

Parity Odd Event Variables for Hadron Colliders

Takis Angelides*

Supervisor: Dr. Christopher Lester

May 17, 2021

Abstract

Parity odd functions of hadron collider events can be used to probe novel parity violating processes through asymmetries in their distributions. In this paper we present results used for the construction of such functions. Geometric Algebra algorithms are provided for the computational validation of our results.

Except where specific reference is made to the work of others, this work is original and has not been already submitted either wholly or in part to satisfy any degree requirement at this or any other university.

Contents

1	Introduction	2
2	Theory	2
3	Method	3
4	Results	4
5	Computational test of results using Geometric Algebra	15
6	Geometric perspective	21
7	Conclusion	23
8	Acknowledgements	23
A	From events to event variables	24
B	Proofs for algorithms	24
C	Notation	26

*Email: takis.angelides@gmail.com

D Case 2 calculation for collision events	27
E Code	29

1 Introduction

Following the results of experiments by Wu et al in 1957 [1] which showed that the weak interaction violates parity, tests of parity violation have hitherto been limited, especially on LHC data [2]. Non-standard parity violating processes beyond the weak interactions can manifest themselves through asymmetries in distributions of event variables which are parity odd, applied to data generated by the LHC.

The data consist of events characterised by a set of real non space-like four-momenta. Given our methods are purely data-driven, at no point in constructing the variables do we rely on any theoretical model. Consequently, the method of this paper that follows from [3] aims to probe all sources of non-standard parity violation if any exist.

In this paper we specifically focus on the case of events which consist of two incoming and four outgoing objects, which can be two incoming protons and four outgoing jets. We aim to identify certain properties of the results obtained that can be used to generalise to N outgoing objects. The paper will present conditions for events to be non-chiral - see section 2 for the definition of non-chiral events - with the ambition of building event variables for different types of two to four events in future work.

In the following section we provide the theoretical background for our work, including necessary definitions and conventions. We then present the general method for this paper and our results for each type of events considered. A computational validation of the results follows, which utilizes the elegant power of Geometric Algebra. We give a geometric picture of our work before concluding in the last section. In the appendix, we provide a flowchart showing the method of this paper, proofs for the validity of our algorithms, some important notation used for the results in this paper and the explicit calculation of one of the cases considered below.

2 Theory

Let $S = \{V_i \mid i = [1, N]\}$ be the set of N event variables and $\Omega = \{e \mid e \text{ is chiral}\}$ the set of chiral events. An event is defined as chiral if after applying parity to it, it cannot be mapped onto itself by any combination of group elements from our choice of a symmetry group G . The symmetry group in this paper will be the Lorentz group $\times S_2 \times S_4$, where S_n is the permutation group of n elements. Note that S_2 only permutes incoming particles (if identical) and S_4 outgoing particles. As discussed with detail in [3], S must satisfy the properties of:

- **Sufficiency:** at least one V_i evaluates to a non-zero value for all $e \in \Omega$
- **Necessity/irreducibility:** removing any V_i would violate sufficiency
- **Reality:** $V_i : e \rightarrow \mathbb{R}$

- **Continuity:** small changes - in energy or momenta - to e should lead to small changes in the output of V_i
- **Lorentz and permutation invariance:** V_i should be Lorentz invariant and invariant under permuting identical objects such as jets or photons in incoming or outgoing states
- **Parity-odd:** all V_i change sign under parity $\vec{x} \rightarrow -\vec{x}$
- **Minimality:** if two sets satisfy all properties, we choose the one with the least number of elements.

Our ultimate aim is to construct the sets S_c and S_{nc} for $2 \rightarrow 4$ chiral, collision and non-collision events respectively. The classes of events called collision events \mathcal{E}^c and non-collision events \mathcal{E}^{nc} are given by definitions 2.23 and 2.26 in [3]. Roughly, a collision event has an initial state of two particles that can define a center of mass frame and in that frame the equal and opposite 3-momenta are non-zero. A non-collision event is an event that cannot satisfy these properties.

Intuitively, non-standard parity violating processes will push the average of at least one of the parity-odd event variables towards being very negative or very positive. This is because the variable will change sign if we evaluate it on a chiral event e or on the same event after applying parity to it $\mathcal{P}e$. Hence, if nature does not prefer a handedness for parity and produces equal numbers of left and right handed chiral events of the same process, then the output of the variable will have on average half of the time a positive sign and the other half a negative sign. Averaging over the ‘+1’ and ‘-1’, corresponding to outputs of positive and negative sign, would of course give 0. Now, if nature decides that it prefers a handedness for a given process, then for that class of chiral events the sign of at least one event variable will be the same throughout the sample of events and would drive the average significantly far from 0.

This is why we are not concerned about non-chiral events. Since they can be mapped onto themselves after parity by actions of the symmetry group, they evaluate to 0 on any parity odd function. Hence, they cannot be used to probe non-standard parity violating processes. The idea is that if we have a logic statement that is true for chiral events, then we can use it to construct event variables while ensuring that at least one of the event variables would evaluate to a non-zero value for any chiral event input. We need to enforce the last property so that any chiral event can be labelled left or right handed.

3 Method

The method to obtain S_c and S_{nc} is to first obtain a logic statement that is true when an event is non-chiral - see figure 5. Since we are concerned only with chiral events, we negate the latter logic statement. By having the logic statement which is true for chiral events, we can ensure that both S_c and S_{nc} satisfy the sufficiency condition. Using the latter logic statement that characterises which events from our class of events are chiral, we construct the variables V_i in each set (S_c , S_{nc}) and ensure these will satisfy all the conditions mentioned in section 2.

Given our symmetry group G , and the parity operation denoted by \mathcal{P} , a non-chiral event e is one for which there exists a set of group elements $\{g_1, \dots, g_N\}$ such that $\mathcal{P}e = \hat{g}e$, where \hat{g} is the

composition of the N group elements in $\{g_1, \dots, g_N\}$. By going through all the possibilities for \hat{g} , we derive the logic statement that is true only for non-chiral events. We can then easily obtain the logic statement for chiral events by negation.

We start by deriving the logic statement which is true only for non-chiral collision events. Let the four-momenta of the incoming particles be denoted by p and q , while for the outgoing particles denoted by a, b, c, d . We can represent an event by e and a collision event by \hat{e} which follows from Lemma 2.27 in [3] as

$$e = \left[\begin{array}{cc|cccc} m_p & m_q & m_a & m_b & m_c & m_d \\ \mathbf{p} & \mathbf{q} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array} \right] \quad \hat{e} = \left[\begin{array}{cc|cccc} m_p & m_q & m_a & m_b & m_c & m_d \\ \mathbf{0} & \mathbf{0} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ p & -p & a_z & b_z & c_z & d_z \end{array} \right]$$

where we have aligned \vec{p} and \vec{q} with the positive and negative z -axis in the rest frame of $(p + q)$ ¹. Here we note $m_i \geq 0$, $\mathbf{j} \in \mathbb{C}$ and $j_z \in \mathbb{R}$ for $j = a, b, c, d$. Given any $g \in G$ has an inverse, we can say

$$g \cdot \hat{e} = \left(\begin{array}{c} \pi \\ \curvearrowright \\ yz \end{array} \right) \cdot \mathcal{P} \cdot \hat{e} \Rightarrow \hat{e} \text{ is non-chiral} \quad (1)$$

Noting that

$$\left(\begin{array}{c} \pi \\ \curvearrowright \\ yz \end{array} \right) \cdot \mathcal{P} \cdot \hat{e} = \left[\begin{array}{cc|cccc} m_p & m_q & m_a & m_b & m_c & m_d \\ \mathbf{0} & \mathbf{0} & -\mathbf{a}^* & -\mathbf{b}^* & -\mathbf{c}^* & -\mathbf{d}^* \\ p & -p & a_z & b_z & c_z & d_z \end{array} \right]$$

we see that we can restrict the part of g that comes from the Lorentz group to only rotations by θ about the z -axis, since \vec{p} and \vec{q} are already fixed to their original state and must remain the same. Hence we can rewrite equation (1) as

$$\left(\begin{array}{c} \theta \\ \curvearrowright \\ xy \end{array} \right) \cdot h \cdot \hat{e} = \left(\begin{array}{c} \pi \\ \curvearrowright \\ yz \end{array} \right) \cdot \mathcal{P} \cdot \hat{e} \Rightarrow \hat{e} \text{ is non-chiral} \quad (2)$$

for any $h \in H = S_2 \times S_4$, giving a total of 48 cases for collision events.

4 Results

4.1 Collision events

We explicitly go through the first case and then state the final results for the rest of the cases. The logic statement that is true for non-chiral collision events is given in section 4.1.1. The explicit calculation for case 2 which is more involved is given in appendix D.

¹This means $\vec{p} + \vec{q} = 0$.

Case 1

In this first case $h = 1_{S_2} \cdot 1_{S_4}$ which represents the identity in both permutation groups. Following equation (2) we have

$$\left(\begin{array}{c} \pi \\ \curvearrowright \\ yz \end{array} \right) \cdot \mathcal{P} \cdot \hat{e} = \left[\begin{array}{cc|cccc} m_p & m_q & | & m_a & m_b & m_c & m_d \\ \mathbf{0} & \mathbf{0} & | & |\mathbf{a}|e^{-i\alpha+i\pi+2i\pi n_a} & |\mathbf{b}|e^{-i\beta+i\pi+2i\pi n_b} & |\mathbf{c}|e^{-i\gamma+i\pi+2i\pi n_c} & |\mathbf{d}|e^{-i\delta+i\pi+2i\pi n_d} \\ p & -p & | & a_z & b_z & c_z & d_z \end{array} \right]$$

$$\left(\begin{array}{c} \theta \\ \curvearrowright \\ xy \end{array} \right) \cdot 1_{S_2} \cdot 1_{S_4} \cdot \hat{e} = \left[\begin{array}{cc|cccc} m_p & m_q & | & m_a & m_b & m_c & m_d \\ \mathbf{0} & \mathbf{0} & | & |\mathbf{a}|e^{i(\theta+\alpha)} & |\mathbf{b}|e^{i(\theta+\beta)} & |\mathbf{c}|e^{i(\theta+\gamma)} & |\mathbf{d}|e^{i(\theta+\delta)} \\ p & -p & | & a_z & b_z & c_z & d_z \end{array} \right]$$

Now we seek a logic statement that is true when the above 2 expressions are equal. Comparing each slot of our representation we get²

$$\begin{aligned} & \left[(|\mathbf{a}| = 0) \vee (-\alpha + \pi + 2\pi n_a = \theta + \alpha) \right] \wedge \left[(|\mathbf{b}| = 0) \vee (-\beta + \pi + 2\pi n_b = \theta + \beta) \right] \wedge \\ & \left[(|\mathbf{c}| = 0) \vee (-\gamma + \pi + 2\pi n_c = \theta + \gamma) \right] \wedge \left[(|\mathbf{d}| = 0) \vee (-\delta + \pi + 2\pi n_d = \theta + \delta) \right] \\ \Rightarrow & \left[(|\mathbf{a}| = 0) \vee \left(\alpha = \frac{1}{2}(\pi - \theta) + n_a\pi \right) \right] \wedge \left[(|\mathbf{b}| = 0) \vee \left(\beta = \frac{1}{2}(\pi - \theta) + n_b\pi \right) \right] \wedge \\ & \left[(|\mathbf{c}| = 0) \vee \left(\gamma = \frac{1}{2}(\pi - \theta) + n_c\pi \right) \right] \wedge \left[(|\mathbf{d}| = 0) \vee \left(\delta = \frac{1}{2}(\pi - \theta) + n_d\pi \right) \right] \end{aligned}$$

Given all final state momenta have the same transverse plane angle modulo π , a general event following the above is one for which the four 3-momenta \vec{a} , \vec{b} , \vec{c} , \vec{d} live on the same plane containing the beam axis and hence must satisfy

$$(\vec{i} \times \vec{j}) \cdot \vec{k} = 0$$

for $i, j, k = \{a, b, c, d, p\}$, $i \neq j \neq k$. This can be translated to a Lorentz invariant form using Lemma 2.35 and A.2.2 of [3]

$$\epsilon_{abpq} = \epsilon_{acpq} = \epsilon_{adpq} = \epsilon_{bdpq} = \epsilon_{bcpq} = \epsilon_{cdpq} = 0 \quad (3)$$

where a, b, c, d, p are 4-momenta and $\epsilon_{abpq} = \epsilon_{\mu\nu\sigma\rho} a^\mu b^\nu p^\sigma q^\rho$. This result generalises to N outgoing particles by writing

$$\epsilon_{ijpq} = 0$$

for $i, j =$ all possible 2-pairs from $\{\text{outgoing 4-momenta}\}$, giving $\binom{N}{2}$ terms.

²The \vee symbol means ‘or’ and the \wedge symbol means ‘and’ in mathematical logic.

Case 2

The explicit calculation for this case can be found in appendix D. It serves as an example of how the calculations for the results presented throughout this paper have been produced. For the meaning of the notation used in the expression below and the rest of the paper see appendix C.

$$h = 1_{S_2} \cdot (ab)$$

$$\begin{aligned}
& (a^2 = b^2) \wedge (G \binom{a-b, p+q}{p-q, p+q} = 0) \wedge (G \binom{a-b, p+q}{a+b, p+q} = 0) \\
& \wedge \left[(\Delta_3(a, p, q) = 0) \wedge (\Delta_3(b, p, q) = 0) \wedge (\Delta_3(c, p, q) \neq 0) \wedge (\Delta_3(d, p, q) \neq 0) \right] \\
& \vee \left[(\Delta_3(a, p, q) \neq 0) \wedge (\Delta_3(b, p, q) \neq 0) \wedge (\Delta_3(c, p, q) \neq 0) \wedge (\Delta_3(d, p, q) = 0) \right. \\
& \wedge (G \binom{a-b, p+q}{c, p+q} = 0) \left. \vee \left[(\Delta_3(a, p, q) \neq 0) \wedge (\Delta_3(b, p, q) \neq 0) \wedge (\Delta_3(d, p, q) \neq 0) \right. \right. \\
& \wedge (\Delta_3(c, p, q) = 0) \wedge (G \binom{a-b, p+q}{d, p+q} = 0) \left. \left. \vee \left[(\Delta_3(a, p, q) \neq 0) \wedge (\Delta_3(b, p, q) \neq 0) \wedge \right. \right. \right. \\
& \left. \left. \left. (\Delta_3(d, p, q) \neq 0) \wedge (\Delta_3(c, p, q) \neq 0) \wedge (G \binom{a-b, p+q}{c-d, p+q} = 0) \right] \right] \right] \tag{4}
\end{aligned}$$

Cases 3-7

Cases 3-7 follow case 2 by symmetry with the element of S_4 in each one being (cd) , (bc) , (bd) , (ac) , (ad) . These are 2-cycles in S_4 with the rest of the elements in 1-cycles. Each additional final state particle will double the number of subcases in this case, as can be seen from appendix D, making the generalisation of equation (4) to N outgoing particles a difficult task. However, one can exploit the apparent pattern in equation (4) to guess the additions that need to be made in each bracket for more outgoing particles. Cases like this one, for which h is guaranteed to be found in S_N for N greater than 4, are likely to generalise to more outgoing particles simply by comparison and without the need for an explicit calculation.

Cases 8-15

Cases 8-15 which are the 3-cycles of S_4 combined with 1_{S_2} were found to be sub-cases of case 1 and were thus discarded. We discard sub-cases since they do not give new information on how a state can be non-chiral. Concretely, we know that if all final state momenta lie in a plane the event is non-chiral. If a new case instructs us that an event is non-chiral if all final state momenta are zero, then that is already included in the information of all momenta lying in a plane and we can thus discard the new case.

Case 16

$$h = 1_{S_2} \cdot (ab)(cd)$$

Remark: The group element in S_4 for this case is not found in S_3 , hence this case presents a new type of non-chirality with respect to the case of $2 \rightarrow 3$ collision events. This new information inhibits a straightforward generalisation of results for $2 \rightarrow N$ collision events, without the explicit calculation of such new type of cases. This remark applies to other cases presented below regardless of the type of events considered.

$$\begin{aligned}
& (a^2 = b^2) \wedge (c^2 = d^2) \wedge (G \binom{a-b, p+q}{p-q, p+q} = 0) \wedge (G \binom{a-b, p+q}{a+b, p+q} = 0) \\
& \wedge (G \binom{c-d, p+q}{p-q, p+q} = 0) \wedge (G \binom{c-d, p+q}{c+d, p+q} = 0) \\
& \wedge \left[(\Delta_3(a, p, q) = 0) \wedge (\Delta_3(b, p, q) = 0) \wedge (\Delta_3(c, p, q) \neq 0) \wedge (\Delta_3(d, p, q) \neq 0) \right] \\
& \vee \left[(\Delta_3(c, p, q) = 0) \wedge (\Delta_3(d, p, q) = 0) \wedge (\Delta_3(a, p, q) \neq 0) \wedge (\Delta_3(b, p, q) \neq 0) \right] \\
& \vee \left[(G \binom{a-b, p+q}{c+d, p+q} = 0) \vee (G \binom{c-d, p+q}{b, p+q} = 0) \vee (G \binom{a-b, p+q}{d, p+q} = 0) \right] \tag{5}
\end{aligned}$$

Cases 17,18

Cases 17 and 18 are the other two elements of S_4 which consist of two 2-cycles and hence follow case 16 by symmetry.

Case 19

$h = 1_{S_2} \cdot (dcba)$, where in our convention, this means $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$.

$$\begin{aligned}
& (a^2 = b^2 = c^2 = d^2) \wedge (G \binom{a-b, p+q}{a+b, p+q} = 0) \wedge (G \binom{b-c, p+q}{b+c, p+q} = 0) \\
& \wedge (G \binom{c-d, p+q}{c+d, p+q} = 0) \wedge (G \binom{a-b, p+q}{p-q, p+q} = 0) \wedge (G \binom{b-c, p+q}{p-q, p+q} = 0) \\
& \wedge (G \binom{c-d, p+q}{p-q, p+q} = 0) \wedge (a = c) \wedge (b = d) \tag{6}
\end{aligned}$$

Cases 20-24

Cases 20 to 24 follow case 19 by symmetry as they are the 4-cycles of S_4 . However case 21 is the same as case 20, case 23 is the same as case 19 and case 24 is the same as case 22. Hence cases 21, 23 and 24 were discarded. One can find the explicit h for each of these cases in code form given in appendix E. The same applies for subsequent references to enumerated cases.

Case 25

$$h = \begin{pmatrix} \theta \\ \curvearrowright \\ xy \end{pmatrix} \cdot \begin{pmatrix} \pi \\ \curvearrowright \\ yz \end{pmatrix} \cdot (pq) \cdot 1_{S_4}$$

$$\begin{aligned}
& (p^2 = q^2) \wedge (G \begin{pmatrix} a & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} b & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \\
& \wedge (G \begin{pmatrix} c & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} d & , p+q \\ p-q & , p+q \end{pmatrix} = 0)
\end{aligned} \tag{7}$$

The generalisation to N outgoing particles for this case is trivial. We simply augment the rest of the Gram determinants $G \begin{pmatrix} \cdot & , p+q \\ p-q & , p+q \end{pmatrix} = 0$ with a \wedge symbol.

Case 26

$$\begin{aligned}
h &= \left(\begin{smallmatrix} \theta \\ \curvearrowright \\ xy \end{smallmatrix} \right) \cdot \left(\begin{smallmatrix} \pi \\ \curvearrowright \\ yz \end{smallmatrix} \right) \cdot (pq) \cdot (ab) \\
& (p^2 = q^2) \wedge (a^2 = b^2) \wedge (G \begin{pmatrix} a+b & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \\
& \wedge (G \begin{pmatrix} a-b & , p+q \\ a+b & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} c & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} d & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \\
& \wedge [(\Delta_3(a, p, q) = 0) \vee (\Delta_3(a+b, p, q) = 0) \vee (\Delta_3(a-b, p, q) = 0)]
\end{aligned} \tag{8}$$

Cases 27-31

Cases 27-31 follow case 26 by symmetry. Their elements are of the form $\left(\begin{smallmatrix} \theta \\ \curvearrowright \\ xy \end{smallmatrix} \right) \cdot \left(\begin{smallmatrix} \pi \\ \curvearrowright \\ yz \end{smallmatrix} \right) \cdot (pq)$ and one of the elements of S_4 consisting of one 2-cycle and two 1-cycles. The generalisation to N outgoing particles follows by symmetry in the second line of equation (8) and gives $\binom{N}{2}$ cases.

Case 32

$$\begin{aligned}
h &= \left(\begin{smallmatrix} \theta \\ \curvearrowright \\ xy \end{smallmatrix} \right) \cdot \left(\begin{smallmatrix} \pi \\ \curvearrowright \\ yz \end{smallmatrix} \right) \cdot (pq) \cdot (dcb) \\
& (p^2 = q^2) \wedge (b^2 = c^2 = d^2) \wedge (G \begin{pmatrix} a & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} b & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \\
& \wedge (G \begin{pmatrix} c & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} d & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} b-c & , p+q \\ b+c & , p+q \end{pmatrix} = 0) \\
& \wedge (G \begin{pmatrix} c-d & , p+q \\ c+d & , p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} a & , p+q \\ p-q & , p+q \end{pmatrix} = 0) \\
& \wedge (\epsilon_{(b+d)cpq} = 0) \wedge (\epsilon_{(b+c)dpq} = 0) \wedge (\epsilon_{(c+d)bpq} = 0)
\end{aligned} \tag{9}$$

Cases 33-39

Cases 33-39 are the 3-cycles of S_4 combined with the non-trivial element of S_2 and follow case 32 by symmetry. However, cases 33, 36, 38 and 39 are included in previously considered cases and were thus discarded.

Case 40

$$h = \left(\underset{\curvearrowright}{\theta} \right)_{xy} \cdot \left(\underset{\curvearrowright}{\pi} \right)_{yz} \cdot (pq) \cdot (ab)(cd)$$

$$\begin{aligned} & (p^2 = q^2) \wedge (a^2 = b^2) \wedge (c^2 = d^2) (G \binom{a+b, p+q}{p-q, p+q} = 0) \\ & \wedge (G \binom{c+d, p+q}{p-q, p+q} = 0) \wedge [(\Delta_3(a+b, p, q) = 0) \vee (\Delta_3(a-b, p, q) = 0)] \\ & \vee [(\Delta_3(c+d, p, q) = 0) \vee (\Delta_3(c-d, p, q) = 0)] \end{aligned} \quad (10)$$

Cases 41,42

Cases 41 and 42 follow case 40 by symmetry with their elements being of the form $h = \left(\underset{\curvearrowright}{\theta} \right)_{xy} \cdot \left(\underset{\curvearrowright}{\pi} \right)_{yz} \cdot (pq)$ and an element of S_4 consisting of two 2-cycles.

Case 43

$$h = \left(\underset{\curvearrowright}{\theta} \right)_{xy} \cdot \left(\underset{\curvearrowright}{\pi} \right)_{yz} \cdot (pq) \cdot (dcba)$$

$$\begin{aligned} & (p^2 = q^2) \wedge (a^2 = b^2 = c^2 = d^2) \wedge (G \binom{a-b, p+q}{a+b, p+q} = 0) \wedge (G \binom{b-c, p+q}{b+c, p+q} = 0) \\ & \wedge (G \binom{c-d, p+q}{c+d, p+q} = 0) \wedge (G \binom{a+b, p+q}{p-q, p+q} = 0) \wedge (G \binom{b+c, p+q}{p-q, p+q} = 0) \\ & \wedge (G \binom{c+d, p+q}{p-q, p+q} = 0) \wedge \left[(\Delta_3(a, p, q) = 0) \vee \left[(\epsilon_{(p+q)(a-c)pb} = 0) \wedge (\epsilon_{(p+q)(b-d)ap} = 0) \right] \right] \\ & \vee \left[(\Delta_3(a-c, p, q) = 0) \wedge (\Delta_3(b-d, p, q) = 0) \wedge \left[(\Delta_3(a+b, p, q) = 0) \right] \right] \\ & \vee \left[\left[(G \binom{a+b, p+q}{p-q, p+q} = 0) \wedge (\Delta_3(a-b, p, q) = 0) \right] \right] \end{aligned} \quad (11)$$

Cases 44-48

Cases 44 to 48 follow case 43 by symmetry and their elements are of the form $h = \left(\underset{\curvearrowright}{\theta} \right)_{xy} \cdot \left(\underset{\curvearrowright}{\pi} \right)_{yz} \cdot (pq)$ and one 4-cycle element of S_4 .

4.1.1 Condition for a collision event to be non-chiral

A collision event is non-chiral iff the logic statement - formed by combining all the logic statements in equations (3)-(11), as well as those that follow by symmetries, with \vee - is true.

4.2 Non-collision events with at least one of p and q being massive

Using corollary 3.3 of [3], a non-collision event with at least one of p and q being massive can be represented by

$$e = \left\{ \begin{array}{cc|cccc} p & q & a & b & c & d \\ m_p & m_q & m_a & m_b & m_c & m_d \\ \vec{0} & \vec{0} & \vec{a} & \vec{b} & \vec{c} & \vec{d} \end{array} \right\}. \quad (12)$$

The action of parity on this representation gives

$$\mathcal{P} \cdot e = \left\{ \begin{array}{cc|cccc} p & q & a & b & c & d \\ m_p & m_q & m_a & m_b & m_c & m_d \\ \vec{0} & \vec{0} & -\vec{a} & -\vec{b} & -\vec{c} & -\vec{d} \end{array} \right\}, \quad (13)$$

from which we can see that our symmetry group can exclude boosts and permutations of p with q . Hence for non-collision events we only consider global general rotations R combined with the S_4 permutation group on the final state momenta. We will explicitly go through case 1 with $h = 1_{S_4} \cdot R$ and state the result of all others.

Case 1

$$h = 1_{S_4} \cdot R$$

$$1_{S_4} \cdot R \cdot \mathcal{P} \cdot e = \left\{ \begin{array}{cc|cccc} p & q & a & b & c & d \\ m_p & m_q & m_a & m_b & m_c & m_d \\ \vec{0} & \vec{0} & -R\vec{a} & -R\vec{b} & -R\vec{c} & -R\vec{d} \end{array} \right\}, \quad (14)$$

For non-chiral events we require equation (14) to be equal to equation (12). This forces the condition $(\vec{a} = -R\vec{a}) \wedge (\vec{b} = -R\vec{b}) \wedge (\vec{c} = -R\vec{c}) \wedge (\vec{d} = -R\vec{d})$. A general event following the latter condition has all four final state momenta lying in a common plane which is perpendicular to the rotation axis of R , where R must represent a π rotation, and any one of them can be the null vector. This can be expressed in a Lorentz invariant way as follows

$$\epsilon_{abc(p+q)} = \epsilon_{abd(p+q)} = \epsilon_{bcd(p+q)} = \epsilon_{acd(p+q)} = 0 \quad (15)$$

The generalisation to N final state particles is trivial and gives $\binom{N}{3}$ ϵ terms. To see equation (15) consider the following. If $\vec{a}, \vec{b}, \vec{c}$ are to be in the same plane we require $(\vec{a} \times \vec{b}) \cdot \vec{c} = 0$. Note that $(p+q) = [m_p + m_q, 0, 0, 0]$ and $(\vec{a} \times \vec{b}) \cdot \vec{c} = 0 \iff \epsilon_{ijk} a_i b_j c_k = 0$, where $i, j, k = \{1, 2, 3\}$. Since $(p+q)$ is only non-zero for the index 0, we can augment it to $\epsilon_{ijk} a_i b_j c_k = 0$ and switch to Greek indices that run from 0 to 3 as $\epsilon_{abc(p+q)} = \epsilon_{\mu\nu\sigma\rho} a^\mu b^\nu c^\sigma (p+q)^\rho = 0 \iff \epsilon_{ijk} a_i b_j c_k = 0$. In this last step we have used the property of the ϵ tensor that is non-zero only when all the indices are different.

Case 2

$$h = (ab)R$$

$$\begin{aligned} & (a^2 = b^2) \wedge a \neq b \wedge (G \binom{a, p+q}{a, p+q} = G \binom{b, p+q}{b, p+q}) \\ & \wedge (G \binom{a, p+q}{c, p+q} = G \binom{b, p+q}{c, p+q}) \wedge (G \binom{a, p+q}{d, p+q} = G \binom{b, p+q}{d, p+q}) \end{aligned} \quad (16)$$

To generalise this expression to N final state particles, we just need to extend the second line in an obvious pattern so that it consists of $N - 2$ terms.

Cases 3-7

Cases 3-7 have an $h = gR$ with g being an element of S_4 that permutes two final state objects and leaves the other two unaffected. These follow equation (16) by symmetry.

Case 8

$$h = (ab)(cd)R$$

$$\begin{aligned} & (a^2 = b^2) \wedge (c^2 = d^2) \wedge (G \binom{a, p+q}{a, p+q} = G \binom{b, p+q}{b, p+q}) \wedge (G \binom{c, p+q}{c, p+q} = G \binom{d, p+q}{d, p+q}) \\ & \wedge [((a+b)^2 = 4a^2 = 4b^2) \wedge ((c+d)^2 = 4c^2 = 4d^2)] \vee [(G \binom{a+b, p+q}{c-d, p+q} = 0) \wedge (c \neq d)] \end{aligned} \quad (17)$$

Cases 9-10

These cases have an $h = gR$ where g is an element of S_4 that consists of two 2-cycles and their statement follows equation (17) by symmetry.

Cases 11-18

These cases have $h = gR$ with $g \in S_4$ being a 3-cycle. They are all subsets of case 1 and were thus discarded.

Case 19

$$h = (abcd)R$$

$$\begin{aligned} & (a^2 = b^2 = c^2 = d^2) \wedge (G \binom{a, p+q}{a, p+q} = G \binom{b, p+q}{b, p+q} = G \binom{c, p+q}{c, p+q} = G \binom{d, p+q}{d, p+q}) \\ & \wedge (G \binom{a-c, p+q}{b-d, p+q} = 0) \end{aligned} \quad (18)$$

Cases 20-24

Cases 20-24 have an $h = gR$ with g being a 4-cycle of S_4 and their statement follows equation (18) by symmetry.

4.2.1 Condition for a non-collision event - with at least one of p and q being massive - to be non-chiral

A non-collision event with at least one of p and q being massive is non-chiral iff the logic statement - formed by combining all the logic statements in equations (15)-(18), as well as those that follow by symmetries, with \vee - is true.

4.3 Non-collision events with both p and q being massless

Following section 3.3 of [3], a non-collision event with both incoming objects being massless can be represented in the $(a+b+c+d)$ rest frame with p and q aligned with the z-axis as

$$\hat{e} = \left[\begin{array}{cc|cccc} 0 & 0 & m_a & m_b & m_c & m_d \\ 0 & 0 & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array} \right] \quad \text{with} \quad \left(\begin{array}{l} p_z, q_z, m_a, m_b, m_c, m_d \geq 0, \\ p_z + q_z > 0, \\ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{C}, \\ \mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d} = 0, \\ a_z, b_z, c_z, d_z \in \mathbb{R} \text{ and } a_z + b_z + c_z + d_z = 0 \end{array} \right) \quad (19)$$

We would like to act with a symmetry group on $R_y(\pi) \cdot \mathcal{P} \cdot e$ and check when it is equal to e . Note that $R_y(\pi)$ reduces the amount of minus signs we have to work with and its inverse always exists, in order to absorb it into the group element that would take the parity inverted e to e . Given the form of $R_y(\pi) \cdot \mathcal{P} \cdot e$ is

$$R_y(\pi) \cdot \mathcal{P} \cdot e = \left[\begin{array}{cc|cccc} 0 & 0 & m_a & m_b & m_c & m_d \\ 0 & 0 & \mathbf{a}^* & \mathbf{b}^* & \mathbf{c}^* & \mathbf{d}^* \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array} \right]. \quad (20)$$

we can see that our symmetry group can exclude boosts, (pq) swaps and any rotation other than about the z-axis, otherwise p and q which are already matched to e will lose this matching. Hence our symmetry group is $R_z(\theta) \cdot g$ where g is an element of S_4 that permutes the final state momenta a,b,c and d. We will explicitly go through case 1 and then state the result for the remaining cases.

We note that an event with $(a + b + c + d)^2 = 0$ has all vectors being massless and pointing in the same spatial direction. Hence there are only two independent 4-momenta in the event which cannot form a Lorentz invariant pseudoscalar (at least four are needed). This means an event with

$$(a + b + c + d)^2 = 0 \quad (21)$$

is always non-chiral.

Case 1

In the first case we have $h = R_z(\theta) \cdot 1_{S_4}$. The explicit form of $e = h \cdot R_y(\pi) \cdot \mathcal{P} \cdot e$ can be written as³

$$\left[\begin{array}{cc|cccc} 0 & 0 & m_a & m_b & m_c & m_d \\ 0 & 0 & |\mathbf{a}|e^{i(\alpha-2\pi n_a)} & |\mathbf{b}|e^{i(\beta-2\pi n_b)} & |\mathbf{c}|e^{i(\gamma-2\pi n_c)} & |\mathbf{d}|e^{i(\delta-2\pi n_d)} \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array} \right] \stackrel{?}{=} \left[\begin{array}{cc|cccc} 0 & 0 & m_a & m_b & m_c & m_d \\ 0 & 0 & |\mathbf{a}|e^{i(\theta-\alpha)} & |\mathbf{b}|e^{i(\theta-\beta)} & |\mathbf{c}|e^{i(\theta-\gamma)} & |\mathbf{d}|e^{i(\theta-\delta)} \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array} \right] \quad (22)$$

$$\Rightarrow ((\mathbf{a} = 0) \vee (\alpha = \theta - \alpha + 2\pi n_a)) \wedge ((\mathbf{b} = 0) \vee (\beta = \theta - \beta + 2\pi n_b)) \wedge ((\mathbf{c} = 0) \vee (\gamma = \theta - \gamma + 2\pi n_c)) \wedge ((\mathbf{d} = 0) \vee (\delta = \theta - \delta + 2\pi n_d))$$

$$\Rightarrow ((\mathbf{a} = 0) \vee (\alpha = \frac{\theta}{2} + \pi n_a)) \wedge ((\mathbf{b} = 0) \vee (\beta = \frac{\theta}{2} + \pi n_b)) \wedge ((\mathbf{c} = 0) \vee (\gamma = \frac{\theta}{2} + \pi n_c)) \wedge ((\mathbf{d} = 0) \vee (\delta = \frac{\theta}{2} + \pi n_d))$$

This is telling us that $\vec{a}, \vec{b}, \vec{c}, \vec{d}, \vec{p}, \vec{q}$ all lie in the same plane which includes the z-axis and has an angle $\frac{\theta}{2}$ to the x-axis. Any of the vectors can be the null vector, but not \vec{p} and \vec{q} simultaneously. We thus want relationships of the form $(\vec{a} \times \vec{b}) \cdot (\vec{p} + \vec{q}) = 0$ in the $(a+b+c+d)$ rest frame. If we let $\Sigma = (a + b + c + d)$, this can be written as follows

$$\epsilon_{ab(p+q)\Sigma} = \epsilon_{ac(p+q)\Sigma} = \epsilon_{ad(p+q)\Sigma} = \epsilon_{bc(p+q)\Sigma} = \epsilon_{bd(p+q)\Sigma} = \epsilon_{cd(p+q)\Sigma} = 0 \quad (23)$$

This can be generalised to N outgoing objects by extending the above expression to all pairs in the first 2 slots of ϵ .

Case 2

$$h = R_z(\theta) \cdot (ab)$$

$$\begin{aligned} (a^2 = b^2) \wedge (G \begin{pmatrix} a-b, \Sigma \\ p+q, \Sigma \end{pmatrix} = 0 \wedge (\Delta_2(a, \Sigma) = \Delta_2(b, \Sigma))) \\ \wedge (G \begin{pmatrix} a, \Sigma \\ c, \Sigma \end{pmatrix} = G \begin{pmatrix} b, \Sigma \\ c, \Sigma \end{pmatrix}) \wedge (G \begin{pmatrix} a, \Sigma \\ d, \Sigma \end{pmatrix} = G \begin{pmatrix} b, \Sigma \\ d, \Sigma \end{pmatrix}) \end{aligned} \quad (24)$$

Cases 3-7

These have $h = R_z(\theta) \cdot g$ with g being a 2-cycle of S_4 with the other two elements left unchanged. They follow equation (24) by symmetry. Extending these statements to N outgoing objects requires the extension of the second line of equation (24) in a straight-forward way giving $N-2$ terms rather than 2.

³The question mark on the equality sign is asking: 'What condition must be met for this equality to hold?'

Case 8

$$h = R_z(\theta) \cdot (ab)(cd)$$

$$\begin{aligned} (a^2 = b^2) \wedge \left(G \begin{pmatrix} a-b, \Sigma \\ p+q, \Sigma \end{pmatrix} = 0 \wedge (\Delta_2(a, \Sigma) = \Delta_2(b, \Sigma)) \right) \\ \wedge (c^2 = d^2) \wedge \left(G \begin{pmatrix} c-d, \Sigma \\ p+q, \Sigma \end{pmatrix} = 0 \wedge (\Delta_2(c, \Sigma) = \Delta_2(d, \Sigma)) \right) \\ \wedge G \begin{pmatrix} a-b, \Sigma \\ c+d, \Sigma \end{pmatrix} = 0 \end{aligned} \quad (25)$$

Cases 9-10

These cases have $h = R_z(\theta) \cdot g$ with g an element of S_4 consisting of two 2-cycles. They follow equation (25) by symmetry.

Cases 11-24

These cases have $h = R_z(\theta) \cdot g$ with $g \in S_4$ a 3-cycle for cases 11-18 and a 4-cycle for cases 19-24. They were found to be sub-cases of previously included cases and were thus discarded.

4.3.1 Condition for a non-collision event with both p and q being massless to be non-chiral

A non-collision event with both p and q being massless is non-chiral iff the logic statement - formed by combining the logic statements in equations (21)-(25) as well as those that follow by symmetries, with \vee - is true.

4.4 Non-collision events with $\mathbf{p} = \mathbf{q} = \mathbf{0}$

We can no longer use p and q and we thus orient our coordinate system with the z-axis aligned with the momentum of particle a. We also work in the (a+b+c+d) rest frame as in the previous section and the results follow directly from section 3.3.1 onwards in [3]. Note that when $a_z = 0$, which implies $\vec{a} = 0$ in our choice of coordinate system, we cannot form a pseudoscalar and those cases are non-chiral leaving the z-axis well defined for the rest of the cases considered.

Our setup can be represented by

$$e = \left[\begin{array}{c|ccc} m_a & m_b & m_c & m_d \\ 0 & |\mathbf{b}|e^{i(\beta-2\pi n_b)} & |\mathbf{c}|e^{i(\gamma-2\pi n_c)} & |\mathbf{d}|e^{i(\delta-2\pi n_d)} \\ a_z & b_z & c_z & d_z \end{array} \right] \quad (26)$$

and so we can quote the result of [3] in corollary 4.6, with the mapping $p+q \rightarrow a$, $\Sigma \rightarrow (a+b+c+d)$, $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow d$. A non-collision event with $\mathbf{p} = \mathbf{q} = \mathbf{0}$ is non-chiral iff

$$\begin{array}{c}
G\left(\begin{array}{c} a, \Sigma \\ a, \Sigma \end{array}\right) = 0 \\
\hline
(a + b + c + d)^2 = 0 \\
\hline
[b, c, d, a] = 0 \\
\hline
\left. \begin{array}{l}
(b^2 = c^2) \wedge \left(G\left(\begin{array}{c} b - c, \Sigma \\ a, \Sigma \end{array}\right) = 0 \right) \wedge (\Delta_2(b, \Sigma) = \Delta_2(c, \Sigma)) \\
\hline
(c^2 = d^2) \wedge \left(G\left(\begin{array}{c} c - d, \Sigma \\ a, \Sigma \end{array}\right) = 0 \right) \wedge (\Delta_2(c, \Sigma) = \Delta_2(d, \Sigma)) \\
\hline
(d^2 = b^2) \wedge \left(G\left(\begin{array}{c} d - b, \Sigma \\ a, \Sigma \end{array}\right) = 0 \right) \wedge (\Delta_2(d, \Sigma) = \Delta_2(b, \Sigma))
\end{array} \right\} \quad (27)
\end{array}$$

where the first line comes from the case when $a_z = 0$.

5 Computational test of results using Geometric Algebra

In this section we describe the method used to test the above results. We randomly generate $2 \rightarrow 4$ events and use geometric algebra to label whether the event is chiral or non-chiral. We then evaluate the logic statements derived above - which are true for non-chiral states - on these events and check that the output is true for non-chiral events and false for chiral events.

The motivation to use geometric algebra (GA) rather than the standard linear algebra is that with GA the geometry is manifest and manipulation of mathematical objects becomes almost trivial, while also boosting the efficiency of computations.

Since from our choice of setup all types of events exclude boosts from their symmetry group, in this section we are working with 3-vectors and do not involve any boost transformations.

5.1 Basics of geometric algebra

There are numerous sources for an introduction to geometric algebra [4],[5], but here we sketch the very basics for 3D Euclidean geometry.

Define the geometric product of two vectors using the usual dot product and outer product as

$$uv = u \cdot v + u \wedge v \quad (28)$$

The outer product forms a bivector and has the following defining properties

$$u \wedge v = -v \wedge u \quad (29)$$

$$u \wedge (v + w) = u \wedge v + u \wedge w \quad (30)$$

The bivector $u \wedge v$ can be visualized as the directed plane formed by the vectors u and v - see Figure 1. Let our vector space be spanned by e_1, e_2, e_3 . We note the following important results

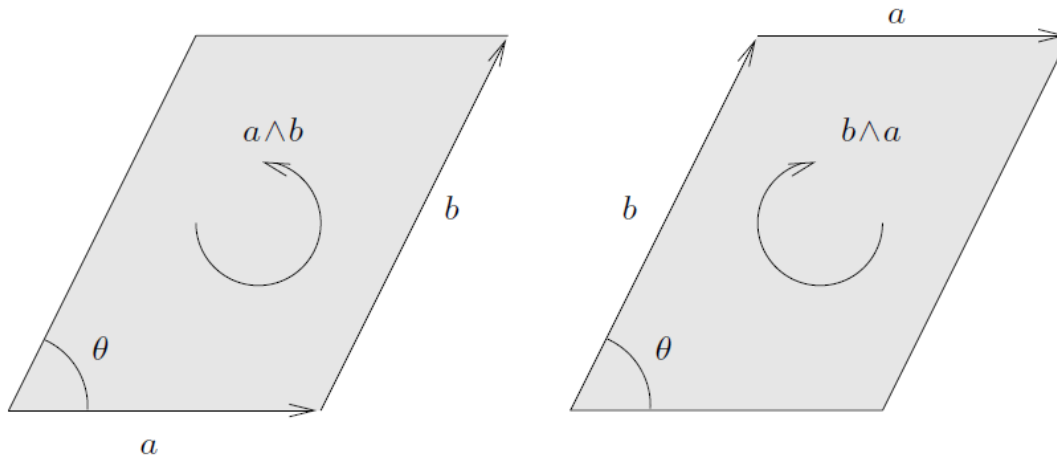


Figure 1: The outer product. The outer or wedge product of a and b returns a directed area element of area $|a||b|\sin(\theta)$. The orientation of the parallelogram is defined by whether the circuit a, b, a, b is right-handed (anticlockwise) or left-handed (clockwise). Interchanging the order of the vectors reverses the orientation and introduces a minus sign in the product. The figure is taken from [4].

$$e_i e_j = e_i \wedge e_j \quad (31)$$

$$e_i e_j = -e_j e_i \quad (32)$$

$$(e_i e_j)^2 = -1 \quad (33)$$

Equation (33) is a profound property of bivectors that behave as imaginary numbers and can be used to generalise quaternions.

A rotor is an object characterised by an angle θ and a bivector B which are enough information to fully parameterise a rotation⁴. We define a rotor as

$$R = e^{-\frac{\theta}{2}B} = \cos\left(\frac{\theta}{2}\right) - B\sin\left(\frac{\theta}{2}\right) \quad (34)$$

⁴Boosts - not used in this paper - can be represented using the spacetime algebra (STA) with basis vectors $\gamma_\mu \cdot \gamma_\nu = \eta_{\mu\nu}$ as $R = e^{\alpha\gamma_i\gamma_0}$, for a boost in the γ_i direction and rapidity α . The transformation with R is the same as for rotations, see [4].

which through the operation RvR^\dagger takes the vector v and rotates it by an angle θ in the directed plane described by B . We define the dagger operator on a bivector to result in the reverse bivector. For example $(e_1e_2)^\dagger = (e_2e_1)$. If we have 2 vectors a and b and we would like to rotate a to b then we construct the appropriate rotor R as described⁵ in Figure 2, resulting in

$$\begin{aligned} n &= \frac{a+b}{\|a+b\|} \\ R &= bn \end{aligned} \tag{35}$$

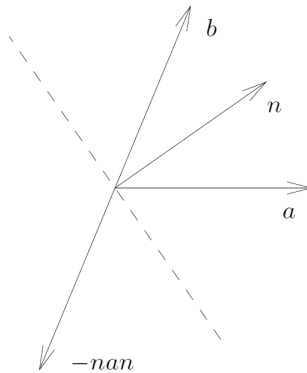


Figure 2: A rotation from a to b . The vector a is rotated onto b by first reflecting in the plane perpendicular to n , and then in the plane perpendicular to b . The vectors a , b and n all have unit length. Figure taken from [4].

5.2 Algorithm for collision events

The purpose of the algorithm is to take in a randomly generated event and label it either chiral or non-chiral, according to the definitions given in section 2.

For a collision event we have p aligned with the z -axis and $q = -p$, as we are working in the $(p+q)$ rest frame. Let an event be described by $S = [p, q, a, b, c, d]$ and define $RSR^\dagger = [RpR^\dagger, RqR^\dagger, \dots, RdR^\dagger]$, where all vectors are to be understood as 3-vectors. We can write the algorithm, that does not incorporate permutations, algebraically as follows

Steps⁶:

⁵In geometric algebra the reflection of a into the plane perpendicular to n is done simply by $a \rightarrow -nan$.

⁶Intuition: p, q, a_{12} define a plane in which we rotate by π using $R = a_{12}e_3$. Note that it does not matter here if we use a_{12} from S or S_1 since we are doing a π rotation.

- 1) Parity inversion: $S \rightarrow S_1 = -S$.
- 2) Project a into the 1-2 plane and normalise: $a_{12} = \frac{a-a_3}{|a-a_3|}$.
- 3) Map p, q, a back to their original state with $R = a_{12}e_3$: $S_1 \rightarrow S_2 = RS_1R^\dagger$.
- 4) If $S_2 = S$ output ‘non-chiral’, otherwise output ‘chiral’.

If a_{12} is zero, repeat the algorithm with b . If b_{12} is zero, then repeat the algorithm with c , and if c_{12} is zero, output ‘non-chiral’ and terminate. The idea is that after we map p, q and a we have no subspace in which to perform any other rotations and so we reach the step where we should check if the rest of the particles mapped back to their original state. With permutations the algorithm becomes more complex but the idea of subspaces remains and can be found in appendix E in code form, where we use the Clifford library [6]. The proof that the general algorithm covers all possible inputs is given in appendix B.1.

5.3 Results for collision events

Now that the algorithm has labeled collision events chiral or non-chiral, we can evaluate the logic statement in section 4.1.1 on these states and expect an output of true when we use the non-chiral states and false when we use the chiral ones. As with the rest of the cases discussed below, we generate 1000 states of each type and check the appropriate logic statement. The results are presented in figure 3.

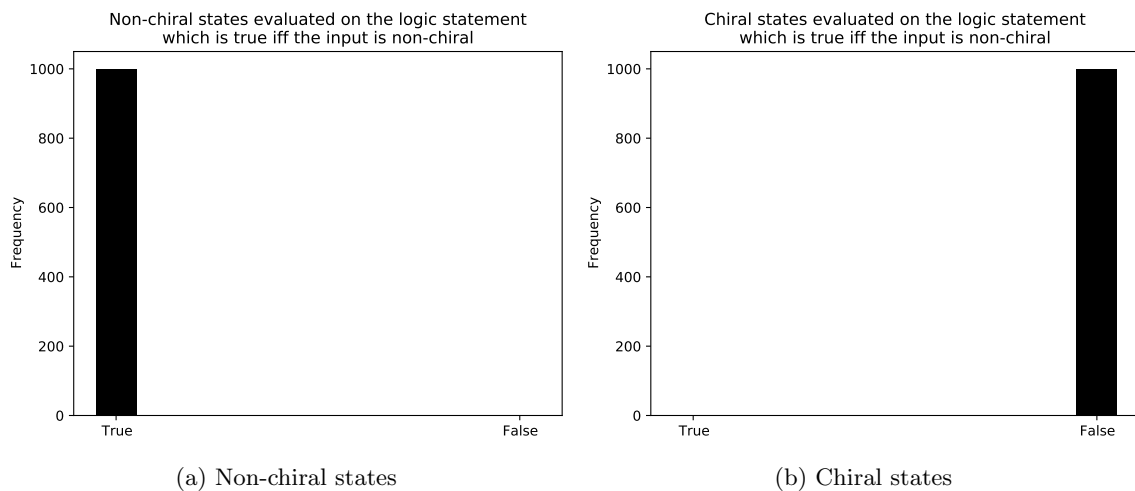


Figure 3: (a) Non-chiral and (b) chiral states evaluated on the logic statement that is true when a collision event is non-chiral. This logic statement can be found in section 4.1.1.

5.4 Algorithm for non-collision events with at least one of p and q being massive

For a non-collision event with at least one of p and q being massive, we have p, q both stationary. Under parity p and q are invariant so we can exclude boosts and only work with general rotations. Below we describe the algorithm for the case where no permutations are available. The algorithm that includes permutations can be found in appendix E in code form, where again we use the Clifford library [6]. The proof for the covering of the general algorithm can be found in appendix B.2.

Steps⁷:

- 1) Parity inversion: $S \rightarrow S_1 = -S$.
- 2) Map a and b back to their original state with $R = \frac{a \wedge b}{|a \wedge b|}$: $S_1 \rightarrow S_2 = RS_1R^\dagger$.
- 3) If $S_2 = S$ output ‘non-chiral’, otherwise output ‘chiral’.

If a, b are collinear, repeat the algorithm with a, c . If the latter are collinear output ‘non-chiral’ and terminate, since 3 out of 4 final state particles are collinear.

5.5 Results for non-collision events with at least one of p and q being massive

We test the logic statement given in section 4.2.1 with 1000 randomly generated chiral events and 1000 randomly generated non-chiral events that are labelled by the generalisation of the algorithm described above. The results are the same as shown in figure 3.

5.6 Non-collision events with both p and q being massless

For non-collision events with both p and q being massless we test the logic statement given in section 4.3.1. We again present the algorithm for the case where no permutations are available, while the full algorithm is given in appendix E in code form. In section B.3 we give the proof that the generalisation of the algorithm below covers all possible inputs.

Steps⁸:

⁷Intuition: For 2 final state particles, parity keeps them in the same original plane, hence a π rotation in that plane brings them back to their original state. A π rotation is achieved with $R = e^{\frac{\pi}{2}B} = B$, where $B = \frac{a \wedge b}{|a \wedge b|}$. Step 4 says that if a, b, c are collinear we can always do a π rotation in the $a - d$ plane and map everything back, so those states are non-chiral.

⁸Intuition: Once we map p and q back to their original state with R_1 , we can only do rotations in the plane perpendicular to p and q . Note step 2 fixed the 3^{rd} component of a and $a_{rot,12}$ is extracted from S_2 .

- 1) Parity inversion: $S \rightarrow S_1 = -S$.
- 2) Map p and q back to themselves with $R_1 = e_1 e_3 : S_1 \rightarrow S_2 = R_1 S_1 R_1^\dagger$.
- 3) In the 1-2 plane, rotate a to its original state with $R_2 = a_{12} a_{rot,12} : S_2 \rightarrow S_3 = R_2 S_2 R_2^\dagger$.
- 4) If $S_3 = S$ output ‘non-chiral’, otherwise output ‘chiral’.

5.7 Results for non-collision events with both p and q being massless

We follow the same test procedure as in the previous sections. The results follow what is expected from the logic statement in section 4.3.1 and match what is shown in figure 3.

5.8 Non-collision events with $p = q = 0$

For non-collision events with both p and q being 0 we test the logic statement given in equation (27). The algorithm for the case where no permutations are available is presented below and the full algorithm is given in appendix E in code form. The proof that the general algorithm covers all possible inputs can be found in appendix B.4.

Steps⁹:

- 1) Parity inversion: $S \rightarrow S_1 = -S$.
- 2) Map a back to its original state with $R_1 = e_1 e_3 : S_1 \rightarrow S_2 = R_1 S_1 R_1^\dagger$.
- 3) In the 1-2 plane, rotate b to its original state with $R_2 = b_{12} b_{rot,12} : S_2 \rightarrow S_3 = R_2 S_2 R_2^\dagger$.
- 4) If $S_3 = S$ output ‘non-chiral’, otherwise output ‘chiral’.

5.9 Results for non-collision events with $p = q = 0$

The test procedure is as mentioned above and the results follow what is expected from the logic statement in equation (27). They follow figure 3.

5.10 Construction of non-chiral states and precision of algorithms

Non-chiral states are extremely rare to occur just by the random generation of events and hence to test the logic statements with non-chiral input we constructed them by hand. For example, in the case of collision events when all particle momenta lie in a given plane (that contains the z -axis) the state is manifestly non-chiral. From the latter we can rotate the whole state with the rotor $e_1 e_2$ and still have a non-chiral state. Since the real number 0 can be computationally stored exactly, if we start all our vectors having for example a 0 y -component, then we know the state is non-chiral and can be stored exactly as non-chiral. We can generate similar states by rotation. Of course

⁹In this algorithm b_{12} is the original b vector projected in the 1-2 plane and $b_{rot,12}$ is the b vector extracted from S_2 and projected into the 1-2 plane.

these latter states cannot be computationally stored exactly as non-chiral due to finite precision, but we call them non-chiral up to that precision.

Events which are non-chiral but require a permutation to be mapped back to their original state after parity were also constructed by hand. Indeed, this is a drawback of the testing done on the logic statements, since we followed the structure of the logic statements themselves to find what makes a state non-chiral. In a way the construction of non-chiral states is circular. Concretely, the previous paragraph's idea for collision events emanates from case 1 in section 4.1. The reason it is a drawback is because while calculating expressions such as that in section 4.1 - that belong to the logic statement - we might have omitted a case of non-chirality. However, since we are also generating states randomly in our testing, nothing stops the latter from being non-chiral of any type and the algorithm - which covers all possibilities - will flag up any non-chiral/chiral states that evaluate to false/true on the logic statements derived in this paper - given the calculations of logic statements contain any errors. Of course we expect the randomly generated states to be chiral since it would be very unlikely to randomly generate a non-chiral one.

At various points in this work some algorithms that test chirality would label correctly a state non-chiral but that state would evaluate to false on the logic statement. This is due to the fact that we are doing comparisons of reals - indeed in the logic statements there is a substantial amount of such comparisons. In Python for example, $0.1 + 0.2 == 0.3$ gives false. To overcome this issue we have used the 'approx' function from the 'pytest' library [7], which uses a tolerance of 10^{-6} on equality comparison. A simple demonstration from real data follows.

Consider a and b from a collision event with $m_a = m_b = 3$. Here \vec{a} has been generated randomly and \vec{b} was obtained by rotating \vec{a} , hence we expect $a^2 == b^2$ to give true. Note that $a^2 == a \cdot a = \text{dot}(a, a)$ where dot is a custom function that performs the Minkowski dot product in Python.

$$\vec{a} = \begin{bmatrix} -2.0142326406285003 \\ 6.306119476657235 \\ -9.67682972129608 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} -3.4702690814617765 \\ 10.864649396714714 \\ 2.7169717385566954 \end{bmatrix}$$

a^2	b^2	$a^2 == b^2$	$a^2 == \text{approx}(b^2)$	Absolute Error
8.999999999980702	8.999999999980762	False	True	$1.7763568394002 \cdot 10^{-14}$

The absolute error is defined as $|a^2 - b^2|$ and emanates from rounding error while using finite arithmetic and from quantisation error due to the inexact computer representation of reals. This demonstration quantifies the magnitude of errors in our computational tests.

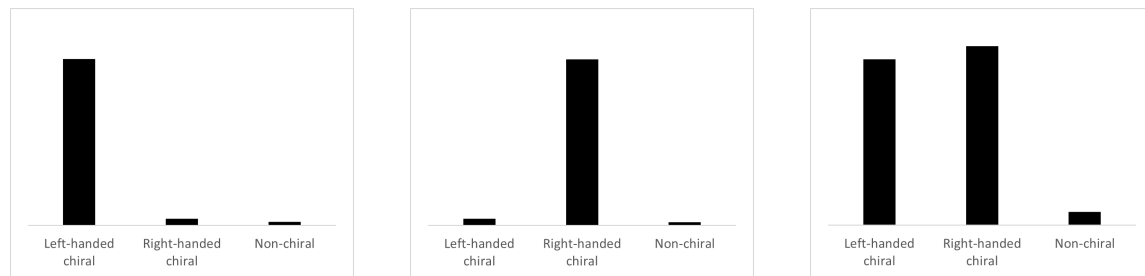
6 Geometric perspective

Following from the discussion in section 1.7 of [3], let H be the Lorentz group and G be a group such that $G/H \cong \mathbb{Z}/2$ - i.e. G contains an element that acts as parity. Let M be a 24-dimensional

manifold¹⁰ which represents the space of $2 \rightarrow 4$ events. The dimensionality comes from 6 masses and 6 3-momenta. For simplicity consider the compactification¹¹ of M into only collision events with certain energy ranges. By gathering data of such events from hadron colliders we statistically build a path that nature traverses on M .

We now look at the orbit space¹² $\mathcal{O} = M/H$. The latter contains two types of orbits, chiral and non-chiral. Let p be the non-trivial element of $\mathbb{Z}/2 \in G$. The chiral orbits are such that p acting on the orbit takes us off the orbit and into the other chiral orbit. We have two types of chiral orbits which we will call left and right handed. In \mathcal{O} the stabiliser set¹³ for chiral orbits thus consists only of the identity element. For non-chiral states, the stabiliser set is $\mathbb{Z}/2$ i.e. the orbit is fixed under both elements of $\mathbb{Z}/2$.

Now that we have labelled our orbit space we can visualize the path nature traverses on \mathcal{O} shown in figure 4. In order to know where each event in our samples lands on \mathcal{O} , we need to have a function that will distinguish chiral from non-chiral events - this is done with the logic statements derived above - and a function that will distinguish left-handed chiral from right-handed chiral. The latter set of functions are the parity odd event variables derived for example for $2 \rightarrow 3$ processes in [3].



(a) Most samples in the data lie in the left-handed orbit of chiral orbits which indicates parity violation.

(b) Most samples from the data lie in the right-handed orbit of chiral orbits which indicates parity violation.

(c) Even distribution of the two different types of handedness of chiral orbits which does not indicate any parity violation.

Figure 4: Examples of hypothetical data on the orbifold \mathcal{O} and their interpretation.

¹⁰An N dimensional manifold is a set equipped with a topology (which forms then a topological space) that can be locally mapped to \mathbb{R}^N . Locally implies over an open subset (no endpoints included) of M . A topology T on M is a set of subsets of M . The subsets must satisfy the following axioms: 1) M and the empty set are in T , 2) the intersection of any subsets in T is also in T , and 3) the union of any pair of subsets is also in T .

¹¹Restricting the set which forms the manifold to a compact set. A compact set is defined as closed (contains its endpoints) and bounded (finite cardinality).

¹²This is pronounced M modulo H and forms the set of equivalence classes such that in each class every element can be reached by the repeated application of a given $h \in H$.

¹³The stabiliser set for $\mathcal{O} = M/H$ is the subset of H that leaves an element of \mathcal{O} fixed.

7 Conclusion

In conclusion, we have looked at four different types of $2 \rightarrow 4$ events, namely collision events, non-collisions events with the two initial state particles being represented by massive, massless and zero 4-momenta. For each one of these types, we have calculated the logic statement which is true when evaluated on non-chiral events.

Further, the validity of these results was scrutinised using algorithms that utilize Geometric Algebra to label the chirality of randomly generated events. We have discussed the feasibility of generalising the results to N outgoing particles, with some cases requiring an explicit calculation to do so, while others were more straightforward to generalise.

The results presented in this paper can be used to generate parity-odd Lorentz invariant event variables that possess permutation symmetries between identical particles in the initial and final states, as demonstrated in [3]. These event variables can hint on non-standard parity violating processes through asymmetries in their distribution on hadron collider data.

8 Acknowledgements

I would like to thank Dr Christopher Lester for his invaluable support, advice and guidance throughout this project.

References

- [1] C. S. Wu, E. Ambler, R. W. Hayward, D. D. Hoppes, and R. P. Hudson. Experimental test of parity conservation in beta decay. *Phys. Rev.*, 105:1413–1415, Feb 1957.
- [2] Christopher G. Lester and Matthias Schott. Testing non-standard sources of parity violation in jets at the LHC, trialled with CMS open data. *Journal of High Energy Physics*, 2019(12), Dec 2019.
- [3] Christopher G. Lester, Ward Haddadin, and Ben Gripaios. Lorentz and permutation invariants of particles III: constraining non-standard sources of parity violation, 2020.
- [4] Chris Doran and Anthony Lasenby. *Geometric Algebra for Physicists*. Cambridge University Press, 2003.
- [5] Anthony N. Lasenby. Geometric Algebra as a Unifying Language for Physics and Engineering and Its Use in the Study of Gravity. *Adv. Appl. Clifford Algebras*, 27(1):733–759, 2017.
- [6] The Pygae Team. Clifford: Numerical geometric algebra module for python. <https://github.com/pygae/clifford>.
- [7] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laughner, and Florian Bruhin. *pytest x.y*, 2004.

- [8] M. Hohenwarter, M. Borchers, G. Ancsin, B. Bencze, M. Blossier, A. Delobelle, C. Denizet, J. Éliás, Á Fekete, L. Gál, Z. Konečný, Z. Kovács, S. Lizelfelner, B. Parisse, and G. Sturr. GeoGebra 4.4, December 2013. <http://www.geogebra.org>.
- [9] Takis Angelides. Part III project. <https://github.com/TakisAngelides/Part-3-Project>, 2021.

A From events to event variables

Here we present a diagram describing in steps the method we use, going from a set of events to building the event variables. The blue part of the diagram is what we have presented in this paper and the red part is what remains for future work.

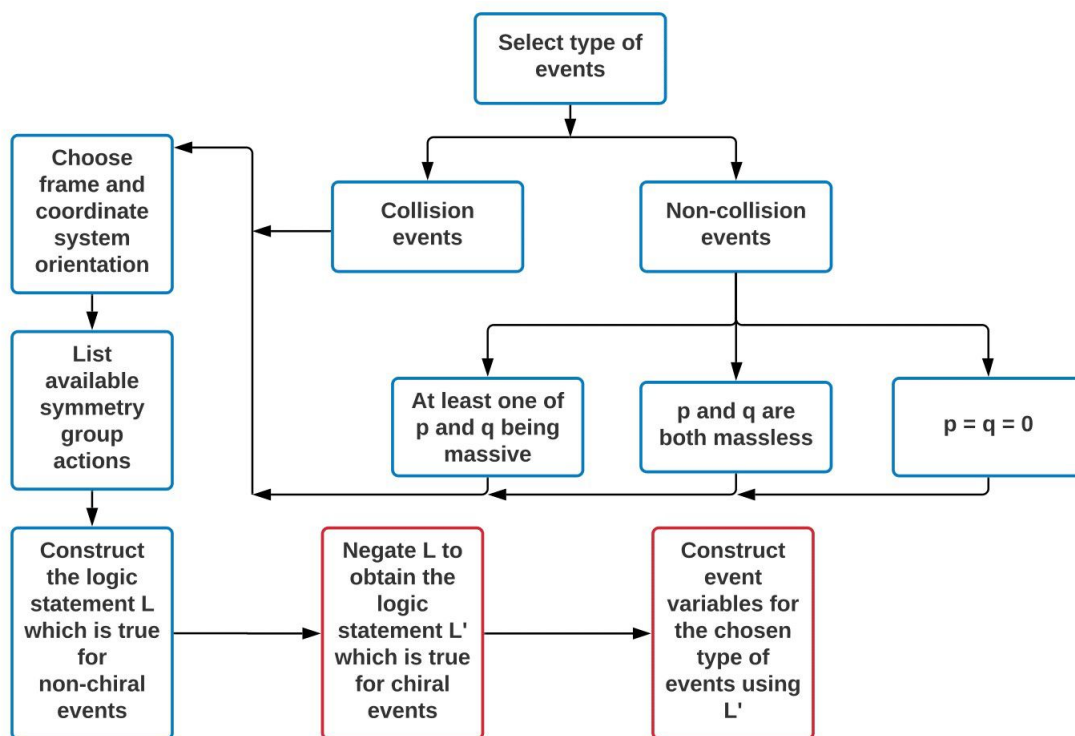


Figure 5: A flow chart of the method to go from events to event variables. The blue part is presented in this paper and the red part remains for future work.

B Proofs for algorithms

In this section we present proofs that the algorithms in section 5 cover all possible inputs, i.e. they can label correctly any state their given either ‘chiral’ or ‘non-chiral’. The proofs work with

3-momenta, as do the algorithms, since we have chosen our coordinate system and frame so as to avoid having boosts in our symmetry group.

B.1 Covering of algorithm for collision events

For this type of events, we have p, q aligned with the z-axis in the $(p+q)$ rest frame. Our symmetry actions exclude boosts.

Claim: The algorithm in section 5.2 with its generalisation found in code form in appendix E cover all possible collision event inputs to correctly label them chiral or non-chiral.

Proof: For collision events with permutations we split the proof into two cases of 1_{S_2} or (pq) .

For the case of 1_{S_2} , we map p and q back with $R_1 = e_1e_3$. Now the available subspace for rotations is the 1-2 plane. We map the (1-2)-projection of a , namely a_{12} , to its original or for any final state particle x that can be permuted with a , we map the x_{12} to the original a_{12} and perform the permutation (ax) . The only degrees of freedom left are permutations between the final state particles excluding a , if any are available. If a is collinear with the 3-axis or a is zero, we repeat the above with b instead of a . If the latter is true for b , we repeat the above with c and if c is also collinear with the 3-axis or zero, we output ‘non-chiral’ and terminate. Checking for non-chirality with all the aforementioned actions achieves exhaustion for the case of 1_{S_2} .

For the case when we have (pq) available, we can try the above or we can repeat the above but omit R_1 and instead perform (pq) . Whenever we check for non-chirality, we can also perform a π rotation in the plane that contains the 3-axis and has normal a_{12} (or any other final state particle we matched to its original in the 1-2 plane), but then we also perform the (pq) swap. The same caveat of a being collinear with the 3-axis or being zero applies here as well. Exhausting degrees of freedom - actions of the symmetry group that leave already matched momenta fixed - ensures covering of the algorithm for any possible input. □

B.2 Covering of algorithm for non-collision events with at least one of p and q being massive

For this type of cases we have p and q being stationary, so we are concerned with mapping the final state particles back to their original state with general rotations.

Claim: The algorithm in section 5.4 with its generalisation found in code form in appendix E cover all possible non-collision event inputs (with at least one massive initial particle) to correctly label them chiral or non-chiral.

Proof: For every particle x that can be permuted with a , including a itself, we map x to a and perform (ax) . Now that a is fixed, we can only rotate in the plane perpendicular a and do permutations between $\{b, c, d\}$. In what follows we are working in the plane perpendicular to a and w.l.o.g. assume that b has components in that plane. For every particle $y \in \{b, c, d\}$ that can be permuted with b , we map y to b and perform (by) . If b is collinear to a , we repeat the latter

with c instead of b , otherwise output ‘non-chiral’ (since 3 out of 4 are collinear). The only degree of freedom left is (cd) , which we also check. We have now exhausted all degrees of freedom and therefore covered all possible inputs. □

B.3 Covering of algorithm for non-collision events with both p and q being massless

For these cases our symmetry group excludes boosts, (pq) swaps and any rotation other than about the z -axis.

Claim: The algorithm in section 5.6 and its generalisation found in code form in appendix E cover all possible non-collision event inputs (with both p and q being massless) to correctly label them chiral or non-chiral.

Proof: Our initial degrees of freedom are rotations about the z -axis and permutations between $\{a, b, c, d\}$. W.l.o.g. we assume that a has permutations and components in the 1-2 plane. Further, all that follows is in the 1-2 plane where rotations are initially allowed. For every particle x that can be permuted with a - including a itself -, we map x to a and perform (ax) . Now, the only degrees of freedom are permutations between $\{b, c, d\}$, which we also check. At this point exhaustion of possible actions has been achieved. □

B.4 Covering of algorithm for non-collision events with $p = q = 0$

Claim: The algorithm in section 5.8 and its generalisation found in code form in appendix E cover all possible non-collision event inputs (with $p = q = 0$) to correctly label them chiral or non-chiral.

Proof: We orient a back to its original state so that the only degrees of freedom are rotations in the subspace perpendicular to a , namely the 1-2 plane, and permutations between $\{b, c, d\}$ if any. In what follows, we are working with vectors projected into the 1-2 plane, leaving the z -component fixed. W.l.o.g. we assume that b has permutations and components in the 1-2 plane. For every particle x that can be permuted with b - including b itself -, we map x to b and perform (bx) , while also checking for (cd) . There are no other degrees of freedom left and thus we have achieved exhaustion. □

C Notation

Following the notation and conventions found in the appendices of [3], we present here the meaning of some notation frequently used in this paper. The G stands for Gram determinant.

$$G\begin{pmatrix} a, b \\ c, d \end{pmatrix} \equiv \begin{vmatrix} a \cdot c & a \cdot d \\ b \cdot c & b \cdot d \end{vmatrix} = (a \cdot c)(b \cdot d) - (b \cdot c)(a \cdot d)$$

$$\Delta_2(a, b) \equiv G \begin{pmatrix} a, b \\ a, b \end{pmatrix}$$

$$\Delta_3(a, b, c) \equiv G \begin{pmatrix} a, b, c \\ a, b, c \end{pmatrix} = \begin{vmatrix} a \cdot a & a \cdot b & a \cdot c \\ b \cdot a & b \cdot b & b \cdot c \\ c \cdot a & c \cdot b & c \cdot c \end{vmatrix}$$

D Case 2 calculation for collision events

This case has $h = 1_{S_2} \cdot (ab)$ and we will go through the explicit calculation of equation (4).

Using

$$\begin{pmatrix} \theta \\ \curvearrowright \\ xy \end{pmatrix} \cdot h \cdot \hat{e} = \begin{pmatrix} \pi \\ \curvearrowright \\ yz \end{pmatrix} \cdot \mathcal{P} \cdot \hat{e} \quad (36)$$

we start from

$$[(m_a = m_b) \wedge (a_z = b_z) \wedge (|\mathbf{a}| = |\mathbf{b}|)] \wedge \quad (37)$$

$$[(|\mathbf{a}| = |\mathbf{b}| = 0) \vee ((\theta + \beta = \pi + 2\pi n_a - \alpha) \wedge (\theta + \alpha = \pi + 2\pi n_b - \beta))] \wedge \quad (38)$$

$$[(|\mathbf{c}| = 0) \vee (\theta + \gamma = \pi + 2\pi n_c - \gamma)] \wedge \quad (39)$$

$$[(|\mathbf{d}| = 0) \vee (\theta + \delta = \pi + 2\pi n_d - \delta)] \quad (40)$$

We now call N the whole bracket of line (37). We also name A_1 the first term in the bracket of line (38) and A_2 the second term in that bracket, namely $A_2 = (\theta + \beta = \pi + 2\pi n_a - \alpha) \wedge (\theta + \alpha = \pi + 2\pi n_b - \beta)$. The same for the rest of the lines using B and C .

Now we have 8 subcases which correspond to choosing one of the two terms in each of the lines (38) to (40). The table below enumerates these subcases.

1)	$A_1 B_1 C_1$	5)	$A_2 B_1 C_1$
2)	$A_1 B_2 C_1$	6)	$A_2 B_2 C_1$
3)	$A_1 B_1 C_2$	7)	$A_2 B_1 C_2$
4)	$A_1 B_2 C_2$	8)	$A_2 B_2 C_2$

D.1 Subcase 1: $A_1 B_1 C_1$

This subcase has $|\mathbf{a}| = |\mathbf{b}| = |\mathbf{c}| = |\mathbf{d}| = 0 \Rightarrow$ all 3-vectors of the final state live on the z-axis which is already covered by case 1 and can thus be discarded.

D.2 Subcases 2-4

These subcases are also covered by case 1 and can be discarded. In each one of them we find that all final state 3-vectors lie in the same plane.

D.3 Subcase 5: $A_2B_1C_1$

Remember N refers to line (37). Subcase 5 has $N \wedge (\theta + \beta = \pi + 2\pi n_a - \alpha) \wedge (\theta + \alpha = \pi + 2\pi n_b - \beta) \wedge (|\mathbf{c}| = 0) \wedge (|\mathbf{d}| = 0)$. The two brackets containing angles give that $\alpha + \beta = \pi - \theta + \pi(n_a + n_b)$. However, we can always find a θ in the symmetry group to satisfy the condition and hence effectively α and β are unconstrained, but we must $(|\mathbf{a}| \neq 0) \wedge (|\mathbf{b}| \neq 0)$. A general event of this subcase can be described by $N \wedge (|\mathbf{a}| \neq 0) \wedge (|\mathbf{b}| \neq 0) \wedge (|\mathbf{c}| = 0) \wedge (|\mathbf{d}| = 0)$. Note that for example $|\mathbf{a}| \neq 0$ can be written in a Lorentz invariant form using $\Delta_3(a, p, q) \neq 0$, which follows from Lemma 3.34 of [3].

D.4 Subcase 6: $A_2B_2C_1$

We start with $N \wedge (\theta + \beta = \pi + 2\pi n_a - \alpha) \wedge (\theta + \alpha = \pi + 2\pi n_b - \beta) \wedge (\gamma = \frac{1}{2}(\pi - \theta) + 2\pi n_c) \wedge (|\mathbf{d}| = 0)$. The bracket with the angle γ has just been algebraically rearranged from line (39) - remember the Greek letter angles refer to angles in the transverse plane perpendicular to the z-axis. Now, the brackets with angles involved give the following with a possible solution for the angles α, β, γ shown on the right hand side

$$\begin{aligned} \alpha + \beta &= \pi + 2\pi n_a - \theta & \alpha &= \pi - \theta + 2\pi n_a \\ \alpha + \beta &= \pi + 2\pi n_b - \theta & \beta &= 0 \\ \gamma &= \frac{1}{2}(\pi - \theta) + 2\pi n_c & \gamma &= \frac{1}{2}(\pi - \theta) + 2\pi n_c \end{aligned}$$

W.l.o.g. setting α and γ anywhere in the transverse plane, β is forced to be the reflection of α in the line defined by γ - see figure 6. Hence we choose $G \begin{pmatrix} a-b, p+q \\ c, p+q \end{pmatrix} = 0$ to describe this. To see the latter we expand the Gram determinant which gives $0 = (a-b) \cdot c = (m_a - m_b)m_c - (\vec{a} - \vec{b}) \cdot \vec{c} = -(\vec{a} - \vec{b}) \cdot \vec{c} = -(\mathbf{a} - \mathbf{b}) \cdot \mathbf{c} = 0$. The last equality follows from the condition $a_z = b_z$ and we also use the condition $m_a = m_b$.

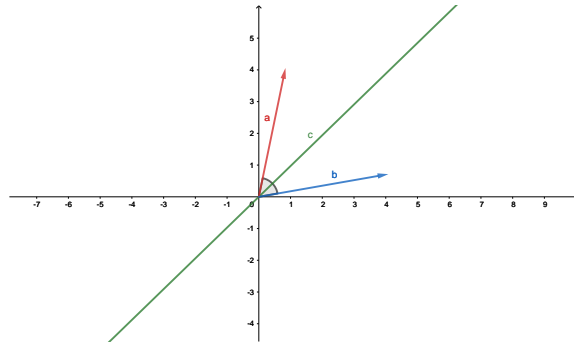


Figure 6: An example of \mathbf{a} and \mathbf{b} being reflections of each other in the ray defined by \mathbf{c} . The figure was constructed using [8].

D.5 Subcase 7: $A_2B_1C_2$

This subcase follows subcase 6 by swapping c with d .

D.6 Subcase 8: $A_2B_2C_2$

This subcase has $N \wedge (\theta + \beta = \pi + 2\pi n_a - \alpha) \wedge (\theta + \alpha = \pi + 2\pi n_b - \beta) \wedge (\theta + \gamma = \pi + 2\pi n_c - \gamma) \wedge (\theta + \delta = \pi + 2\pi n_d - \delta)$. We can see immediately that the brackets other than N give

$$\begin{aligned} n_a &= n_b \\ \gamma - \delta &= 2\pi(n_c - n_d) = 0 \pmod{2\pi} \\ \gamma - \frac{1}{2}(\alpha + \beta) &= \pi(n_a - n_c) \\ \delta - \frac{1}{2}(\alpha + \beta) &= \pi(n_a - n_d) \\ \Rightarrow \gamma &= \delta \pmod{\pi} \end{aligned}$$

This is telling us that \mathbf{c} and \mathbf{d} form a ray in the transverse plane in which \mathbf{a} and \mathbf{b} are reflections of each other. We can describe this in a Lorentz invariant form as $((G \begin{pmatrix} a-b, p+q \\ c, p+q \end{pmatrix} = 0) \wedge (G \begin{pmatrix} a-b, p+q \\ d, p+q \end{pmatrix} = 0)) \Rightarrow G \begin{pmatrix} a-b, p+q \\ c-d, p+q \end{pmatrix} = 0$.

D.7 Final result

Gathering all the results from these subcases, we combine them to form the result presented in equation (4). □

E Code

Below we present the full code for all types of events considered. The titles of the files correspond to the initial state particles and the code can also be found in [9].

Listings

collision_events_algorithm.py	29
non_collision_one_massive_algorithm.py	45
non_collision_massless_algorithm.py	50
non_collision_zeros_algorithm.py	55

Code file: collision_events_algorithm.py

```

1 import time
2 from numpy import pi, cos, sin, e, tan, arctan
3 from clifford.g3 import blades
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 import numpy as np
7 from random import uniform, seed, randint
8 from sympy import LeviCivita as eps
9
10 # fig = plt.figure()
11 # ax = fig.add_subplot(111, projection='3d')
```

```

12
13 e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
14 I = e1*e2*e3 # Pseudoscalar of 3D Euclidean geometric algebra
15
16 def plot_state(ax, p, q, a, b, c, d):
17
18     X = (0)
19     Y = (0)
20     Z = (0)
21
22     p1, p2, p3 = p[0], p[1], p[2]
23     q1, q2, q3 = q[0], q[1], q[2]
24     a1, a2, a3 = a[0], a[1], a[2]
25     b1, b2, b3 = b[0], b[1], b[2]
26     c1, c2, c3 = c[0], c[1], c[2]
27     d1, d2, d3 = d[0], d[1], d[2]
28
29     ax.quiver(X, Y, Z, p1, p2, p3, color='r', linestyle='-', label='p')
30     ax.quiver(X, Y, Z, q1, q2, q3, color='k', linestyle='-', label='q')
31     ax.quiver(X, Y, Z, a1, a2, a3, color='b', linestyle='-', label='a')
32     ax.quiver(X, Y, Z, b1, b2, b3, color='g', linestyle='-', label='b')
33     ax.quiver(X, Y, Z, c1, c2, c3, color='y', linestyle='-', label='c')
34     ax.quiver(X, Y, Z, d1, d2, d3, color='orange', linestyle='-', label='d')
35     ax.set_xlabel('x')
36     ax.set_ylabel('y')
37     ax.set_zlabel('z')
38     ax.set_xticks([])
39     ax.set_yticks([])
40     ax.set_zticks([])
41     limit = 10
42     ax.set_xlim([-limit, limit])
43     ax.set_ylim([-limit, limit])
44     ax.set_zlim([-limit, limit])
45     ax.view_init(elev=1, azim=pi / 2)
46     plt.legend()
47
48 def plot_parity_state(ax, p, q, a, b, c, d):
49
50     X = (0)
51     Y = (0)
52     Z = (0)
53
54     p1, p2, p3 = -p[0], -p[1], -p[2]
55     q1, q2, q3 = -q[0], -q[1], -q[2]
56     a1, a2, a3 = -a[0], -a[1], -a[2]
57     b1, b2, b3 = -b[0], -b[1], -b[2]
58     c1, c2, c3 = -c[0], -c[1], -c[2]
59     d1, d2, d3 = -d[0], -d[1], -d[2]
60
61     ax.quiver(X, Y, Z, p1, p2, p3, color='r', linestyle='--', label='p')
62     ax.quiver(X, Y, Z, q1, q2, q3, color='k', linestyle='--', label='q')
63     ax.quiver(X, Y, Z, a1, a2, a3, color='b', linestyle='--', label='a')
64     ax.quiver(X, Y, Z, b1, b2, b3, color='g', linestyle='--', label='b')
65     ax.quiver(X, Y, Z, c1, c2, c3, color='y', linestyle='--', label='c')
66     ax.quiver(X, Y, Z, d1, d2, d3, color='orange', linestyle='--', label='d')
67     ax.set_xlabel('x')
68     ax.set_ylabel('y')
69     ax.set_zlabel('z')
70     ax.set_xticks([])
71     ax.set_yticks([])
72     ax.set_zticks([])
73     limit = 10
74     ax.set_xlim([-limit, limit])
75     ax.set_ylim([-limit, limit])
76     ax.set_zlim([-limit, limit])
77     ax.view_init(elev=1, azim=pi / 2)
78     plt.legend()
79     plt.show()
80

```

```

81 # -----
82
83 # For a collision event:
84
85 def parity(S):
86     return [-v for v in S]
87
88 def rotate(S, R):
89     return [R*v~R for v in S]
90
91 def multivec_to_vec(a):
92     return np.array([a[1], a[2], a[3]])
93
94 def energy(m,p):
95     p = multivec_to_vec(p)
96     return np.sqrt(m**2 + np.linalg.norm(p)**2)
97
98 def epsilon(a, b, c, d):
99
100     summation = 0
101
102     for i in range(0, 4):
103         for j in range(0, 4):
104             for k in range(0, 4):
105                 for l in range(0, 4):
106                     summation += eps(i, j, k, l) * a[i] * b[j] * c[k] * d[l]
107
108     return summation
109
110 def dot(a, b):
111
112     # Minkowski metric
113
114     return a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3]
115
116 def Gram_det_2(a,b,c,d):
117
118     # a b
119     # c d
120
121     return (dot(a, c))*(dot(b, d)) - (dot(a, d))*(dot(b, c))
122
123 def sym_2_Gram_det(a,b):
124     return Gram_det_2(a,b,a,b)
125
126 def sym_3_Gram_det(a,b,c):
127     M = [[dot(a,a),dot(a,b),dot(a,c)], [dot(b,a),dot(b,b),dot(b,c)], [dot(c,a),dot(c,b),dot(c,c)]]
128     return np.linalg.det(M)
129
130 def swap(S,idx_1,idx_2):
131
132     tmp = S[idx_1]
133     S[idx_1] = S[idx_2]
134     S[idx_2] = tmp
135
136     return S
137
138 def permute_with_idx(M, E, idx_to_permute):
139
140     same_mass_with_idx = [idx for idx in range(len(M)) if M[idx] == M[idx_to_permute] and idx !=
141         idx_to_permute and idx != 0 and idx != 1]
142     same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == E[idx_to_permute] and idx
143         != idx_to_permute]
144
145     return list(set(same_mass_with_idx) and set(same_energy_with_idx))
146
147 def permutation_boolean(M, E, idx_1, idx_2):
148
149     if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):

```

```

148     return True
149 else:
150     return False
151
152 def logic_statement_true_for_non_chiral(S, E):
153
154     p = multivec_to_vec(S[0])
155     p = np.insert(p, 0, E[0])
156     q = multivec_to_vec(S[1])
157     q = np.insert(q, 0, E[1])
158     a = multivec_to_vec(S[2])
159     a = np.insert(a, 0, E[2])
160     b = multivec_to_vec(S[3])
161     b = np.insert(b, 0, E[3])
162     c = multivec_to_vec(S[4])
163     c = np.insert(c, 0, E[4])
164     d = multivec_to_vec(S[5])
165     d = np.insert(d, 0, E[5])
166     RF = p + q
167
168     case_1 = ((epsilon(a,b,p,q) == 0) and (epsilon(a,c,p,q) == 0) and (epsilon(a,d,p,q) == 0)
169              and (epsilon(b,d,p,q) == 0)
170              and (epsilon(b,c,p,q) == 0) and (epsilon(c,d,p,q) == 0))
171
172     case_2 = ((dot(a,a) == dot(b,b)) and (Gram_det_2(a-b,RF,p-q,RF) == 0) and (Gram_det_2(a-b,
173              RF,a+b,RF) == 0)
174              and (((sym_3_Gram_det(a,p,q) == 0) and (sym_3_Gram_det(b,p,q) == 0) and (
175              sym_3_Gram_det(c,p,q) != 0) and (sym_3_Gram_det(d,p,q) == 0))
176              or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
177              sym_3_Gram_det(c,p,q) != 0) and (sym_3_Gram_det(d,p,q) == 0) and (Gram_det_2(a-b,RF,c,RF)
178              == 0))
179              or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
180              sym_3_Gram_det(d,p,q) != 0) and (sym_3_Gram_det(c,p,q) == 0) and (Gram_det_2(a-b,RF,d,RF)
181              == 0))
182              or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
183              sym_3_Gram_det(d,p,q) != 0) and (sym_3_Gram_det(c,p,q) != 0) and (Gram_det_2(a-b,RF,c,RF)
184              == 0) and (Gram_det_2(a-b,RF,d,RF) == 0))))))
185
186     def case_2_symmetry(p, q, a, b, c, d):
187
188         return ((dot(a,a) == dot(b,b)) and (Gram_det_2(a-b,RF,p-q,RF) == 0) and (Gram_det_2(a-b,
189         RF,a+b,RF) == 0)
190         and (((sym_3_Gram_det(a,p,q) == 0) and (sym_3_Gram_det(b,p,q) == 0) and (
191         sym_3_Gram_det(c,p,q) != 0) and (sym_3_Gram_det(d,p,q) == 0))
192         or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
193         sym_3_Gram_det(c,p,q) != 0) and (sym_3_Gram_det(d,p,q) == 0) and (Gram_det_2(a-b,RF,c,RF)
194         == 0))
195         or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
196         sym_3_Gram_det(d,p,q) != 0) and (sym_3_Gram_det(c,p,q) == 0) and (Gram_det_2(a-b,RF,d,RF)
197         == 0))
198         or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (
199         sym_3_Gram_det(d,p,q) != 0) and (sym_3_Gram_det(c,p,q) != 0) and (Gram_det_2(a-b,RF,c,RF)
200         == 0) and (Gram_det_2(a-b,RF,d,RF) == 0))))))
201
202     case_3 = case_2_symmetry(p, q, c, d, a, b) # h = (cd)
203     case_4 = case_2_symmetry(p, q, b, c, a, d) # h = (bc)
204     case_5 = case_2_symmetry(p, q, b, d, a, c) # h = (bd)
205     case_6 = case_2_symmetry(p, q, a, c, b, d) # h = (ac)
206     case_7 = case_2_symmetry(p, q, a, d, c, b) # h = (ad)
207
208     # TODO: In case 16 green and blue bracket are the negation of each other so I can evaluate
209     # once for efficiency
210
211     case_16 = ((dot(a,a) == dot(b,b)) and (dot(c,c) == dot(d,d)) and (Gram_det_2(a-b,RF,p-q,RF)
212              == 0) and (Gram_det_2(a-b,RF,a+b,RF) == 0) and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (
213              Gram_det_2(c-d,RF,c+d,RF) == 0)
214              and (((sym_3_Gram_det(a,p,q)==0) and (sym_3_Gram_det(b,p,q)==0) and (
215              sym_3_Gram_det(c,p,q)!=0) and (sym_3_Gram_det(d,p,q)!=0) or (sym_3_Gram_det(c,p,q)==0) and
216              (sym_3_Gram_det(d,p,q)==0) and (sym_3_Gram_det(a,p,q)!=0) and (sym_3_Gram_det(b,p,q)!=0))

```



```

195         or ((Gram_det_2(a-b,RF,c+d,RF)==0) or (Gram_det_2(c-d,RF,b,RF)==0) or (
Gram_det_2(a-b,RF,d,RF)==0)))
196
197 def case_16_symmetry(p,q,a,b,c,d):
198
199     return ((dot(a,a) == dot(b,b)) and (dot(c,c) == dot(d,d)) and (Gram_det_2(a-b,RF,p-q,RF)
== 0) and (Gram_det_2(a-b,RF,a+b,RF) == 0) and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (
Gram_det_2(c-d,RF,c+d,RF) == 0)
200         and (((sym_3_Gram_det(a,p,q)==0) and (sym_3_Gram_det(b,p,q)==0) and (
sym_3_Gram_det(c,p,q)!=0) and (sym_3_Gram_det(d,p,q)!=0) or (sym_3_Gram_det(c,p,q)==0) and
(sym_3_Gram_det(d,p,q)==0) and (sym_3_Gram_det(a,p,q)!=0) and (sym_3_Gram_det(b,p,q)!=0))
201         or ((Gram_det_2(a-b,RF,c+d,RF)==0) or (Gram_det_2(c-d,RF,b,RF)==0) or (
Gram_det_2(a-b,RF,d,RF)==0))))
202
203 case_17 = case_16_symmetry(p, q, a, c, b, d) # h = (ac)(bd)
204 case_18 = case_16_symmetry(p, q, a, d, c, b) # h = (ad)(bc)
205
206 case_19 = ((dot(a,a) == dot(b,b) == dot(c,c) == dot(d,d)) and (Gram_det_2(a-b,RF,a+b,RF) ==
0) and (Gram_det_2(b-c,RF,b+c,RF) == 0) \
207         and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a-b,RF,p-q,RF) == 0) and (
Gram_det_2(b-c,RF,p-q,RF) == 0) \
208         and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (a == c) and (b == d))
209
210 def case_19_symmetry(p,q,a,b,c,d):
211
212     return ((dot(a,a) == dot(b,b) == dot(c,c) == dot(d,d)) and (Gram_det_2(a-b,RF,a+b,RF) ==
0) and (Gram_det_2(b-c,RF,b+c,RF) == 0) \
213         and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a-b,RF,p-q,RF) == 0) and (
Gram_det_2(b-c,RF,p-q,RF) == 0) \
214         and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (a == c) and (b == d))
215
216 case_20 = case_19_symmetry(p,q,a,b,d,c) # h = (dbac)
217 case_22 = case_19_symmetry(p,q,a,c,d,b) # h = (bcad)
218
219 case_25 = ((dot(p,p) == dot(q,q)) and (Gram_det_2(a,RF,p-q,RF) == 0) and (Gram_det_2(b,RF,p-
q,RF) == 0) \
220         and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0))
221
222 case_26 = ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b)) and (Gram_det_2(a+b,RF,p-q,RF)
== 0) and (Gram_det_2(a-b,RF,a+b,RF) == 0)
223         and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0)
224         and (sym_3_Gram_det(a,p,q) == 0 or sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det
(a-b,p,q) == 0))
225
226 def case_26_symmetry(p,q,a,b,c,d):
227
228     return ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b)) and (Gram_det_2(a+b,RF,p-q,RF)
== 0) and (Gram_det_2(a-b,RF,a+b,RF) == 0)
229         and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0)
230         and (sym_3_Gram_det(a,p,q) == 0 or sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det
(a-b,p,q) == 0))
231
232 case_27 = case_26_symmetry(p, q, c, d, a, b) # h = (cd)
233 case_28 = case_26_symmetry(p, q, c, b, a, d) # h = (bc)
234 case_29 = case_26_symmetry(p, q, d, b, c, a) # h = (bd)
235 case_30 = case_26_symmetry(p, q, a, c, b, d) # h = (ac)
236 case_31 = case_26_symmetry(p, q, a, d, c, b) # h = (ad)
237
238
239 case_32 = ((dot(p,p) == dot(q,q)) and (dot(b,b) == dot(c,c) == dot(d,d)) and (Gram_det_2(a,
RF,p-q,RF) == 0) and (Gram_det_2(b,RF,p-q,RF) == 0) \
240         and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0) and (
Gram_det_2(b-c,RF,b+c,RF) == 0) \
241         and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a,RF,p-q,RF) == 0) and (
epsilon((b+d),c,p,q) == 0) \
242         and (epsilon((b+c),d,p,q) == 0) and (epsilon((c+d),b,p,q) == 0))
243
244 def case_32_symmetry(p,q,a,b,c,d):
245

```

```

246     return ((dot(p,p) == dot(q,q)) and (dot(b,b) == dot(c,c) == dot(d,d)) and (Gram_det_2(a,
RF,p-q,RF) == 0) and (Gram_det_2(b,RF,p-q,RF) == 0) \
247     and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0) and (
Gram_det_2(b-c,RF,b+c,RF) == 0) \
248     and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a,RF,p-q,RF) == 0) and (
epsilon((b+d),c,p,q) == 0) \
249     and (epsilon((b+c),d,p,q) == 0) and (epsilon((c+d),b,p,q) == 0))
250
251 case_34 = case_32_symmetry(p, q, d, b, c, a) # h = (cba)
252 case_35 = case_32_symmetry(p, q, c, a, d, b) # h = (dba)
253 case_37 = case_32_symmetry(p, q, b, a, c, d) # h = (dca)
254
255 case_40 = ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b)) and (dot(c,c) == dot(d,d)) and
(Gram_det_2(a+b,RF,p-q,RF) == 0) \
256     and (Gram_det_2(c+d,RF,p-q,RF) == 0) \
257     and ((sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det(a-b,p,q) == 0)
258     or ((sym_3_Gram_det(c+d,p,q) == 0) or (sym_3_Gram_det(c-d,p,q) == 0))))
259
260 def case_40_symmetry(p,q,a,b,c,d):
261
262     return ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b)) and (dot(c,c) == dot(d,d)) and
(Gram_det_2(a+b,RF,p-q,RF) == 0) \
263     and (Gram_det_2(c+d,RF,p-q,RF) == 0) \
264     and ((sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det(a-b,p,q) == 0)
265     or ((sym_3_Gram_det(c+d,p,q) == 0) or (sym_3_Gram_det(c-d,p,q) == 0))))
266
267 case_41 = case_40_symmetry(p, q, a, c, b, d) # h = (ac)(bd)
268 case_42 = case_40_symmetry(p, q, a, d, c, b) # h = (ad)(cb)
269
270 case_43 = ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b) == dot(c,c) == dot(d,d)) and (
Gram_det_2(a-b,RF,a+b,RF) == 0)
271 and (Gram_det_2(b-c,RF,b+c,RF) == 0) and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a+
b,RF,p-q,RF) == 0) and (Gram_det_2(b+c,RF,p-q,RF) == 0)
272 and (Gram_det_2(c+d,RF,p-q,RF) == 0) and ((sym_3_Gram_det(a,p,q) == 0)
273 or ((epsilon(p+q,a-c,p,b) == 0) and (epsilon(p+q,b-d,a,p) == 0))
274 or ((sym_3_Gram_det(a-c,p,q) == 0) and (sym_3_Gram_det(b-d,p,q) == 0)
275 and ((sym_3_Gram_det(a+b,p,q) == 0) or ((Gram_det_2(a+b,RF,p-q,RF) == 0) and (
sym_3_Gram_det(a-b,p,q) == 0))))))
276
277 def case_43_symmetry(p,q,a,b,c,d):
278
279     return ((dot(p,p) == dot(q,q)) and (dot(a,a) == dot(b,b) == dot(c,c) == dot(d,d)) and (
Gram_det_2(a-b,RF,a+b,RF) == 0)
280 and (Gram_det_2(b-c,RF,b+c,RF) == 0) and (Gram_det_2(c-d,RF,c+d,RF) == 0) and (Gram_det_2(a+
b,RF,p-q,RF) == 0) and (Gram_det_2(b+c,RF,p-q,RF) == 0)
281 and (Gram_det_2(c+d,RF,p-q,RF) == 0) and ((sym_3_Gram_det(a,p,q) == 0)
282 or ((epsilon(p+q,a-c,p,b) == 0) and (epsilon(p+q,b-d,a,p) == 0))
283 or ((sym_3_Gram_det(a-c,p,q) == 0) and (sym_3_Gram_det(b-d,p,q) == 0)
284 and ((sym_3_Gram_det(a+b,p,q) == 0) or ((Gram_det_2(a+b,RF,p-q,RF) == 0) and (
sym_3_Gram_det(a-b,p,q) == 0))))))
285
286 case_44 = case_43_symmetry(p, q, a, b, d, c) # h = (dbac)
287 case_45 = case_43_symmetry(p, q, b, a, c, d) # h = (dcab)
288 case_46 = case_43_symmetry(p, q, a, c, b, d) # h = (dbca)
289 case_47 = case_43_symmetry(p, q, c, b, a, d) # h = (dabc)
290 case_48 = case_43_symmetry(p, q, b, c, a, d) # h = (dacb)
291
292 return (case_1 or case_2 or case_3 or case_4 or case_5 or case_6 or case_7 or case_16 or
case_17
293 or case_18
294 or case_19 or case_20 or case_22 or case_25 or case_26 or case_27 or case_28 or
case_29
295 or case_30
296 or case_31 or case_32 or case_34 or case_35 or case_37 or case_40 or case_41 or
case_42
297 or case_43
298 or case_44 or case_45 or case_46 or case_47 or case_48)
299
def construct_state():
mp, mq, ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10),
randint(0, 10), randint(0,10)

```

```

300 p = uniform(-10, 10) * e3
301 q = -p
302
303
304 a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
305 b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
306 c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
307 d = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
308
309 Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(
mc, c), energy(md, d)
310
311 M = [mp, mq, ma, mb, mc, md]
312 E = [Ep, Eq, Ea, Eb, Ec, Ed]
313 S = [p, q, a, b, c, d]
314
315 return S, E, M
316
317 def construct_non_chiral_state(): # Trick
318
319 mp, mq, ma, mb, mc, md = 1, 1, 1, 1, 1, 1
320
321 p = uniform(-10, 10) * e3
322 q = -p
323
324 a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
325 b = a[1] * e1 - a[2] * e2 + a[3] * e3
326 R = e**(uniform(0,2*pi)*e1*e2)
327 c = R*a*~R
328 d = c[1] * e1 - c[2] * e2 + c[3] * e3
329
330 Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(
mc, c), energy(md, d)
331
332 M = [mp, mq, ma, mb, mc, md]
333 E = [Ep, Eq, Ea, Eb, Ec, Ed]
334 S = [p, q, a, b, c, d]
335
336 return S, E, M
337
338 def chirality_test():
339
340 chiral_states = []
341 non_chiral_states = []
342
343 S, E, M = construct_non_chiral_state()
344 a, b, c, d = S[2], S[3], S[4], S[5]
345
346 # These lists hold indices (as they appear in S) of the particles that can be permuted with
a,b,c and d respectively
347 permute_with_a = permute_with_idx(M, E, 2)
348 permute_with_b = permute_with_idx(M, E, 3)
349 permute_with_c = permute_with_idx(M, E, 4)
350 permute_with_d = permute_with_idx(M, E, 5)
351
352 permutation_dictionary = {3: permute_with_b, 4: permute_with_c, 5: permute_with_d}
353
354 S_parity = parity(S) # Perform parity on the set of momenta
355
356 flag = False
357
358 if not permutation_boolean(M, E, 0, 1): # If we cant permute p (index 0) with q (index 1)
359
360     if (a[1] == 0 and a[2] == 0) or (a == 0):
361
362         if (b[1] == 0 and b[2] == 0) or (b == 0):
363
364             if (c[1] == 0 and c[2] == 0) or (c == 0):
365

```

```

366         flag = True
367
368     else: # Here we consider that a,b are either e3 collinear or 0, only c and d are
in the 1-2 plane
369
370     # We can map c_12 to its original or d_12 to the original c_12 and swap (cd)
if they can be
371     # permuted.
372
373     # Since a and b are fixed by R1 = e1e3 we only want to consider permutations
(cd)
374     # Remember that idx 2 corresponds to a and idx 3 corresponds to b in S
375     permute_with_c_new = list([idx for idx in permute_with_c if (idx != 2) and (
idx != 3)])
376
377     R1 = e1*e3
378
379     S1 = rotate(S_parity, R1)
380
381     for idx in permute_with_c_new+[4]: # The index of c is 4 in the list S
382
383         # Map x_12_rotated to c_12_original
384
385         x = S1[idx]
386         x_12 = x - x[3]*e3
387         c_12 = c - c[3]*e3
388         n = (x_12 + c_12).normal()
389         if n == 0: # If x_12 and c_12 are anti-parallel then we need a pi
rotation
390             R2 = e1*e2
391         else: # Otherwise we construct the rotor as usual with the form R = (
final destination)*n
392             R2 = c_12.normal()*n
393             S2 = rotate(S1, R2)
394             S3 = swap(S2, idx, 4)
395
396             if S == S3:
397                 flag = True
398
399     else: # Here we consider that a is collinear with e3 or is 0 but b is not
400
401     # We map x_12 to b_12_original and consider permutations between (cd), where
x_12 can be (b,c,d)_12
402
403     permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])
404
405     R1 = e1 * e3
406
407     S1 = rotate(S_parity, R1)
408
409     for idx in permute_with_b_new + [3]: # The index of b is 3 in the list S
410
411         # Map x_12_rotated to b_12_original
412
413         x = S1[idx]
414         x_12 = x - x[3] * e3
415         b_12 = b - b[3] * e3
416         n = (x_12 + b_12).normal()
417         if n == 0: # If x_12 and b_12 are anti-parallel then we need a pi rotation
418             R2 = e1 * e2
419         else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
420             R2 = b_12.normal() * n
421             S2 = rotate(S1, R2)
422             S3 = swap(S2, idx, 3)
423
424             if S == S3:
425                 flag = True
426

```

```

427         if permutation_boolean(M, E, 4, 5): # If (cd) is possible
428
429             S4 = swap(S3, 4, 5)
430
431             if S == S4:
432
433                 flag = True
434
435     else: # Here we consider that a is not collinear with e3 and not 0
436
437         # We need to map x_12 to a_12_original and consider permutations (bcd)
438
439         R1 = e1 * e3
440
441         S1 = rotate(S_parity, R1)
442
443         for idx in permute_with_a + [2]: # The index of a is 2 in the list S
444
445             # Map x_12_rotated to a_12_original
446
447             x = S1[idx]
448             x_12 = x - x[3] * e3
449             a_12 = a - a[3] * e3
450             n = (x_12 + a_12).normal()
451             if n == 0: # If x_12 and a_12 are anti-parallel then we need a pi rotation
452                 R2 = e1 * e2
453             else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
454                 R2 = a_12.normal() * n
455             S2 = rotate(S1, R2)
456             S3 = swap(S2, idx, 2)
457
458             if S == S3:
459                 flag = True
460
461             flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available
462
463             if flag_tmp_1:
464                 S4 = swap(S3, 3, 4)
465                 if S == S4:
466                     flag = True
467
468             flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available
469
470             if flag_tmp_2:
471                 S5 = swap(S3, 3, 5)
472                 if S == S5:
473                     flag = True
474
475             if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
476                 # The following achieves b->c->d->b
477                 S6 = swap(S3, 3, 4) # (bc)
478                 S6 = swap(S6, 3, 5) # (bd), here in the 3 index lies c but we name it b
still, notice we use S6
479                 if S == S6:
480                     flag = True
481                 # The following achieves b->d->c->b
482                 S7 = swap(S3, 3, 5) # (bd)
483                 S7 = swap(S7, 3, 4) # (bc), here in the 3 index lies d but we name it b
still, notice we use S7
484                 if S == S7:
485                     flag = True
486
487             if permutation_boolean(M, E, 4, 5): # If we have (cd)
488
489                 S8 = swap(S3, 4, 5)
490                 if S == S8:
491                     flag = True
492

```

```

493 else: # Now we consider the case where (pq) is available
494
495     # We can either try the above or omit using R1 and just use (pq)
496     # Whenever we check for non-chirality we can also perform a pi rotation in the plane
497     # that contains the
498     # 3-axis and has normal the final_state_particle_12 we matched in the 1-2 plane (usually
499     # this is a unless a has
500     # no 1,2 components)
501
502     # The following is repeating the above but incorporating the (pq) degree of freedom
503
504     if (a[1] == 0 and a[2] == 0) or (a == 0):
505         if (b[1] == 0 and b[2] == 0) or (b == 0):
506             if (c[1] == 0 and c[2] == 0) or (c == 0):
507                 flag = True
508
509             else: # Here we consider that a,b are either e3 collinear or 0, only c and d are
510             # in the 1-2 plane
511
512                 # We can map c_12 to its original or d_12 to the original c_12 and swap (cd)
513                 # if they can be
514                 # permuted.
515
516                 # Since a and b are fixed by R1 = e1e3 we only want to consider permutations
517                 # (cd)
518                 # Remember that idx 2 corresponds to a and idx 3 corresponds to b in S
519                 permute_with_c_new = list([idx for idx in permute_with_c if (idx != 2) and (
520                 idx != 3)])
521
522                 R1 = e1*e3
523
524                 S1 = rotate(S_parity, R1)
525
526                 for idx in permute_with_c_new+[4]: # The index of c is 4 in the list S
527
528                     # Map x_12_rotated to c_12_original
529
530                     x = S1[idx]
531                     x_12 = x - x[3]*e3
532                     c_12 = c - c[3]*e3
533                     n = (x_12 + c_12).normal()
534                     if n == 0: # If x_12 and c_12 are anti-parallel then we need a pi
535                     rotation
536                         R2 = e1*e2
537                     else: # Otherwise we construct the rotor as usual with the form R = (
538                     final destination)*n
539                         R2 = c_12.normal()*n
540                     S2 = rotate(S1, R2)
541                     S3 = swap(S2, idx, 4)
542
543                     if S == S3:
544                         flag = True
545
546                     # Now we can try the pi rotation in the plane v-e3 where v is
547                     # perpendicular c_12 and e3
548
549                     v = (-I*(e3^c_12)).normal() # Cross product in geometric algebra, I is
550                     # the pseudoscalar, v = e3 x c_12
551                     R3 = e3^v
552
553                     S4 = rotate(S3, R3)
554                     S5 = swap(S4, 0, 1)
555
556                     if S == S5:
557                         flag = True

```

```

552     else: # Here we consider that a is collinear with e3 or is 0 but b is not
553
554         # We map x_12 to b_12_original and consider permutations between (cd), where
555         x_12 can be (b,c,d)_12
556
557         permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])
558
559         R1 = e1 * e3
560
561         S1 = rotate(S_parity, R1)
562
563         for idx in permute_with_b_new + [3]: # The index of b is 3 in the list S
564
565             # Map x_12_rotated to b_12_original
566
567             x = S1[idx]
568             x_12 = x - x[3] * e3
569             b_12 = b - b[3] * e3
570             n = (x_12 + b_12).normal()
571             if n == 0: # If x_12 and b_12 are anti-parallel then we need a pi rotation
572                 R2 = e1 * e2
573             else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
574                 R2 = b_12.normal() * n
575                 S2 = rotate(S1, R2)
576                 S3 = swap(S2, idx, 3)
577
578                 if S == S3:
579                     flag = True
580
581                 v = (-I * (e3 ^ b_12)).normal() # Cross product in geometric algebra, I is
the pseudoscalar, v = e3 x c_12
582                 R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v
583
584                 S4 = rotate(S3, R3)
585                 S5 = swap(S4, 0, 1)
586
587                 if S == S5:
588                     flag = True
589
590                 if permutation_boolean(M, E, 4, 5): # If (cd) is possible
591
592                     S6 = swap(S3, 4, 5)
593
594                     if S == S6:
595
596                         flag = True
597
598                     S7 = rotate(S6, R3)
599                     S8 = swap(S7, 0, 1)
600
601                     if S == S8:
602                         flag = True
603
604     else: # Here we consider that a is not collinear with e3 and not 0
605
606         # We need to map x_12 to a_12_original and consider permutations (bcd)
607
608         R1 = e1 * e3
609
610         S1 = rotate(S_parity, R1)
611
612         for idx in permute_with_a + [2]: # The index of a is 2 in the list S
613
614             # Map x_12_rotated to a_12_original
615
616             x = S1[idx]
617             x_12 = x - x[3] * e3

```

```

618     a_12 = a - a[3] * e3
619     n = (x_12 + a_12).normal()
620     if n == 0: # If x_12 and a_12 are anti-parallel then we need a pi rotation
621         R2 = e1 * e2
622     else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
623         R2 = a_12.normal() * n
624     S2 = rotate(S1, R2)
625     S3 = swap(S2, idx, 2)
626
627     if S == S3:
628         flag = True
629
630     v = (-I * (e3 ^ a_12)).normal() # Cross product in geometric algebra, I is the
pseudoscalar, v = e3 x c_12
631     R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v
632
633     S4 = rotate(S3, R3)
634     S5 = swap(S4, 0, 1)
635
636     if S == S5:
637         flag = True
638
639     flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available
640
641     if flag_tmp_1:
642         S6 = swap(S3, 3, 4)
643         if S == S6:
644             flag = True
645
646         S7 = rotate(S6, R3)
647         S8 = swap(S7, 0, 1)
648
649         if S == S8:
650             flag = True
651
652     flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available
653
654     if flag_tmp_2:
655         S9 = swap(S3, 3, 5)
656         if S == S9:
657             flag = True
658         S10 = rotate(S9, R3)
659         S11 = swap(S10, 0, 1)
660
661         if S == S11:
662             flag = True
663
664
665
666     if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
667         # The following achieves b->c->d->b
668         S12 = swap(S3, 3, 4) # (bc)
669         S13 = swap(S12, 3, 5) # (bd), here in the 3 index lies c but we name it b
still, notice we use S6
670         if S == S13:
671             flag = True
672         S14 = rotate(S13, R3)
673         S15 = swap(S14, 0, 1)
674
675         if S == S15:
676             flag = True
677         # The following achieves b->d->c->b
678         S16 = swap(S3, 3, 5) # (bd)
679         S17 = swap(S16, 3, 4) # (bc), here in the 3 index lies d but we name it b
still, notice we use S7
680         if S == S17:
681             flag = True
682         S18 = rotate(S17, R3)

```



```

683         S19 = swap(S18, 0, 1)
684
685         if S == S19:
686             flag = True
687
688         if permutation_boolean(M, E, 4, 5): # If we have (cd)
689
690             S20 = swap(S3, 4, 5)
691             if S == S20:
692                 flag = True
693             S21 = rotate(S20, R3)
694             S22 = swap(S21, 0, 1)
695
696             if S == S22:
697                 flag = True
698
699
700     # The following is omitting R1 and just uses (pq), it again incorporates the (pq) degree
of freedom
701
702     if (a[1] == 0 and a[2] == 0) or (a == 0):
703
704         if (b[1] == 0 and b[2] == 0) or (b == 0):
705
706             if (c[1] == 0 and c[2] == 0) or (c == 0):
707
708                 flag = True
709
710             else: # Here we consider that a,b are either e3 collinear or 0, only c and d
are in the 1-2 plane
711
712                 # We can map c_12 to its original or d_12 to the original c_12 and swap (cd)
if they can be
713                 # permuted.
714
715                 # Since a and b are fixed by R1 = e1e3 we only want to consider permutations
(cd)
716                 # Remember that idx 2 corresponds to a and idx 3 corresponds to b in S
permute_with_c_new = list([idx for idx in permute_with_c if (idx != 2) and (
idx != 3)])
717
718                 S1 = swap(S_parity, 0, 1)
719
720                 for idx in permute_with_c_new + [4]: # The index of c is 4 in the list S
721
722                     # Map x_12_rotated to c_12_original
723
724                     x = S1[idx]
725                     x_12 = x - x[3] * e3
726                     c_12 = c - c[3] * e3
727                     n = (x_12 + c_12).normal()
728                     if n == 0: # If x_12 and c_12 are anti-parallel then we need a pi
rotation
729
730                         R2 = e1 * e2
731                     else: # Otherwise we construct the rotor as usual with the form R = (
final destination)*n
732                         R2 = c_12.normal() * n
733                     S2 = rotate(S1, R2)
734                     S3 = swap(S2, idx, 4)
735
736                     if S == S3:
737                         flag = True
738
739                     # Now we can try the pi rotation in the plane v-e3 where v is
perpendicular c_12 and e3
740
741                     v = (-I * (e3 ^ c_12)).normal() # Cross product in geometric algebra, I
is the pseudoscalar, v = e3 x c_12
742                     R3 = e3 ^ v

```

```

743         S4 = rotate(S3, R3)
744         S5 = swap(S4, 0, 1)
745
746         if S == S5:
747             flag = True
748
749     else: # Here we consider that a is collinear with e3 or is 0 but b is not
750
751         # We map x_12 to b_12_original and consider permutations between (cd), where
752         x_12 can be (b,c,d)_12
753
754         permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])
755
756         S1 = swap(S_parity, 0, 1)
757
758         for idx in permute_with_b_new + [3]: # The index of b is 3 in the list S
759
760             # Map x_12_rotated to b_12_original
761
762             x = S1[idx]
763             x_12 = x - x[3] * e3
764             b_12 = b - b[3] * e3
765             n = (x_12 + b_12).normal()
766             if n == 0: # If x_12 and b_12 are anti-parallel then we need a pi rotation
767                 R2 = e1 * e2
768             else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
769                 R2 = b_12.normal() * n
770             S2 = rotate(S1, R2)
771             S3 = swap(S2, idx, 3)
772
773             if S == S3:
774                 flag = True
775
776             v = (-I * (e3 ^ b_12)).normal() # Cross product in geometric algebra, I is
the pseudoscalar, v = e3 x c_12
777             R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v
778
779             S4 = rotate(S3, R3)
780             S5 = swap(S4, 0, 1)
781
782             if S == S5:
783                 flag = True
784
785             if permutation_boolean(M, E, 4, 5): # If (cd) is possible
786
787                 S6 = swap(S3, 4, 5)
788
789                 if S == S6:
790                     flag = True
791
792                 S7 = rotate(S6, R3)
793                 S8 = swap(S7, 0, 1)
794
795                 if S == S8:
796                     flag = True
797
798     else: # Here we consider that a is not collinear with e3 and not 0
799
800         # We need to map x_12 to a_12_original and consider permutations (bcd)
801
802         S1 = swap(S_parity, 0, 1)
803
804         for idx in permute_with_a + [2]: # The index of a is 2 in the list S
805
806             # Map x_12_rotated to a_12_original
807
808             x = S1[idx]

```

```

809     x_12 = x - x[3] * e3
810     a_12 = a - a[3] * e3
811     n = (x_12 + a_12).normal()
812     if n == 0: # If x_12 and a_12 are anti-parallel then we need a pi rotation
813         R2 = e1 * e2
814     else: # Otherwise we construct the rotor as usual with the form R = (final
destination)*n
815         R2 = a_12.normal() * n
816     S2 = rotate(S1, R2)
817     S3 = swap(S2, idx, 2)
818
819     if S == S3:
820         flag = True
821
822     v = (-I * (e3 ^ a_12)).normal() # Cross product in geometric algebra, I is the
pseudoscalar, v = e3 x c_12
823     R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v
824
825     S4 = rotate(S3, R3)
826     S5 = swap(S4, 0, 1)
827
828     if S == S5:
829         flag = True
830
831     flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available
832
833     if flag_tmp_1:
834         S6 = swap(S3, 3, 4)
835         if S == S6:
836             flag = True
837
838         S7 = rotate(S6, R3)
839         S8 = swap(S7, 0, 1)
840
841         if S == S8:
842             flag = True
843
844     flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available
845
846     if flag_tmp_2:
847         S9 = swap(S3, 3, 5)
848         if S == S9:
849             flag = True
850         S10 = rotate(S9, R3)
851         S11 = swap(S10, 0, 1)
852
853         if S == S11:
854             flag = True
855
856     if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
857         # The following achieves b->c->d->b
858         S12 = swap(S3, 3, 4) # (bc)
859         S13 = swap(S12, 3, 5) # (bd), here in the 3 index lies c but we name it b
still, notice we use S6
860         if S == S13:
861             flag = True
862         S14 = rotate(S13, R3)
863         S15 = swap(S14, 0, 1)
864
865         if S == S15:
866             flag = True
867         # The following achieves b->d->c->b
868         S16 = swap(S3, 3, 5) # (bd)
869         S17 = swap(S16, 3, 4) # (bc), here in the 3 index lies d but we name it b
still, notice we use S7
870         if S == S17:
871             flag = True
872         S18 = rotate(S17, R3)
873         S19 = swap(S18, 0, 1)

```

```

874
875         if S == S19:
876             flag = True
877
878         if permutation_boolean(M, E, 4, 5): # If we have (cd)
879
880             S20 = swap(S3, 4, 5)
881             if S == S20:
882                 flag = True
883             S21 = rotate(S20, R3)
884             S22 = swap(S21, 0, 1)
885
886             if S == S22:
887                 flag = True
888
889     if flag:
890         non_chiral_states.append([S, E])
891     else:
892         chiral_states.append([S, E])
893
894     return non_chiral_states, chiral_states
895
896 # chiral_states = []
897 # non_chiral_states = []
898 #
899 # for _ in range(10):
900 #
901 #     non_chiral_states_tmp, chiral_states_tmp = chirality_test()
902 #     non_chiral_states += non_chiral_states_tmp
903 #     chiral_states += chiral_states_tmp
904 #
905 # # The first slot is incremented for every true and the second for every false
906 # non_chiral_evaluation_on_logic_statement = [0, 0]
907 # chiral_evaluation_on_logic_statement = [0, 0]
908 #
909 # for non_chiral_state in non_chiral_states:
910 #     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
911 #     if flag:
912 #         non_chiral_evaluation_on_logic_statement[0] += 1
913 #     else:
914 #         non_chiral_evaluation_on_logic_statement[1] += 1
915 #
916 # for chiral_state in chiral_states:
917 #     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
918 #     if flag:
919 #         chiral_evaluation_on_logic_statement[0] += 1
920 #     else:
921 #         chiral_evaluation_on_logic_statement[1] += 1
922 #
923 # x = ['True', 'False']
924 # height_non_chiral = [non_chiral_evaluation_on_logic_statement[0],
925 #                       non_chiral_evaluation_on_logic_statement[1]]
926 # height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement
927 #                  [1]]
928 #
929 # plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
930 # plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is
931 #           non-chiral')
932 # plt.ylabel('Frequency')
933 # plt.show()
934 # # plt.savefig('non_chiral_collision_logic_statement_test.pdf', bbox_inches='tight')
935 # #
936 # plt.bar(x, height_chiral, color = 'k', width = 0.1)
937 # plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-
938 #           chiral')
939 # plt.ylabel('Frequency')
940 # plt.show()
941 # plt.savefig('chiral_collision_logic_statement_test.pdf', bbox_inches='tight')

```

Code file: non_collision_one_massive_algorithm.py

```

1 import time
2 from numpy import pi, cos, sin, e, tan, arctan
3 from clifford.g3 import blades
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 import numpy as np
7 from random import uniform, seed, randint
8 from sympy import LeviCivita as eps
9 from main import parity, rotate, energy, epsilon, Gram_det_2
10 from pytest import approx
11
12 # Works only for a,b,c,d != 0 (it is very unlikely that any one will be randomly generated in
13   momentum 0)
14 e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
15 I = e1^e2^e3
16
17 def dot(a,b):
18     dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
19     return dot
20
21 def multivec_to_vec(a):
22     return np.array([a[1], a[2], a[3]])
23
24 def swap(S,idx_1,idx_2):
25
26     tmp = S[idx_1]
27     S[idx_1] = S[idx_2]
28     S[idx_2] = tmp
29
30     return S
31
32 def logic_statement_true_for_non_chiral(S, E, mp, mq):
33
34     p = np.array([mp, 0, 0, 0])
35     q = np.array([mq, 0, 0, 0])
36     a = multivec_to_vec(S[0])
37     a = np.insert(a, 0, E[0])
38     b = multivec_to_vec(S[1])
39     b = np.insert(b, 0, E[1])
40     c = multivec_to_vec(S[2])
41     c = np.insert(c, 0, E[2])
42     d = multivec_to_vec(S[3])
43     d = np.insert(d, 0, E[3])
44     RF = p + q
45
46     case_1 = ((epsilon(a, b, c, RF) == approx(0)) and (epsilon(a, b, d, RF) == approx(0)) and (
47         epsilon(b, c, d, RF) == approx(0)) and (
48             epsilon(a, c, d, RF) == approx(0)))
49
50     case_2 = ((dot(a,a) == approx(dot(b,b))) and (not (a == b).all()) and (Gram_det_2(a,RF,a,RF)
51         == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(a,RF,c,RF) == approx((Gram_det_2(b,RF,c,
52         RF)))) and (Gram_det_2(a,RF,d,RF) == approx(Gram_det_2(b,RF,d,RF))))
53
54     def case_2_symmetry(a,b,c,d):
55         return ((dot(a,a) == approx(dot(b,b))) and (not (a == b).all()) and (Gram_det_2(a,RF,a,
56         RF) == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(a,RF,c,RF) == approx((Gram_det_2(b,RF,
57         c,RF) == 0))) and (Gram_det_2(a,RF,d,RF) == approx(Gram_det_2(b,RF,d,RF))))
58
59     case_3 = case_2_symmetry(a,c,b,d)
60     case_4 = case_2_symmetry(a,d,c,b)
61     case_5 = case_2_symmetry(b,c,a,d)
62     case_6 = case_2_symmetry(b,d,c,a)
63     case_7 = case_2_symmetry(c,d,a,b)

```

```

60 case_8 = ((dot(a,a) == approx(dot(b,b))) and (dot(c,c) == approx(dot(d,d))) and (Gram_det_2(
a,RF,a,RF) == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(c,RF,c,RF) == approx(
Gram_det_2(d,RF,d,RF))) and (((dot(a+b,a+b) == approx(4*dot(a,a)) == approx(4*dot(b,b)))
and ((dot(c+d,c+d) == approx(4*dot(c,c)) == approx(4*dot(d,d)))) or ((Gram_det_2(a+b,RF,c-
d,RF) == approx(0)) or (not (c == d).all()))))
61
62 def case_8_symmetry(a,b,c,d):
63     return ((dot(a,a) == approx(dot(b,b))) and (dot(c,c) == approx(dot(d,d))) and (
Gram_det_2(a,RF,a,RF) == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(c,RF,c,RF) ==
approx(Gram_det_2(d,RF,d,RF))) and (((dot(a+b,a+b) == approx(4*dot(a,a)) == approx(4*dot(b
,b))) and ((dot(c+d,c+d) == approx(4*dot(c,c)) == approx(4*dot(d,d)))) or ((Gram_det_2(a+
b,RF,c-d,RF) == approx(0)) or (not (c == d).all()))))
64
65 case_9 = case_8_symmetry(a,c,b,d)
66 case_10 = case_8_symmetry(a,d,b,c)
67
68 case_19 = ((dot(a,a) == approx(dot(b,b)) == approx(dot(c,c)) == approx(dot(d,d))) and (
Gram_det_2(a,RF,a,RF) == approx(Gram_det_2(b,RF,b,RF)) == approx(Gram_det_2(c,RF,c,RF)) ==
approx(Gram_det_2(d,RF,d,RF))) and approx((Gram_det_2(a - c,RF,b - d,RF) == 0))
69
70 def case_19_symmetry(a,b,c,d):
71     return ((dot(a,a) == approx(dot(b,b)) == approx(dot(c,c)) == approx(dot(d,d))) and (
Gram_det_2(a,RF,a,RF) == approx(Gram_det_2(b,RF,b,RF)) == approx(Gram_det_2(c,RF,c,RF)) ==
approx(Gram_det_2(d,RF,d,RF))) and approx((Gram_det_2(a - c,RF,b - d,RF) == 0))
72
73 case_20 = case_19_symmetry(a,b,d,c)
74 case_21 = case_19_symmetry(a,c,b,d)
75 case_22 = case_19_symmetry(a,c,d,b)
76 case_23 = case_19_symmetry(a,d,c,b)
77 case_24 = case_19_symmetry(a,d,b,c)
78
79 return (case_1 or case_2 or case_3 or case_4 or case_5 or case_6 or case_7 or case_8 or
case_9 or case_10
80         or case_19 or case_20 or case_21 or case_22 or case_23 or case_24)
81
82 def construct_state():
83
84     B = (uniform(-10, 10) * (e1 ^ e2) + uniform(-10, 10) * (e1 ^ e3) + uniform(-10, 10) * (e2 ^
e3)).normal()
85     R = (e ^ (uniform(0, 2 * pi) * B)).normal()
86
87     rdm = randint(0, 4)
88
89     if rdm == 0:
90         ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
91         a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
92         b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
93         c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
94         d = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
95
96     # Type of non-chiral (checked) (ab)
97     if rdm == 1:
98         ma, mb, mc, md = 1, 1, 2, 3
99         a = e1
100        b = e2
101        c = e3
102        d = (-e3)
103
104     # Type of non-chiral (checked) (ab)(cd)
105     if rdm == 2:
106         ma, mb, mc, md = 1, 1, 2, 2
107         n = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
108         n = n.normal()
109         a = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
110         b = -n*a*n
111         c = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
112         d = -n*c*n
113
114     # Type of non-chiral (checked) (abc)

```

```

115     if rdm == 3:
116         ma, mb, mc, md = 1, 1, 1, 3
117         mag = uniform(-10, 10)
118         a = mag*e1 + mag*e2
119         b = mag*e1 - mag*e2
120         c = -mag*e1 + mag*e2
121         d = mag*e1 - mag*e2
122
123     # Type of non-chiral (checked) (abcd)
124     if rdm == 4:
125         ma, mb, mc, md = 1, 1, 1, 1
126         a = R*(-e1 + e3)*~R
127         b = R*(e1 + e2)*~R
128         c = R*(-e1 - e3)*~R
129         d = R*(e1 - e2)*~R
130
131     Ea, Eb, Ec, Ed = energy(ma, a), energy(mb, b), energy(mc, c), energy(md, d)
132
133     M = [ma, mb, mc, md]
134     E = [Ea, Eb, Ec, Ed]
135     S = [a, b, c, d]
136
137     print(rdm)
138
139     return S, E, M
140
141 def permute_with_idx(M, E, idx_to_permute):
142     same_mass_with_idx = [idx for idx in range(len(M)) if M[idx] == M[idx_to_permute] and idx !=
143                          idx_to_permute]
144     same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute])
145                            and idx != idx_to_permute]
146
147     return list(set(same_mass_with_idx) and set(same_energy_with_idx))
148
149 def permutation_boolean(M, E, idx_1, idx_2):
150     if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
151         return True
152     else:
153         return False
154
155 def chirality_test():
156     chiral_states = []
157     non_chiral_states = []
158
159     S, E, M = construct_state()
160     a, b, c, d = S[0], S[1], S[2], S[3] # p and q are 0 so we do not carry them around in the S
161     list
162
163     # These lists hold indices (as they appear in S) of the particles that can be permuted with
164     # a,b,c and d respectively
165     permute_with_a = permute_with_idx(M, E, 0)
166     permute_with_b = permute_with_idx(M, E, 1)
167     permute_with_c = permute_with_idx(M, E, 2)
168     permute_with_d = permute_with_idx(M, E, 3)
169
170     S_parity = parity(S) # Perform parity on the set of momenta
171
172     flag = False # The flag is set to true if the state is non-chiral
173
174     for idx in permute_with_a+[0]: # For every x that can be permuted with a, map x to a and
175     perform (ax)
176
177         x = S_parity[idx]
178         n = a+x
179         if n == 0: # If a and x are collinear we construct any v perpendicular to a and do a pi
180         rotation in the plane av

```

```

178         if a[2] != 0 or a[3] != 0:
179             v = -I*(a^e1) # Cross product a x e1 in geometric algebra, I is the pseudoscalar
I = e1e2e3
180         else:
181             v = -I*(a^e2)
182             R1 = (a^v).normal()
183         else:
184             R1 = a.normal()*n.normal()
185
186         S1 = rotate(S_parity, R1)
187         S2 = swap(S1, 0, idx) # Performs (ax) where the index 0 corresponds to a in the S list
188
189         # At this point a is fixed to its original state
190
191         # Now we need to fix b in the plane perpendicular to a, if b has no component (a^b==0)
there then we try c
192
193         if a^b != 0: # If b is not collinear to a then it has components in the plane
perpendicular to a
194
195             for idx_plane in permute_with_b + [1]:
196
197                 if idx_plane == 0: # Since a is already fixed
198                     continue
199
200                 # If we have a plane P with its perpendicular being a and we have a vector y,
then the component of y
201                 # in the plane P is given in geometric algebra by the rejection y_plane = a*(a^y
)
202
203                 b_plane = a*(a^b)
204                 y = S2[idx_plane]
205                 y_plane = a*(a^y)
206
207                 # If y is collinear with a then it has no component in the plane perpendicular
to a, so even if it can
208                 # be permuted with b, we cannot map y_plane to b_plane and then swap because
y_plane is 0
209
210                 if y_plane == 0:
211                     continue
212                 else: # Here we map y_plane to b_plane and perform (yb)
213                     m = y_plane + b_plane
214                     if m == 0: # In this case we need a pi rotation in this plane we are working
in
215
216                     # We construct another vector in this plane with the cross product a x
b_plane
217                     w_plane = -I*(a^b_plane)
218                     R2 = (w_plane^b_plane).normal()
219                 else:
220                     R2 = b_plane.normal()*m.normal()
221
222                 S3 = rotate(S2, R2) # Map y_plane to b_plane
223                 S4 = swap(S3, 1, idx_plane) # Perform (yb)
224
225                 if S == S4:
226                     flag = True
227
228                 if permutation_boolean(M, E, 2, 3): # If we can perform (cd)
229                     S5 = swap(S4, 2, 3)
230                     if S == S5:
231                         flag = True
232
233             elif a^c != 0:
234
235                 for idx_plane in permute_with_c + [2]:
236
237                     # Since a and b are already fixed, b is fixed because we fixed a and to get into

```



```

238     this elif we need
239         # b to be collinear with a and hence when we fixed a we automatically fixed b
240         if idx_plane == 0 or idx_plane == 1:
241             continue
242
243         # If we have a plane P with its perpendicular being a and we have a vector y,
244         # then the component of y
245         # in the plane P is given in geometric algebra by the rejection  $y_{\text{plane}} = a \wedge y$ 
246         )
247
248         c_plane = a * (a ^ c)
249         y = S2[idx_plane]
250         y_plane = a * (a ^ y)
251
252         # If y is collinear with a then it has no component in the plane perpendicular
253         # to a, so even if it can
254         # be permuted with c, we cannot map y_plane to c_plane and then swap because
255         # y_plane is 0
256
257         if y_plane == 0:
258             continue
259         else: # Here we map y_plane to c_plane and perform (yc)
260             m = y_plane + c_plane
261             if m == 0: # In this case we need a pi rotation in this plane we are
262                 working in
263                     # We construct another vector in this plane with the cross product a x
264                     c_plane
265
266                     w_plane = -I * (a ^ c_plane)
267                     R2 = (w_plane ^ c_plane).normal()
268                 else:
269                     R2 = c_plane.normal() * m.normal()
270
271             S6 = rotate(S2, R2) # Map y_plane to c_plane
272             S7 = swap(S6, 2, idx_plane) # Perform (yc)
273
274             if S == S7:
275                 flag = True
276
277             if permutation_boolean(M, E, 1, 3): # If we can perform (bd)
278                 S8 = swap(S7, 1, 3)
279                 if S == S8:
280                     flag = True
281
282             else:
283                 flag = True # If a,b,c are collinear then the state is non-chiral
284
285         if flag:
286             non_chiral_states.append([S, E])
287         else:
288             chiral_states.append([S, E])
289
290     return non_chiral_states, chiral_states
291
292 # non_chiral_states_list = []
293 # chiral_states_list = []
294 # for iterations in range(1000):
295 #     S, E, M = construct_state()
296 #     non_chiral_states, chiral_states = chirality_test()
297 #     non_chiral_states_list += non_chiral_states
298 #     chiral_states_list += chiral_states
299 #
300 # print(len(non_chiral_states_list), len(chiral_states_list))
301 #
302 # non_chiral_evaluation_on_logic_statement = [0, 0]
303 # for non_chiral_state in non_chiral_states_list:
304 #     mp, mq = uniform(1, 10), uniform(1, 10)
305 #     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1], mp,
306 #     mq)
307 #     if flag:

```

```

299 #         non_chiral_evaluation_on_logic_statement[0] += 1
300 #     else:
301 #         non_chiral_evaluation_on_logic_statement[1] += 1
302 #
303 # chiral_evaluation_on_logic_statement = [0, 0]
304 # for chiral_state in chiral_states_list:
305 #     mp, mq = uniform(1, 10), uniform(1, 10)
306 #     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1], mp, mq)
307 #     if flag:
308 #         chiral_evaluation_on_logic_statement[0] += 1
309 #     else:
310 #         chiral_evaluation_on_logic_statement[1] += 1
311 #
312 # x = ['True', 'False']
313 # height_non_chiral = [non_chiral_evaluation_on_logic_statement[0],
314 #                     non_chiral_evaluation_on_logic_statement[1]]
315 # height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement
316 #                 [1]]
317 #
318 # plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
319 # plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is
320 #           non-chiral')
321 # plt.ylabel('Frequency')
322 # plt.show()
323 # plt.savefig('non_chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
324 #
325 # plt.bar(x, height_chiral, color = 'k', width = 0.1)
326 # plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-
327 #           chiral')
328 # plt.ylabel('Frequency')
329 # plt.show()
330 # plt.savefig('chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')

```

Code file: non_collision_massless_algorithm.py

```

1 import time
2 from numpy import pi, cos, sin, e, tan, arctan
3 from clifford.g3 import blades
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 import numpy as np
7 from random import uniform, seed, randint
8 from sympy import LeviCivita as eps
9 from main import parity, rotate, energy, epsilon, Gram_det_2
10 from pytest import approx
11
12 e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
13 I = e1^e2^e3
14
15 def dot(a,b):
16     dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
17     return dot
18
19 def multivec_to_vec(a):
20     return np.array([a[1], a[2], a[3]])
21
22 def swap(S,idx_1,idx_2):
23
24     tmp = S[idx_1]
25     S[idx_1] = S[idx_2]
26     S[idx_2] = tmp
27
28     return S
29
30 def sym_2_Gram_det(a,b):
31     return Gram_det_2(a,b,a,b)
32
33 def logic_statement_true_for_non_chiral(S, E):
34

```

```

35 p = multivec_to_vec(S[0])
36 p = np.insert(p, 0, E[0])
37 q = multivec_to_vec(S[1])
38 q = np.insert(q, 0, E[1])
39 a = multivec_to_vec(S[2])
40 a = np.insert(a, 0, E[2])
41 b = multivec_to_vec(S[3])
42 b = np.insert(b, 0, E[3])
43 c = multivec_to_vec(S[4])
44 c = np.insert(c, 0, E[4])
45 d = multivec_to_vec(S[5])
46 d = np.insert(d, 0, E[5])
47 RF = a + b + c + d
48
49 case_1 = (epsilon(a,b,p+q,RF) == epsilon(a,c,p+q,RF) == epsilon(a,d,p+q,RF) == epsilon(b,c,p
+q,RF) == epsilon(b,d,p+q,RF) == epsilon(c,d,p+q,RF) == 0)
50
51 case_2 = ((dot(a,a) == approx(dot(b,b))) and (Gram_det_2(a-b,RF,p+q,RF) == approx(0)) and (
sym_2_Gram_det(a,RF) == approx(sym_2_Gram_det(b,RF))))
52
53 def case_2_symmetry(a,b,c,d):
54     return ((dot(a,a) == dot(b,b)) and (Gram_det_2(a-b,RF,p+q,RF) == 0) and (sym_2_Gram_det(
a,RF) == sym_2_Gram_det(b,RF)))
55
56 case_3 = case_2_symmetry(a, c, b, d)
57 case_4 = case_2_symmetry(a, d, c, b)
58 case_5 = case_2_symmetry(b, c, a, d)
59 case_6 = case_2_symmetry(b, d, a, c)
60 case_7 = case_2_symmetry(c, d, a, b)
61
62 case_8 = ((dot(a,a) == approx(dot(b,b))) and (Gram_det_2(a-b,RF,p+q,RF) == approx(0)) and (
sym_2_Gram_det(a,RF) == approx(sym_2_Gram_det(b,RF))) and (dot(c,c) == approx(dot(d,d)))
and (Gram_det_2(c-d,RF,p+q,RF) == approx(0)) and (sym_2_Gram_det(c,RF) == approx(
sym_2_Gram_det(d,RF))) and (Gram_det_2(a-b,RF,c+d,RF) == approx(0)))
63
64 def case_8_symmetry(a,b,c,d):
65     return ((dot(a,a) == approx(dot(b,b))) and (Gram_det_2(a-b,RF,p+q,RF) == approx(0)) and
(sym_2_Gram_det(a,RF) == approx(sym_2_Gram_det(b,RF))) and (dot(c,c) == approx(dot(d,d)))
and (Gram_det_2(c-d,RF,p+q,RF) == approx(0)) and (sym_2_Gram_det(c,RF) == approx(
sym_2_Gram_det(d,RF))) and (Gram_det_2(a-b,RF,c+d,RF) == approx(0)))
66
67 case_9 = case_8_symmetry(a,c,b,d)
68 case_10 = case_8_symmetry(a,d,b,c)
69
70 return (case_1 or case_2 or case_3 or case_4 or case_5 or case_6 or case_7 or case_8 or
case_9 or case_10)
71
72 def permute_with_idx(M, E, idx_to_permute):
73
74     same_mass_with_idx = [idx for idx in range(len(M)) if M[idx] == M[idx_to_permute] and idx !=
idx_to_permute and idx != 0 and idx != 1]
75     same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute])
and idx != idx_to_permute]
76
77     return list(set(same_mass_with_idx) and set(same_energy_with_idx))
78
79 def permutation_boolean(M, E, idx_1, idx_2):
80
81     if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
82         return True
83     else:
84         return False
85
86 def construct_state():
87
88     rdm = randint(0, 0)
89
90     if rdm == 0:
91         mp, mq, ma, mb, mc, md = 0, 0, randint(0, 10), randint(0, 10), randint(0, 10), randint

```

```

92 (0, 10)
93 p = uniform(-10, 10)*e3
94 q = uniform(-10, 10)*e3
95 a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
96 b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
97 c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
98 d = - a - b - c
99
100 if rdm == 1:
101     # non chiral
102     mp, mq, ma, mb, mc, md = 0, 0, randint(0, 10), randint(0, 10), randint(0, 10), randint
103     (0, 10)
104     p = uniform(-10, 10)*e3
105     q = uniform(-10, 10)*e3
106     a = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
107     b = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
108     c = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
109     d = - a - b - c
110
111 if rdm == 2:
112     # (ab) non chiral
113     mp, mq, ma, mb, mc, md = 0, 0, 1, 1, randint(0, 10), randint(0, 10)
114     p = uniform(-10, 10)*e3
115     q = uniform(-10, 10)*e3
116     a = 4 * e1 + 5 * e2 + 3 * e3
117     B = (uniform(-10, 10) * (e1 ^ e2) + uniform(-10, 10) * (e1 ^ e3) + uniform(-10, 10) * (
118     e2 ^ e3)).normal()
119     R = e ^ (uniform(0, 2 * pi) * B)
120     b = -5 * e1 + 4 * e2 + 3 * e3
121     c = a+b
122     d = -c
123
124 if rdm == 3:
125     # (ab)(cd) non chiral
126     mp, mq, ma, mb, mc, md = 0, 0, 1, 1, 2, 2
127     p = uniform(-10, 10)*e3
128     q = uniform(-10, 10)*e3
129     angle_a = uniform(0, pi/5)
130     angle_c = uniform(0, pi / 5)
131     a = cos(angle_a)*e1 + sin(angle_a)*e2 + uniform(-10, 10)*e3
132     b = sin(angle_a)*e1 + cos(angle_a)*e2 + a[3]*e3
133     c = -cos(angle_c)*e1 - sin(angle_c)*e2 - uniform(-10, 10)*e3
134     d = -sin(angle_c)*e1 - cos(angle_c)*e2 + c[3]*e3
135
136 Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(
137 mc, c), energy(md, d)
138
139 M = [mp, mq, ma, mb, mc, md]
140 E = [Ep, Eq, Ea, Eb, Ec, Ed]
141 S = [p, q, a, b, c, d]
142
143 return S, E, M
144
145 def chirality_test():
146     chiral_states = []
147     non_chiral_states = []
148
149     S, E, M = construct_state()
150     p, q, a, b, c, d = S[0], S[1], S[2], S[3], S[4], S[5]
151
152     # These lists hold indices (as they appear in S) of the particles that can be permuted with
153     # a,b,c and d respectively
154     permute_with_a = permute_with_idx(M, E, 2)
155     permute_with_b = permute_with_idx(M, E, 3)
156     permute_with_c = permute_with_idx(M, E, 4)
157     permute_with_d = permute_with_idx(M, E, 5)
158
159     S_parity = parity(S) # Perform parity on the set of momenta

```

```

156 flag = False # The flag is set to true if the state is non-chiral
157
158
159 R1 = e1*e3
160 S1 = rotate(S_parity, R1) # Now p and q are fixed back to their original state
161
162 if (a[1] != 0) or (a[2] != 0): # If a has components in the 1-2 plane
163
164     for idx in permute_with_a + [2]: # For every x that can be permuted with a, map x to a
165         and perform (ax)
166
167         x = S1[idx]
168         x_12 = x - x[3]*e3
169         a_12 = a - a[3]*e3
170
171         n = a_12 + x_12
172
173         if n == 0:
174             R2 = e1*e2
175
176         else:
177             R2 = a_12.normal()*n.normal()
178
179         S2 = rotate(S1, R2)
180         S3 = swap(S2, 2, idx)
181
182         if S == S3:
183             flag = True
184
185     # The degrees of freedom left now that a is fixed to its original state are
186     # permutations between b,c,d
187
188     flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available
189
190     if flag_tmp_1:
191         S4 = swap(S3, 3, 4)
192         if S == S4:
193             flag = True
194
195     flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available
196
197     if flag_tmp_2:
198         S5 = swap(S3, 3, 5)
199         if S == S5:
200             flag = True
201
202     flag_tmp_3 = permutation_boolean(M, E, 4, 5) # If we have (cd)
203
204     if flag_tmp_3:
205         S6 = swap(S3, 4, 5)
206         if S == S6:
207             flag = True
208
209     if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
210         # The following achieves b->c->d->b
211         S7 = swap(S3, 3, 4) # (bc)
212         S8 = swap(S7, 3, 5) # (bd), here in the 3 index lies c but we name it b still,
213         notice we use S6
214         if S == S8:
215             flag = True
216
217         # The following achieves b->d->c->b
218         S9 = swap(S3, 3, 5) # (bd)
219         S10 = swap(S9, 3, 4) # (bc), here in the 3 index lies d but we name it b still,
220         notice we use S7
221         if S == S10:
222             flag = True

```

```

221
222 elif (b[1] != 0) or (b[2] != 0): # To get here we asserted that a is collinear with e3 so
    fixed by R1
223
224     for idx in permute_with_b + [3]: # For every x that can be permuted with b, map x to b
    and perform (bx)
225
226         if idx == 2: # We do not want permutations with a since a is fixed by R1 when
    collinear with e3
227             continue
228
229             x = S1[idx]
230             x_12 = x - x[3] * e3
231             b_12 = b - b[3] * e3
232
233             n = b_12 + x_12
234
235             if n == 0:
236
237                 R2 = e1 * e2
238
239             else:
240
241                 R2 = b_12.normal() * n.normal()
242
243             S2 = rotate(S1, R2)
244             S3 = swap(S2, 3, idx)
245
246             if S == S3:
247                 flag = True
248
249             # The degrees of freedom left now that b is fixed to its original state are
    permutations between c,d
250
251             flag_tmp_1 = permutation_boolean(M, E, 4, 5) # This checks if (cd) is available
252
253             if flag_tmp_1:
254                 S4 = swap(S3, 4, 5)
255                 if S == S4:
256                     flag = True
257
258 elif (c[1] != 0) or (c[2] != 0): # To get here we asserted that a,b are collinear with e3 so
    fixed by R1
259
260     for idx in permute_with_c + [4]: # For every x that can be permuted with c, map x to c
    and perform (cx)
261
262         if (idx == 2) or (idx == 3): # We do not want permutations with a,b since a,b are
    fixed by R1
263             continue
264
265             x = S1[idx]
266             x_12 = x - x[3] * e3
267             c_12 = c - c[3] * e3
268
269             n = c_12 + x_12
270
271             if n == 0:
272
273                 R2 = e1 * e2
274
275             else:
276
277                 R2 = c_12.normal() * n.normal()
278
279             S2 = rotate(S1, R2)
280             S3 = swap(S2, 4, idx)
281
282             if S == S3:

```

```

283         flag = True
284
285     if flag:
286         non_chiral_states.append([S, E])
287     else:
288         chiral_states.append([S, E])
289
290     return non_chiral_states, chiral_states
291
292 # non_chiral_states_list = []
293 # chiral_states_list = []
294 # for iterations in range(1000):
295 #     S, E, M = construct_state()
296 #     non_chiral_states, chiral_states = chirality_test()
297 #     non_chiral_states_list += non_chiral_states
298 #     chiral_states_list += chiral_states
299 #
300 # print(len(non_chiral_states_list), len(chiral_states_list))
301 #
302 # non_chiral_evaluation_on_logic_statement = [0, 0]
303 # for non_chiral_state in non_chiral_states_list:
304 #     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
305 #     if flag:
306 #         non_chiral_evaluation_on_logic_statement[0] += 1
307 #     else:
308 #         non_chiral_evaluation_on_logic_statement[1] += 1
309 #
310 # chiral_evaluation_on_logic_statement = [0, 0]
311 # for chiral_state in chiral_states_list:
312 #     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
313 #     if flag:
314 #         chiral_evaluation_on_logic_statement[0] += 1
315 #     else:
316 #         chiral_evaluation_on_logic_statement[1] += 1
317 #
318 # x = ['True', 'False']
319 # height_non_chiral = [non_chiral_evaluation_on_logic_statement[0],
320 #                     non_chiral_evaluation_on_logic_statement[1]]
321 # height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement
322 #                 [1]]
323 #
324 # plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
325 # plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is
326 #           non-chiral')
327 # plt.ylabel('Frequency')
328 # plt.show()
329 # plt.savefig('non_chiral_non_collision_massless_logic_statement_test.pdf', bbox_inches='tight')
330 #
331 # plt.bar(x, height_chiral, color = 'k', width = 0.1)
332 # plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-
333 #           chiral')
334 # plt.ylabel('Frequency')
335 # plt.show()
336 # plt.savefig('chiral_non_collision_massless_logic_statement_test.pdf', bbox_inches='tight')
337 #
338 # print(chirality_test())

```

Code file: non_collision_zeros_algorithm.py

```

1 import time
2 from numpy import pi, cos, sin, e, tan, arctan
3 from clifford.g3 import blades
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 import numpy as np
7 from random import uniform, seed, randint
8 from sympy import LeviCivita as eps
9 from main import parity, rotate, energy, epsilon, Gram_det_2
10 from pytest import approx

```

```

11
12 # Works only for a,b,c,d != 0 (it is very unlikely that any one will be randomly generated in
    momentum 0)
13
14 e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
15 I = e1^e2^e3
16
17 def dot(a,b):
18     dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
19     return dot
20
21 def multivec_to_vec(a):
22     return np.array([a[1], a[2], a[3]])
23
24 def swap(S,idx_1,idx_2):
25
26     tmp = S[idx_1]
27     S[idx_1] = S[idx_2]
28     S[idx_2] = tmp
29
30     return S
31
32 def sym_2_Gram_det(a,b):
33     return Gram_det_2(a,b,a,b)
34
35 def logic_statement_true_for_non_chiral(S, E):
36
37     a = multivec_to_vec(S[0])
38     a = np.insert(a, 0, E[0])
39     b = multivec_to_vec(S[1])
40     b = np.insert(b, 0, E[1])
41     c = multivec_to_vec(S[2])
42     c = np.insert(c, 0, E[2])
43     d = multivec_to_vec(S[3])
44     d = np.insert(d, 0, E[3])
45     RF = a+b+c+d
46
47     case_1 = (Gram_det_2(a,RF,a,RF) == approx(0))
48
49     case_2 = (dot(a+b+c+d, a+b+c+d) == approx(0))
50
51     case_3 = (epsilon(b,c,d,RF) == approx(0))
52
53     case_4 = ((dot(b,b) == dot(c,c)) and (Gram_det_2(b-c,RF,a,RF) == 0) and (sym_2_Gram_det(b,RF)
    ) == sym_2_Gram_det(c,RF)))
54
55     case_5 = ((dot(c,c) == dot(d,d)) and (Gram_det_2(c-d,RF,a,RF) == 0) and (sym_2_Gram_det(c,RF)
    ) == sym_2_Gram_det(d,RF)))
56
57     case_6 = ((dot(d,d) == dot(b,b)) and (Gram_det_2(d-b,RF,a,RF) == 0) and (sym_2_Gram_det(d,RF)
    ) == sym_2_Gram_det(b,RF)))
58
59     return (case_1 or case_2 or case_3 or case_4 or case_5 or case_6)
60
61 def permute_with_idx(M, E, idx_to_permute):
62
63     same_mass_with_idx = [idx for idx in range(len(M)) if M[idx] == M[idx_to_permute] and idx !=
    idx_to_permute and idx != 0 and idx != 1]
64     same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute])
    and idx != idx_to_permute]
65
66     return list(set(same_mass_with_idx) and set(same_energy_with_idx))
67
68 def permutation_boolean(M, E, idx_1, idx_2):
69
70     if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
71         return True
72     else:
73         return False

```



```

74
75 def construct_state():
76
77     rdm = randint(0, 2)
78
79     if rdm == 0:
80         ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
81         a = uniform(-10, 10) * e3
82         b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
83         c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
84         d = -a-b-c
85
86     if rdm == 1:
87         # This is the no permutation non-chiral case
88         ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
89         a = uniform(-10, 10) * e3
90         b = e1 + e2 + uniform(-10, 10)*e3
91         c = -e1 - e2 + uniform(-10, 10)*e3
92         d = -a-b-c
93
94     if rdm == 2:
95         # Permute 2 non chiral case
96         ma, mb, mc, md = 1, 1, 1, 1
97         a = uniform(-10, 10) * e3
98         angle = uniform(0, pi / 5)
99         b = sin(angle)*e1 + cos(angle)*e2 + uniform(-10, 10)*e3
100        c = cos(angle)*e1 + sin(angle)*e2 + b[3]*e3
101        d = -a-b-c
102
103        Ea, Eb, Ec, Ed = energy(ma, a), energy(mb, b), energy(mc, c), energy(md, d)
104
105        M = [ma, mb, mc, md]
106        E = [Ea, Eb, Ec, Ed]
107        S = [a, b, c, d]
108
109        return S, E, M
110
111 def chirality_test():
112
113     chiral_states = []
114     non_chiral_states = []
115
116     S, E, M = construct_state()
117     a, b, c, d = S[0], S[1], S[2], S[3]
118
119     permute_with_b = permute_with_idx(M, E, 1)
120     permute_with_c = permute_with_idx(M, E, 2)
121
122     S_parity = parity(S) # Perform parity on the set of momenta
123
124     flag = False # The flag is set to true if the state is non-chiral
125
126     R1 = e1 * e3
127     S1 = rotate(S_parity, R1) # Now a is fixed back to their original state
128
129     if (b[1] != 0) or (b[2] != 0): # if b has components in the 1-2 plane
130
131         for idx in permute_with_b + [1]:
132
133             if idx == 0:
134                 continue
135
136             x = S1[idx]
137             x_12 = x - x[3]*e3
138             b_12 = b - b[3]*e3
139
140             n = b_12 + x_12
141
142             if n == 0:

```

```

143         R2 = e1*e2
144
145     else:
146
147         R2 = b_12.normal()*n.normal()
148
149     S2 = rotate(S1, R2)
150     S3 = swap(S2, 1, idx) # Index 1 corresponds to b
151
152     if S == S3:
153         flag = True
154
155     if permutation_boolean(M, E, 2, 3): # If we can permute (cd)
156         S4 = swap(S3, 2, 3)
157         if S == S4:
158             flag = True
159
160     elif (c[1] != 0) or (c[2] != 0):
161
162         for idx in permute_with_c + [2]:
163
164             if idx == 0 or idx == 1:
165                 continue
166
167             x = S1[idx]
168             x_12 = x - x[3] * e3
169             c_12 = c - c[3] * e3
170
171             n = c_12 + x_12
172
173             if n == 0:
174
175                 R2 = e1 * e2
176
177             else:
178
179                 R2 = c_12.normal() * n.normal()
180
181             S2 = rotate(S1, R2)
182             S3 = swap(S2, 2, idx) # Index 2 corresponds to c
183
184             if S == S3:
185                 flag = True
186
187     if flag:
188         non_chiral_states.append([S, E])
189     else:
190         chiral_states.append([S, E])
191
192     return non_chiral_states, chiral_states
193
194 # non_chiral_states_list = []
195 # chiral_states_list = []
196 # for iterations in range(1000):
197 #     S, E, M = construct_state()
198 #     non_chiral_states, chiral_states = chirality_test()
199 #     non_chiral_states_list += non_chiral_states
200 #     chiral_states_list += chiral_states
201 #
202 # print(len(non_chiral_states_list), len(chiral_states_list))
203 #
204 # non_chiral_evaluation_on_logic_statement = [0, 0]
205 # for non_chiral_state in non_chiral_states_list:
206 #     mp, mq = uniform(1, 10), uniform(1, 10)
207 #     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
208 #     if flag:
209 #         non_chiral_evaluation_on_logic_statement[0] += 1
210 #     else:
211

```

```

212 #         non_chiral_evaluation_on_logic_statement[1] += 1
213 #
214 # chiral_evaluation_on_logic_statement = [0, 0]
215 # for chiral_state in chiral_states_list:
216 #     mp, mq = uniform(1, 10), uniform(1, 10)
217 #     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
218 #     if flag:
219 #         chiral_evaluation_on_logic_statement[0] += 1
220 #     else:
221 #         chiral_evaluation_on_logic_statement[1] += 1
222 #
223 # x = ['True', 'False']
224 # height_non_chiral = [non_chiral_evaluation_on_logic_statement[0],
225 #                     non_chiral_evaluation_on_logic_statement[1]]
226 # height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement
227 #                  [1]]
228 # plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
229 # plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is
230 #           non-chiral')
231 # plt.ylabel('Frequency')
232 # plt.show()
233 # plt.savefig('non_chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
234 #
235 # plt.bar(x, height_chiral, color = 'k', width = 0.1)
236 # plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-
237 #           chiral')
238 # plt.ylabel('Frequency')
239 # plt.show()
240 # plt.savefig('chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
241 #
242 # print(chirality_test())

```