftarrow i$th generator in S
        EXTENDGROUP(`elements`, S_{i-1}, s_i)
    end for
end function
function EXTENDGROUP(Elements of group H, old generators S, new generator s_n)
    if $s_n \in H$ **then** ▷ New generator is redundant
        return
    end if
    Add entire coset Hs_n to H
    for all cosets added **do** ▷ Check if any additional cosets to add are found
        $c \leftarrow$ representative of added coset
        for all $s \in (S \cup \{s_n\})$ **do**
            if $c \cdot s \notin H$ **then**
                Add entire coset $H(c \cdot s)$ to H ▷ This creates another iteration in the outer loop
            end if
        end for
    end for
end function

B. Defining ordering between permutations

The process for introducing a total ordering of permutations is taken from Manssur *et al.* [6]. It starts by introducing a list $\mathbf{b} = [b_0, b_1, \ldots]$ of N distinct numbers of the set $\{0, 1, \ldots, n = N-1\}$, where n is the biggest point that is permuted by any of the permutations that are supposed to be compared.

This list of points allows us to define the order of points. $\alpha \prec \beta$, if α comes before β in $\mathbf{b}$ (left-to-right).[6]

We define $\mathbf{b}^p = [b_0^p, b_1^p, \ldots]$ as the image of $\mathbf{b}$ under the permutation p. Such lists of points are compared element-wise, such that $\mathbf{b}^p \prec \mathbf{b}^q$ if for the first (left-to-right) point $\alpha \in \mathbf{b}^p, \beta \in \mathbf{b}^q$ for which $\alpha \neq \beta$, it is $\alpha \prec \beta$.[6]

The relative order between permutations is then simply based on the order of the images of $\mathbf{b}$ under the permutations. For two permutations p, q, we therefore define[6]

$$p \prec q \iff \mathbf{b}^p \prec \mathbf{b}^q \tag{5}$$

Inside `PermLib`, our permutations also carry sign information, but since the same permutation can't be contained in a single group with both possible sign values[6], we can simply ignore the sign when comparing permutations.[6]

We want to put emphasize on the fact that the choice of a specific $\mathbf{b}$ to use for defining order is arbitrary. Certain situations might make it beneficial to select the points and their order in $\mathbf{b}$ in a

specific way, but this has no effect on the concepts used to define ordering. It may, however, affect the specific order established this way. Therefore, it is important to consistently use the same **b** throughout a given (part of a) code.

References

1. https://github.com/tremlin/PermLib (2022-11-25).
2. Rehn, T. *Fundamental Permutation Group Algorithms for Symmetry Computation* Diploma thesis (Otto-von-Guericke University Magdeburg, 2010).
3. https://en.wikipedia.org/wiki/Symmetric_group (2022-11-25).
4. https://github.com/Krzmbrzl/polymorphic_variant (2022-11-25).
5. Butler, G. *Fundamental Algorithms for Permutation Groups* doi:10.1007/3-540-54955-2 (Springer-Verlag, 1991).
6. Manssur, L. R. U., Portugal, R. & Svaiter, B. F. Group-theoretic approach for symbolic tensor manipulation. *Int J Mod Phys C* **13,** 859–879. doi:10.1142/S0129183102004571 (2002).