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

*Email: takis.angelides@gmail.com

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 conluding 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**: $V_i : e \to \mathbb{R}$

- **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 $\vec{x} \to -\vec{x}$
- **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 $2 \to 4$ chiral, collision and non-collision events respectively. The classes of events called collision events $\mathscr{E}^c$ and non-collision events $\mathscr{E}^{nc}$ 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 $\mathscr{P}e$. 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 $\mathscr{P}$, a non-chiral event e is one for which there exists a set of group elements $\{g_1, ..., g_N\}$ such that $\mathscr{P}e = \hat{g}e$, where $\hat{g}$ is the composition of the N group elements in $\{g_1, ..., g_N\}$. 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

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

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 $m_i \geqslant 0$, $\mathbf{j} \in \mathbb{C}$ and $j_z \in \mathbb{R}$ for $j = a, b, c, d$. Given any $g \in G$ has an inverse, we can say

$$g \cdot \hat{e} = \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right) \cdot \mathscr{P} \cdot \hat{e} \Rightarrow \hat{e} \text{ is non-chiral} \tag{1}$$

Noting that

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

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

$$\left(\overset{\theta}{\underset{xy}{\curvearrowleft}}\right) \cdot h \cdot \hat{e} = \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right) \cdot \mathscr{P} \cdot \hat{e} \Rightarrow \hat{e} \text{ is non-chiral} \tag{2}$$

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.

[1] This 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

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

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

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

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

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

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]

$$\epsilon_{abpq} = \epsilon_{acpq} = \epsilon_{adpq} = \epsilon_{bdpq} = \epsilon_{bcpq} = \epsilon_{cdpq} = 0 \tag{3}$$

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 {outgoing 4-momenta}, giving $\binom{N}{2}$ terms.

[2] The $\vee$ 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.

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

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

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 $2 \to 3$ collision events. This new information inhibits a straightforward generalisation of results for $2 \to N$ 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.

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

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 $a \to b \to c \to d \to a$.

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

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

$h = \left(\underset{xy}{\overset{\theta}{\curvearrowleft}}\right) \cdot \left(\underset{yz}{\overset{\pi}{\curvearrowleft}}\right) \cdot (pq) \cdot 1_{S_4}$

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

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

$h = \left(\overset{\theta}{\underset{xy}{\curvearrowleft}}\right) \cdot \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right) \cdot (pq) \cdot (ab)$

$$
\begin{aligned}
&(p^2 = q^2) \wedge (a^2 = b^2) \wedge (G\begin{pmatrix} a+b, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \\
&\wedge (G\begin{pmatrix} a-b, & p+q \\ a+b, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} c, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} d, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \\
&\wedge [(\Delta_3\,(a,p,q) = 0) \vee (\Delta_3\,(a+b,p,q) = 0) \vee (\Delta_3\,(a-b,p,q) = 0)]
\end{aligned} \tag{8}
$$

Cases 27-31

Cases 27-31 follow case 26 by symmetry. Their elements are of the form $\left(\overset{\theta}{\underset{xy}{\curvearrowleft}}\right) \cdot \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right) \cdot (pq)$ 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

$h = \left(\overset{\theta}{\underset{xy}{\curvearrowleft}}\right) \cdot \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right) \cdot (pq) \cdot (dcb)$

$$
\begin{aligned}
&(p^2 = q^2) \wedge (b^2 = c^2 = d^2) \wedge (G\begin{pmatrix} a, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} b, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \\
&\wedge (G\begin{pmatrix} c, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} d, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} b-c, & p+q \\ b+c, & p+q \end{pmatrix} = 0) \\
&\wedge (G\begin{pmatrix} c-d, & p+q \\ c+d, & p+q \end{pmatrix} = 0) \wedge (G\begin{pmatrix} a, & p+q \\ p-q, & p+q \end{pmatrix} = 0) \\
&\wedge (\epsilon_{(b+d)cpq} = 0) \wedge (\epsilon_{(b+c)dpq} = 0) \wedge (\epsilon_{(c+d)bpq} = 0)
\end{aligned} \tag{9}
$$

Cases 33-39

Cases 33-39 are the 3-cyles 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

$$h = \begin{pmatrix} \theta \\ \curvearrowleft \\ xy \end{pmatrix} \cdot \begin{pmatrix} \pi \\ \curvearrowleft \\ yz \end{pmatrix} \cdot (pq) \cdot (ab)(cd)$$

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

Cases 41,42

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

Case 43

$$h = \begin{pmatrix} \theta \\ \curvearrowleft \\ xy \end{pmatrix} \cdot \begin{pmatrix} \pi \\ \curvearrowleft \\ yz \end{pmatrix} \cdot (pq) \cdot (dcba)$$

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

Cases 44-48

Cases 44 to 48 follow case 43 by symmetry and their elements are of the form $h = \begin{pmatrix} \theta \\ \curvearrowleft \\ xy \end{pmatrix} \cdot \begin{pmatrix} \pi \\ \curvearrowleft \\ yz \end{pmatrix} \cdot (pq)$ 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

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

The action of parity on this representation gives

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

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

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

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

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

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

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

$h = (ab)R$

$$(a^2 = b^2) \wedge a \neq b \wedge (G\binom{a, p+q}{a, p+q} = G\binom{b, p+q}{b, p+q})$$
$$\wedge\, (G\binom{a, p+q}{c, p+q} = G\binom{b, p+q}{c, p+q}) \wedge (G\binom{a, p+q}{d, p+q} = G\binom{b, p+q}{d, p+q}) \tag{16}$$

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

$h = (ab)(cd)R$

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

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

$h = (abcd)R$

$$(a^2 = b^2 = c^2 = d^2) \wedge (G\binom{a, p+q}{a, p+q} = G\binom{b, p+q}{b, p+q} = G\binom{c, p+q}{c, p+q} = G\binom{d, p+q}{d, p+q})$$
$$\wedge\, (G\binom{a-c, p+q}{b-d, p+q} = 0) \tag{18}$$

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 masssive - 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

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

We would like to act with a symmetry group on $R_y(\pi) \cdot \mathscr{P} \cdot e$ 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 $R_y(\pi) \cdot \mathscr{P} \cdot e$ is

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

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

$$(a + b + c + d)^2 = 0 \tag{21}$$

is always non-chiral.

Case 1

In the first case we have $h = R_z(\theta) \cdot 1_{S_4}$. The explicit form of $e = h \cdot R_y(\pi) \cdot \mathscr{P} \cdot e$ can be written as[3]

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

$$\left[\begin{array}{cc|cccc} 0 & 0 & m_a & m_b & m_c & m_d \\ 0 & 0 & |\mathbf{a}|e^{i(\theta-\alpha)} & |\mathbf{b}|e^{i(\theta-\beta)} & |\mathbf{c}|e^{i(\theta-\gamma)} & |\mathbf{d}|e^{i(\theta-\delta)} \\ p_z & q_z & a_z & b_z & c_z & d_z \end{array}\right] \tag{22}$$

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

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

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

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

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

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

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

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.

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

Case 8

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

$$
\begin{aligned}
&(a^2 = b^2) \wedge (G\binom{a-b, \Sigma}{p+q, \Sigma} = 0 \wedge (\Delta_2\,(a, \Sigma) = \Delta_2\,(b, \Sigma))\\
&\wedge (c^2 = d^2) \wedge (G\binom{c-d, \Sigma}{p+q, \Sigma} = 0 \wedge (\Delta_2\,(c, \Sigma) = \Delta_2\,(d, \Sigma))\\
&\wedge G\binom{a-b, \Sigma}{c+d, \Sigma} = 0
\end{aligned}
\tag{25}
$$

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 p = q = 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

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

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

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

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

$$uv = u \cdot v + u \wedge v \tag{28}$$

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

$$u \wedge v = -v \wedge u \tag{29}$$

$$u \wedge (v + w) = u \wedge v + u \wedge w \tag{30}$$

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

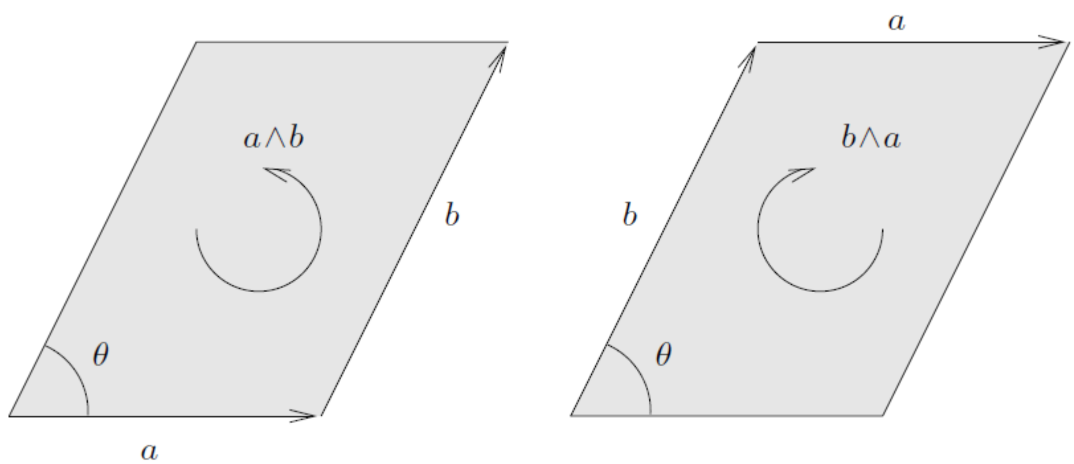


Figure 1: The outer product. The outer or wedge product of a and b returns a directed area element of area $|a||b|sin(\theta)$. The orientation of the parallelogram is defined by whether the circuit a, b, a, b is right-handed (anticlockwise) or left-handed (clockwise). Interchanging the order of the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

$$e_i e_j = e_i \wedge e_j \tag{31}$$

$$e_i e_j = -e_j e_i \tag{32}$$

$$(e_i e_j)^2 = -1 \tag{33}$$

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 rotation[4]. We define a rotor as

$$R = e^{-\frac{\theta}{2}B} = cos\Big(\frac{\theta}{2}\Big) - Bsin\Big(\frac{\theta}{2}\Big) \tag{34}$$

[4]Boosts - 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 described[5] in Figure 2, resulting in

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

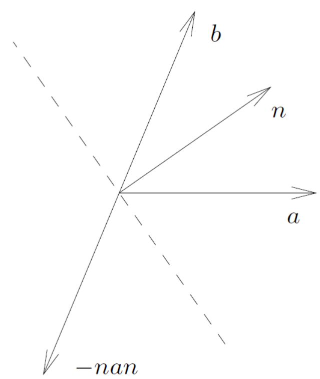


Figure 2: A rotation from a to b. The vector a is rotated onto b by first reflecting in the plane perpendicular to n, and then in the plane perpendicular to b. The vectors a, b and n all have unit length. Figure taken from [4].

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 $RSR^\dagger = [RpR^\dagger, RqR^\dagger, ..., RdR^\dagger]$, where all vectors are to be understood as 3-vectors. We can write the algorithm, that does not incorporate permutations, algebraically as follows

Steps[6]:

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

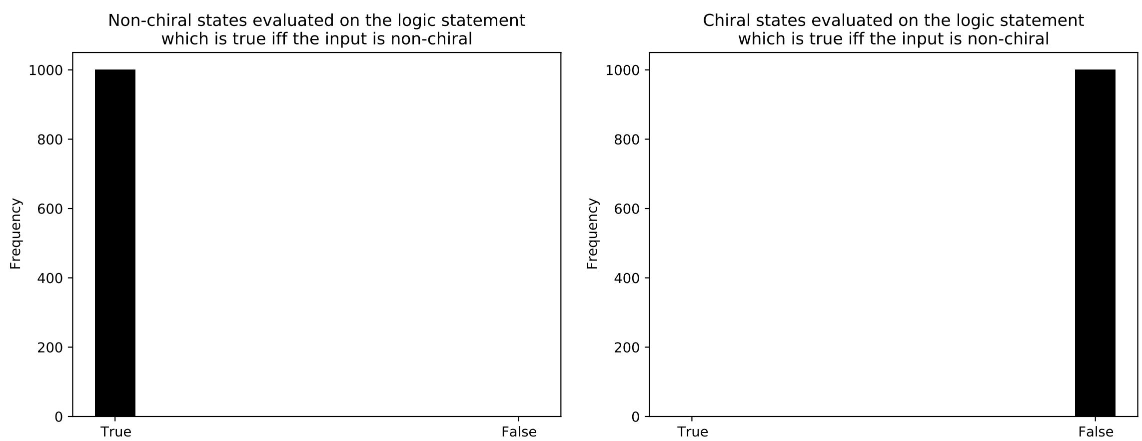
[6] Intuition: p, q, a_{12} define a plane in which we rotate by π using $R = a_{12}e_3$. Note that it does not matter here if we use a_{12} from S or S_1 since we are doing a π rotation.

1) Parity inversion: $S \to S_1 = -S$.
2) Project a into the 1-2 plane and normalise: $a_{12} = \frac{a-a_3}{\lvert a-a_3 \rvert}$.
3) Map p, q, a back to their original state with $R = a_{12}e_3$: $S_1 \to S_2 = RS_1R^\dagger$.
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.



(a) Non-chiral states

(b) Chiral states

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.

Steps[7]:

1) Parity inversion: $S \to S_1 = -S$.

2) Map a and b back to their original state with $R = \frac{a \wedge b}{|a \wedge b|}$: $S_1 \to S_2 = RS_1R^\dagger$.

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.

Steps[8]:

[7] Intuition: 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.

[8] Intuition: 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: $S \to S_1 = -S$.

2) Map p and q back to themselves with $R_1 = e_1 e_3 : S_1 \to S_2 = R_1 S_1 R_1^\dagger$.

3) In the 1-2 plane, rotate a to its original state with $R_2 = a_{12} a_{rot,12} : S_2 \to S_3 = R_2 S_2 R_2^\dagger$.

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.

Steps[9]:

1) Parity inversion: $S \to S_1 = -S$.

2) Map a back to its original state with $R_1 = e_1 e_3 : S_1 \to S_2 = R_1 S_1 R_1^\dagger$.

3) In the 1-2 plane, rotate b to its original state with $R_2 = b_{12} b_{rot,12} : S_2 \to S_3 = R_2 S_2 R_2^\dagger$.

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

[9] In 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.

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

a^2	b^2	$a^2 == b^2$	$a^2 ==$ **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 manifold[10] which represents the space of $2 \to 4$ events. The dimensionality comes from 6 masses and 6 3-momenta. For simplicity consider the compactification[11] 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 space[12] $\mathcal{O} = M/H$. 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 set[13] 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 $2 \to 3$ 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.

[10] An 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.

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

[12] This 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 $h \in H$.

[13] The 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 $2 \to 4$ 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 Laugher, and Florian Bruhin. pytest x.y, 2004.

[8] M. Hohenwarter, M. Borcherds, 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.

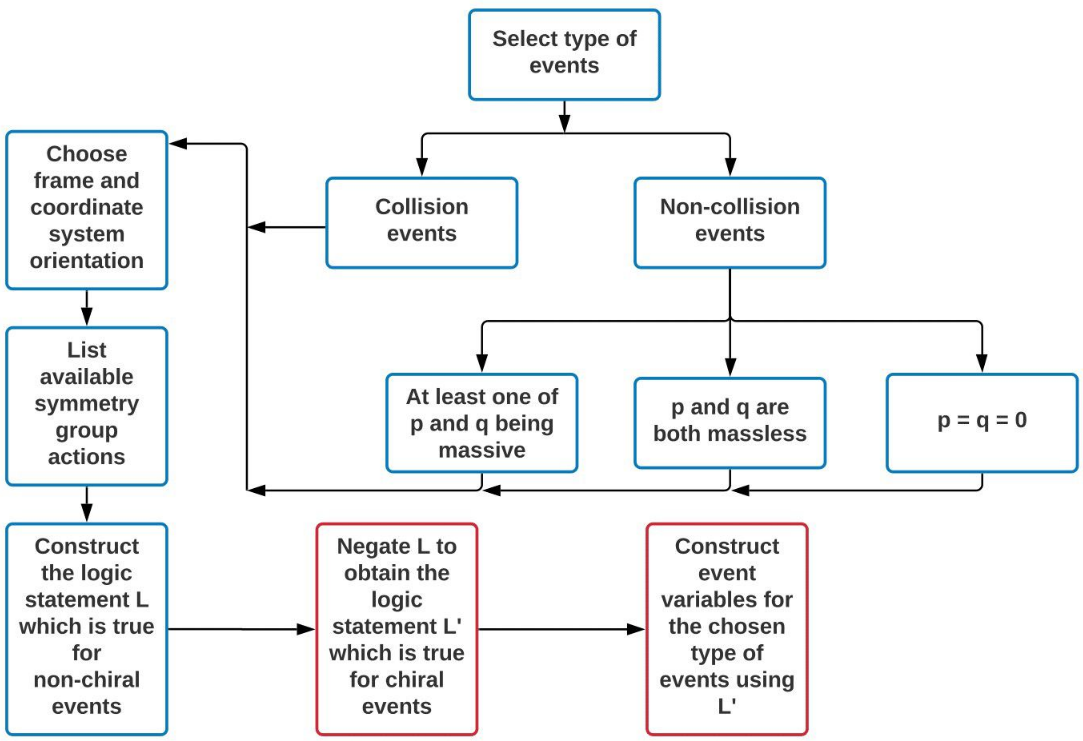


Figure 5: A flow chart of the method to go from events to event variables. The blue part is presented in this paper and the red part remains for future work.

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 $R_1 = e_1 e_3$. 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.

$\square$

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

For theses 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.

$\square$

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.

$\square$

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.

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

$$\Delta_2\left(a,b\right) \equiv G\binom{a,b}{a,b}$$

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

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

$$\left(\overset{\theta}{\underset{xy}{\curvearrowleft}}\right)\cdot h\cdot\hat{e} = \left(\overset{\pi}{\underset{yz}{\curvearrowleft}}\right)\cdot\mathscr{P}\cdot\hat{e} \tag{36}$$

we start from

$$[(m_a = m_b)\wedge(a_z = b_z)\wedge(|\mathbf{a}| = |\mathbf{b}|)]\wedge \tag{37}$$
$$[(|\mathbf{a}| = |\mathbf{b}| = 0)\vee((\theta+\beta = \pi + 2\pi n_a - \alpha)\wedge(\theta+\alpha = \pi+2\pi n_b - \beta))]\wedge \tag{38}$$
$$[(|\mathbf{c}| = 0)\vee(\theta+\gamma = \pi+2\pi n_c - \gamma)]\wedge \tag{39}$$
$$[(|\mathbf{d}| = 0)\vee(\theta+\delta = \pi+2\pi n_d - \delta)] \tag{40}$$

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)	$A_1B_1C_1$	**5)**	$A_2B_1C_1$
2)	$A_1B_2C_1$	**6)**	$A_2B_2C_1$
3)	$A_1B_1C_2$	**7)**	$A_2B_1C_2$
4)	$A_1B_2C_2$	**8)**	$A_2B_2C_2$

D.1 Subcase 1: $A_1B_1C_1$

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 algebraicly 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

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

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 $G\begin{pmatrix} a - b, p + q \\ c \quad , p + q \end{pmatrix} = 0$ 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 $m_a = m_b$.

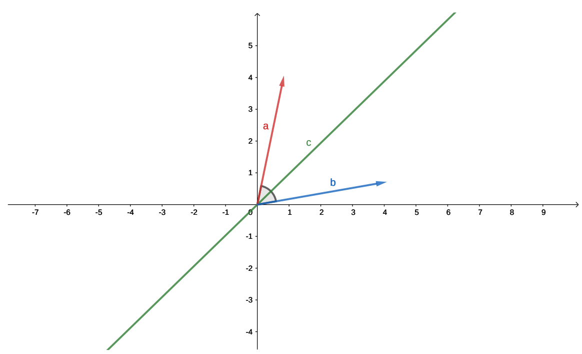


Figure 6: An example of $\mathbf{a}$ and $\mathbf{b}$ being reflections of each other in the ray defined by $\mathbf{c}$. The figure was constructed using [8].

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

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

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

D.7 Final result

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

$\square$

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

Code file: collision_events_algorithm.py

```
import time
from numpy import pi, cos, sin, e, tan, arctan
from clifford.g3 import blades
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from random import uniform, seed, randint
from sympy import LeviCivita as eps

# fig = plt.figure()
# ax = fig.add_subplot(111, projection='3d')
```

```python

e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
I = e1*e2*e3 # Pseudoscalar of 3D Euclidean geometric algebra

def plot_state(ax, p, q, a, b, c, d):

    X = (0)
    Y = (0)
    Z = (0)

    p1, p2, p3 = p[0], p[1], p[2]
    q1, q2, q3 = q[0], q[1], q[2]
    a1, a2, a3 = a[0], a[1], a[2]
    b1, b2, b3 = b[0], b[1], b[2]
    c1, c2, c3 = c[0], c[1], c[2]
    d1, d2, d3 = d[0], d[1], d[2]

    ax.quiver(X, Y, Z, p1, p2, p3, color='r', linestyle='-', label='p')
    ax.quiver(X, Y, Z, q1, q2, q3, color='k', linestyle='-', label='q')
    ax.quiver(X, Y, Z, a1, a2, a3, color='b', linestyle='-', label='a')
    ax.quiver(X, Y, Z, b1, b2, b3, color='g', linestyle='-', label='b')
    ax.quiver(X, Y, Z, c1, c2, c3, color='y', linestyle='-', label='c')
    ax.quiver(X, Y, Z, d1, d2, d3, color='orange', linestyle='-', label='d')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])
    limit = 10
    ax.set_xlim([-limit, limit])
    ax.set_ylim([-limit, limit])
    ax.set_zlim([-limit, limit])
    ax.view_init(elev=1, azim=pi / 2)
    plt.legend()

def plot_parity_state(ax, p, q, a, b, c, d):

    X = (0)
    Y = (0)
    Z = (0)

    p1, p2, p3 = -p[0], -p[1], -p[2]
    q1, q2, q3 = -q[0], -q[1], -q[2]
    a1, a2, a3 = -a[0], -a[1], -a[2]
    b1, b2, b3 = -b[0], -b[1], -b[2]
    c1, c2, c3 = -c[0], -c[1], -c[2]
    d1, d2, d3 = -d[0], -d[1], -d[2]

    ax.quiver(X, Y, Z, p1, p2, p3, color='r', linestyle='--', label='p')
    ax.quiver(X, Y, Z, q1, q2, q3, color='k', linestyle='--', label='q')
    ax.quiver(X, Y, Z, a1, a2, a3, color='b', linestyle='--', label='a')
    ax.quiver(X, Y, Z, b1, b2, b3, color='g', linestyle='--', label='b')
    ax.quiver(X, Y, Z, c1, c2, c3, color='y', linestyle='--', label='c')
    ax.quiver(X, Y, Z, d1, d2, d3, color='orange', linestyle='--', label='d')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])
    limit = 10
    ax.set_xlim([-limit, limit])
    ax.set_ylim([-limit, limit])
    ax.set_zlim([-limit, limit])
    ax.view_init(elev=1, azim=pi / 2)
    plt.legend()
    plt.show()

```

```python
# ------------------------------------------

# For a collision event:

def parity(S):
    return [-v for v in S]

def rotate(S, R):
    return [R*v*~R for v in S]

def multivec_to_vec(a):
    return np.array([a[1], a[2], a[3]])

def energy(m,p):
    p = multivec_to_vec(p)
    return np.sqrt(m**2 + np.linalg.norm(p)**2)

def epsilon(a, b, c, d):

    summation = 0

    for i in range(0, 4):
        for j in range(0, 4):
            for k in range(0, 4):
                for l in range(0, 4):
                    summation += eps(i, j, k, l) * a[i] * b[j] * c[k] * d[l]

    return summation

def dot(a, b):

    #  Minkowski metric

    return a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3]

def Gram_det_2(a,b,c,d):

    # a b
    # c d

    return (dot(a, c))*(dot(b, d)) - (dot(a, d))*(dot(b, c))

def sym_2_Gram_det(a,b):
    return Gram_det_2(a,b,a,b)

def sym_3_Gram_det(a,b,c):
    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)]]
    return np.linalg.det(M)

def swap(S,idx_1,idx_2):

    tmp = S[idx_1]
    S[idx_1] = S[idx_2]
    S[idx_2] = tmp

    return S

def permute_with_idx(M, E, idx_to_permute):

    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]
    same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == E[idx_to_permute] and idx != idx_to_permute]

    return list(set(same_mass_with_idx) and set(same_energy_with_idx))

def permutation_boolean(M, E, idx_1, idx_2):

    if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
```

```
        return True
    else:
        return False

def logic_statement_true_for_non_chiral(S, E):

    p = multivec_to_vec(S[0])
    p = np.insert(p, 0, E[0])
    q = multivec_to_vec(S[1])
    q = np.insert(q, 0, E[1])
    a = multivec_to_vec(S[2])
    a = np.insert(a, 0, E[2])
    b = multivec_to_vec(S[3])
    b = np.insert(b, 0, E[3])
    c = multivec_to_vec(S[4])
    c = np.insert(c, 0, E[4])
    d = multivec_to_vec(S[5])
    d = np.insert(d, 0, E[5])
    RF = p + q

    case_1 = ((epsilon(a,b,p,q) == 0) and (epsilon(a,c,p,q) == 0) and (epsilon(a,d,p,q) == 0) and (epsilon(b,d,p,q) == 0)
             and (epsilon(b,c,p,q) == 0) and (epsilon(c,d,p,q) == 0))

    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,RF,a+b,RF) == 0)
              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(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) and (Gram_det_2(a-b,RF,c,RF) == 0))
               or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (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) == 0))
               or ((sym_3_Gram_det(a,p,q) !=0) and (sym_3_Gram_det(b,p,q) != 0) and (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) == 0) and (Gram_det_2(a-b,RF,d,RF) == 0))))

    def case_2_symmetry(p, q, a, b, c, d):

        return ((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)
              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(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) and (Gram_det_2(a-b,RF,c,RF) == 0))
               or ((sym_3_Gram_det(a,p,q) != 0) and (sym_3_Gram_det(b,p,q) != 0) and (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) == 0))
               or ((sym_3_Gram_det(a,p,q) !=0) and (sym_3_Gram_det(b,p,q) != 0) and (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) == 0) and (Gram_det_2(a-b,RF,d,RF) == 0))))

    case_3 = case_2_symmetry(p, q, c, d, a, b)  # h = (cd)
    case_4 = case_2_symmetry(p, q, b, c, a, d)  # h = (bc)
    case_5 = case_2_symmetry(p, q, b, d, a, c)  # h = (bd)
    case_6 = case_2_symmetry(p, q, a, c, b, d)  # h = (ac)
    case_7 = case_2_symmetry(p, q, a, d, c, b)  # h = (ad)

    # TODO: In case 16 green and blue bracket are the negation of each other so I can evaluate once for efficiency

    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) == 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)
              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))
```

```
                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))))

def case_16_symmetry(p,q,a,b,c,d):

    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)
          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))
                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))))

case_17 = case_16_symmetry(p, q, a, c, b, d)  # h = (ac)(bd)
case_18 = case_16_symmetry(p, q, a, d, c, b)  # h = (ad)(bc)

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) \
          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) \
          and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (a == c) and (b == d))

def case_19_symmetry(p,q,a,b,c,d):

    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) \
          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) \
          and (Gram_det_2(c-d,RF,p-q,RF) == 0) and (a == c) and (b == d))

case_20 = case_19_symmetry(p,q,a,b,d,c) # h = (dbac)
case_22 = case_19_symmetry(p,q,a,c,d,b) # h = (bcad)

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) \
         and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0))

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)
           and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0)
           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))

def case_26_symmetry(p,q,a,b,c,d):

    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)
           and (Gram_det_2(c,RF,p-q,RF) == 0) and (Gram_det_2(d,RF,p-q,RF) == 0)
           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))

case_27 = case_26_symmetry(p, q, c, d, a, b)  # h = (cd)
case_28 = case_26_symmetry(p, q, c, b, a, d)  # h = (bc)
case_29 = case_26_symmetry(p, q, d, b, c, a)  # h = (bd)
case_30 = case_26_symmetry(p, q, a, c, b, d)  # h = (ac)
case_31 = case_26_symmetry(p, q, a, d, c, b)  # h = (ad)


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) \
          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) \
          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) \
          and (epsilon((b+c),d,p,q) == 0) and (epsilon((c+d),b,p,q) == 0))

def case_32_symmetry(p,q,a,b,c,d):

```

```python
        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) \
            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) \
            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) \
            and (epsilon((b+c),d,p,q) == 0) and (epsilon((c+d),b,p,q) == 0))

    case_34 = case_32_symmetry(p, q, d, b, c, a)  # h = (cba)
    case_35 = case_32_symmetry(p, q, c, a, d, b)  # h = (dba)
    case_37 = case_32_symmetry(p, q, b, a, c, d)  # h = (dca)

    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) \
            and (Gram_det_2(c+d,RF,p-q,RF) == 0) \
            and ((sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det(a-b,p,q) == 0)
                 or ((sym_3_Gram_det(c+d,p,q) == 0) or (sym_3_Gram_det(c-d,p,q) == 0))))

    def case_40_symmetry(p,q,a,b,c,d):

        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) \
            and (Gram_det_2(c+d,RF,p-q,RF) == 0) \
            and ((sym_3_Gram_det(a+b,p,q) == 0 or sym_3_Gram_det(a-b,p,q) == 0)
                 or ((sym_3_Gram_det(c+d,p,q) == 0) or (sym_3_Gram_det(c-d,p,q) == 0))))

    case_41 = case_40_symmetry(p, q, a, c, b, d)  # h = (ac)(bd)
    case_42 = case_40_symmetry(p, q, a, d, c, b)  # h = (ad)(cb)

    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)
    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)
    and (Gram_det_2(c+d,RF,p-q,RF) == 0) and ((sym_3_Gram_det(a,p,q) == 0)
    or ((epsilon(p+q,a-c,p,b) == 0) and (epsilon(p+q,b-d,a,p) == 0))
    or ((sym_3_Gram_det(a-c,p,q) == 0) and (sym_3_Gram_det(b-d,p,q) == 0)
        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))))))

    def case_43_symmetry(p,q,a,b,c,d):

        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)
    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)
    and (Gram_det_2(c+d,RF,p-q,RF) == 0) and ((sym_3_Gram_det(a,p,q) == 0)
    or ((epsilon(p+q,a-c,p,b) == 0) and (epsilon(p+q,b-d,a,p) == 0))
    or ((sym_3_Gram_det(a-c,p,q) == 0) and (sym_3_Gram_det(b-d,p,q) == 0)
        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))))))

    case_44 = case_43_symmetry(p, q, a, b, d, c)  # h = (dbac)
    case_45 = case_43_symmetry(p, q, b, a, c, d)  # h = (dcab)
    case_46 = case_43_symmetry(p, q, a, c, b, d)  # h = (dbca)
    case_47 = case_43_symmetry(p, q, c, b, a, d)  # h = (dabc)
    case_48 = case_43_symmetry(p, q, b, c, a, d)  # h = (dacb)

    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 or case_18
            or case_19 or case_20 or case_22 or case_25 or case_26 or case_27 or case_28 or case_29 or case_30
            or case_31 or case_32 or case_34 or case_35 or case_37 or case_40 or case_41 or case_42 or case_43
            or case_44 or case_45 or case_46 or case_47 or case_48)

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)
```

```
    p = uniform(-10, 10) * e3
    q = -p

    a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
    b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
    c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
    d = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3

    Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(
     mc, c), energy(md, d)

    M = [mp, mq, ma, mb, mc, md]
    E = [Ep, Eq, Ea, Eb, Ec, Ed]
    S = [p, q, a, b, c, d]

    return S, E, M

def construct_non_chiral_state(): # Trick

    mp, mq, ma, mb, mc, md = 1, 1, 1, 1, 1, 1

    p = uniform(-10, 10) * e3
    q = -p

    a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
    b = a[1] * e1 - a[2] * e2 + a[3] * e3
    R = e**(uniform(0,2*pi)*e1*e2)
    c = R*a*~R
    d = c[1] * e1 - c[2] * e2 + c[3] * e3

    Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(
     mc, c), energy(md, d)

    M = [mp, mq, ma, mb, mc, md]
    E = [Ep, Eq, Ea, Eb, Ec, Ed]
    S = [p, q, a, b, c, d]

    return S, E, M

def chirality_test():

    chiral_states = []
    non_chiral_states = []

    S, E, M = construct_non_chiral_state()
    a, b, c, d = S[2], S[3], S[4], S[5]

    # These lists hold indices (as they appear in S) of the particles that can be permuted with
     a,b,c and d respectively
    permute_with_a = permute_with_idx(M, E, 2)
    permute_with_b = permute_with_idx(M, E, 3)
    permute_with_c = permute_with_idx(M, E, 4)
    permute_with_d = permute_with_idx(M, E, 5)

    permutation_dictionary = {3: permute_with_b, 4: permute_with_c, 5: permute_with_d}

    S_parity = parity(S)  # Perform parity on the set of momenta

    flag = False

    if not permutation_boolean(M, E, 0, 1): # If we cant permute p (index 0) with q (index 1)

        if (a[1] == 0 and a[2] == 0) or (a == 0):

            if (b[1] == 0 and b[2] == 0) or (b == 0):

                if (c[1] == 0 and c[2] == 0) or (c == 0):

```

```
                flag = True

            else: # Here we consider that a,b are either e3 collinear or 0, only c and d are in the 1-2 plane

                # We can map c_12 to its original or d_12 to the original c_12 and swap (cd) if they can be
                # permuted.

                # Since a and b are fixed by R1 = e1e3 we only want to consider permutations (cd)
                # 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)])

                R1 = e1*e3

                S1 = rotate(S_parity, R1)

                for idx in permute_with_c_new+[4]: # The index of c is 4 in the list S

                    # Map x_12_rotated to c_12_original

                    x = S1[idx]
                    x_12 = x - x[3]*e3
                    c_12 = c - c[3]*e3
                    n = (x_12 + c_12).normal()
                    if n == 0: # If x_12 and c_12 are anti-parallel then we need a pi rotation
                        R2 = e1*e2
                    else: # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                        R2 = c_12.normal()*n
                    S2 = rotate(S1, R2)
                    S3 = swap(S2, idx, 4)

                    if S == S3:
                        flag = True

        else: # Here we consider that a is collinear with e3 or is 0 but b is not

            # We map x_12 to b_12_original and consider permutations between (cd), where x_12 can be (b,c,d)_12

            permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])

            R1 = e1 * e3

            S1 = rotate(S_parity, R1)

            for idx in permute_with_b_new + [3]:  # The index of b is 3 in the list S

                # Map x_12_rotated to b_12_original

                x = S1[idx]
                x_12 = x - x[3] * e3
                b_12 = b - b[3] * e3
                n = (x_12 + b_12).normal()
                if n == 0:  # If x_12 and b_12 are anti-parallel then we need a pi rotation
                    R2 = e1 * e2
                else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                    R2 = b_12.normal() * n
                S2 = rotate(S1, R2)
                S3 = swap(S2, idx, 3)

                if S == S3:
                    flag = True

```

```
                    if permutation_boolean(M, E, 4, 5): # If (cd) is possible

                        S4 = swap(S3, 4, 5)

                        if S == S4:

                            flag = True

        else: # Here we consider that a is not collinear with e3 and not 0

            # We need to map x_12 to a_12_original and consider permutations (bcd)

            R1 = e1 * e3

            S1 = rotate(S_parity, R1)

            for idx in permute_with_a + [2]:  # The index of a is 2 in the list S

                # Map x_12_rotated to a_12_original

                x = S1[idx]
                x_12 = x - x[3] * e3
                a_12 = a - a[3] * e3
                n = (x_12 + a_12).normal()
                if n == 0:  # If x_12 and a_12 are anti-parallel then we need a pi rotation
                    R2 = e1 * e2
                else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                    R2 = a_12.normal() * n
                S2 = rotate(S1, R2)
                S3 = swap(S2, idx, 2)

                if S == S3:
                    flag = True

                flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available

                if flag_tmp_1:
                    S4 = swap(S3, 3, 4)
                    if S == S4:
                        flag = True

                flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available

                if flag_tmp_2:
                    S5 = swap(S3, 3, 5)
                    if S == S5:
                        flag = True

                if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
                    # The following achieves b->c->d->b
                    S6 = swap(S3, 3, 4) # (bc)
                    S6 = swap(S6, 3, 5) # (bd), here in the 3 index lies c but we name it b still, notice we use S6
                    if S == S6:
                        flag = True
                    # The following achieves b->d->c->b
                    S7 = swap(S3, 3, 5)  # (bd)
                    S7 = swap(S7, 3, 4)  # (bc), here in the 3 index lies d but we name it b still, notice we use S7
                    if S == S7:
                        flag = True

                if permutation_boolean(M, E, 4, 5):  # If we have (cd)

                    S8 = swap(S3, 4, 5)
                    if S == S8:
                        flag = True

```

```
    else: # Now we consider the case where (pq) is available

        # We can either try the above or omit using R1 and just use (pq)
        # Whenever we check for non-chirality we can can also perform a pi rotation in the plane that contains the
        # 3-axis and has normal the final_state_particle_12 we matched in the 1-2 plane (usually this is a unless a has
        # no 1,2 components)

        # The following is repeating the above but incorporating the (pq) degree of freedom

        if (a[1] == 0 and a[2] == 0) or (a == 0):

            if (b[1] == 0 and b[2] == 0) or (b == 0):

                if (c[1] == 0 and c[2] == 0) or (c == 0):

                    flag = True

                else: # Here we consider that a,b are either e3 collinear or 0, only c and d are in the 1-2 plane

                    # We can map c_12 to its original or d_12 to the original c_12 and swap (cd) if they can be
                    # permuted.

                    # Since a and b are fixed by R1 = e1e3 we only want to consider permutations (cd)
                    # 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)])

                    R1 = e1*e3

                    S1 = rotate(S_parity, R1)

                    for idx in permute_with_c_new+[4]: # The index of c is 4 in the list S

                        # Map x_12_rotated to c_12_original

                        x = S1[idx]
                        x_12 = x - x[3]*e3
                        c_12 = c - c[3]*e3
                        n = (x_12 + c_12).normal()
                        if n == 0: # If x_12 and c_12 are anti-parallel then we need a pi rotation
                            R2 = e1*e2
                        else: # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                            R2 = c_12.normal()*n
                        S2 = rotate(S1, R2)
                        S3 = swap(S2, idx, 4)

                        if S == S3:
                            flag = True

                        # Now we can try the pi rotation in the plane v-e3 where v is perpendicular c_12 and e3

                        v = (-I*(e3^c_12)).normal() # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                        R3 = e3^v

                        S4 = rotate(S3, R3)
                        S5 = swap(S4, 0, 1)

                        if S == S5:
                            flag = True
```

```
            else: # Here we consider that a is collinear with e3 or is 0 but b is not

                # We map x_12 to b_12_original and consider permutations between (cd), where x_12 can be (b,c,d)_12

                permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])

                R1 = e1 * e3

                S1 = rotate(S_parity, R1)

                for idx in permute_with_b_new + [3]:  # The index of b is 3 in the list S

                    # Map x_12_rotated to b_12_original

                    x = S1[idx]
                    x_12 = x - x[3] * e3
                    b_12 = b - b[3] * e3
                    n = (x_12 + b_12).normal()
                    if n == 0:  # If x_12 and b_12 are anti-parallel then we need a pi rotation
                        R2 = e1 * e2
                    else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                        R2 = b_12.normal() * n
                    S2 = rotate(S1, R2)
                    S3 = swap(S2, idx, 3)

                    if S == S3:
                        flag = True

                    v = (-I * (e3 ^ b_12)).normal()  # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                    R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v

                    S4 = rotate(S3, R3)
                    S5 = swap(S4, 0, 1)

                    if S == S5:
                        flag = True

                    if permutation_boolean(M, E, 4, 5): # If (cd) is possible

                        S6 = swap(S3, 4, 5)

                        if S == S6:

                            flag = True

                        S7 = rotate(S6, R3)
                        S8 = swap(S7, 0, 1)

                        if S == S8:
                            flag = True

        else: # Here we consider that a is not collinear with e3 and not 0

            # We need to map x_12 to a_12_original and consider permutations (bcd)

            R1 = e1 * e3

            S1 = rotate(S_parity, R1)

            for idx in permute_with_a + [2]:  # The index of a is 2 in the list S

                # Map x_12_rotated to a_12_original

                x = S1[idx]
                x_12 = x - x[3] * e3
```

```
                a_12 = a - a[3] * e3
                n = (x_12 + a_12).normal()
                if n == 0:  # If x_12 and a_12 are anti-parallel then we need a pi rotation
                    R2 = e1 * e2
                else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                    R2 = a_12.normal() * n
                S2 = rotate(S1, R2)
                S3 = swap(S2, idx, 2)

                if S == S3:
                    flag = True

                v = (-I * (e3 ^ a_12)).normal()  # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                R3 = e3 ^ v  # (pq) degree of freedom, rotation by pi in the plane e3-v

                S4 = rotate(S3, R3)
                S5 = swap(S4, 0, 1)

                if S == S5:
                    flag = True

                flag_tmp_1 = permutation_boolean(M, E, 3, 4) # This checks if (bc) is available

                if flag_tmp_1:
                    S6 = swap(S3, 3, 4)
                    if S == S6:
                        flag = True

                    S7 = rotate(S6, R3)
                    S8 = swap(S7, 0, 1)

                    if S == S8:
                        flag = True

                flag_tmp_2 = permutation_boolean(M, E, 3, 5) # This checks if (bd) is available

                if flag_tmp_2:
                    S9 = swap(S3, 3, 5)
                    if S == S9:
                        flag = True
                    S10 = rotate(S9, R3)
                    S11 = swap(S10, 0, 1)

                    if S == S11:
                        flag = True



                if flag_tmp_1 and flag_tmp_2: # If we have (bc) and (bd) then we have (bcd)
                    # The following achieves b->c->d->b
                    S12 = swap(S3, 3, 4) # (bc)
                    S13 = swap(S12, 3, 5) # (bd), here in the 3 index lies c but we name it b still, notice we use S6
                    if S == S13:
                        flag = True
                    S14 = rotate(S13, R3)
                    S15 = swap(S14, 0, 1)

                    if S == S15:
                        flag = True
                    # The following achieves b->d->c->b
                    S16 = swap(S3, 3, 5)  # (bd)
                    S17 = swap(S16, 3, 4)  # (bc), here in the 3 index lies d but we name it b still, notice we use S7
                    if S == S17:
                        flag = True
                    S18 = rotate(S17, R3)
```

```
                S19 = swap(S18, 0, 1)

                if S == S19:
                    flag = True

            if permutation_boolean(M, E, 4, 5):  # If we have (cd)

                S20 = swap(S3, 4, 5)
                if S == S20:
                    flag = True
                S21 = rotate(S20, R3)
                S22 = swap(S21, 0, 1)

                if S == S22:
                    flag = True


    # The following is omitting R1 and just uses (pq), it again incorporates the (pq) degree of freedom

    if (a[1] == 0 and a[2] == 0) or (a == 0):

        if (b[1] == 0 and b[2] == 0) or (b == 0):

            if (c[1] == 0 and c[2] == 0) or (c == 0):

                flag = True

            else:  # Here we consider that a,b are either e3 collinear or 0, only c and d are in the 1-2 plane

                # We can map c_12 to its original or d_12 to the original c_12 and swap (cd) if they can be
                # permuted.

                # Since a and b are fixed by R1 = e1e3 we only want to consider permutations (cd)
                # 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)])

                S1 = swap(S_parity, 0, 1)

                for idx in permute_with_c_new + [4]:  # The index of c is 4 in the list S

                    # Map x_12_rotated to c_12_original

                    x = S1[idx]
                    x_12 = x - x[3] * e3
                    c_12 = c - c[3] * e3
                    n = (x_12 + c_12).normal()
                    if n == 0:  # If x_12 and c_12 are anti-parallel then we need a pi rotation
                        R2 = e1 * e2
                    else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                        R2 = c_12.normal() * n
                    S2 = rotate(S1, R2)
                    S3 = swap(S2, idx, 4)

                    if S == S3:
                        flag = True

                    # Now we can try the pi rotation in the plane v-e3 where v is perpendicular c_12 and e3

                    v = (-I * (e3 ^ c_12)).normal()  # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                    R3 = e3 ^ v
```

```

                        S4 = rotate(S3, R3)
                        S5 = swap(S4, 0, 1)

                        if S == S5:
                            flag = True

            else:  # Here we consider that a is collinear with e3 or is 0 but b is not

                # We map x_12 to b_12_original and consider permutations between (cd), where x_12 can be (b,c,d)_12

                permute_with_b_new = list([idx for idx in permute_with_b if (idx != 2)])

                S1 = swap(S_parity, 0, 1)

                for idx in permute_with_b_new + [3]:  # The index of b is 3 in the list S

                    # Map x_12_rotated to b_12_original

                    x = S1[idx]
                    x_12 = x - x[3] * e3
                    b_12 = b - b[3] * e3
                    n = (x_12 + b_12).normal()
                    if n == 0:  # If x_12 and b_12 are anti-parallel then we need a pi rotation
                        R2 = e1 * e2
                    else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                        R2 = b_12.normal() * n
                    S2 = rotate(S1, R2)
                    S3 = swap(S2, idx, 3)

                    if S == S3:
                        flag = True

                    v = (-I * (e3 ^ b_12)).normal()  # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                    R3 = e3 ^ v # (pq) degree of freedom, rotation by pi in the plane e3-v

                    S4 = rotate(S3, R3)
                    S5 = swap(S4, 0, 1)

                    if S == S5:
                        flag = True

                    if permutation_boolean(M, E, 4, 5):  # If (cd) is possible

                        S6 = swap(S3, 4, 5)

                        if S == S6:
                            flag = True

                        S7 = rotate(S6, R3)
                        S8 = swap(S7, 0, 1)

                        if S == S8:
                            flag = True

        else:  # Here we consider that a is not collinear with e3 and not 0

            # We need to map x_12 to a_12_original and consider permutations (bcd)

            S1 = swap(S_parity, 0, 1)

            for idx in permute_with_a + [2]:  # The index of a is 2 in the list S

                # Map x_12_rotated to a_12_original

                x = S1[idx]
```

```
                x_12 = x - x[3] * e3
                a_12 = a - a[3] * e3
                n = (x_12 + a_12).normal()
                if n == 0:  # If x_12 and a_12 are anti-parallel then we need a pi rotation
                    R2 = e1 * e2
                else:  # Otherwise we construct the rotor as usual with the form R = (final destination)*n
                    R2 = a_12.normal() * n
                S2 = rotate(S1, R2)
                S3 = swap(S2, idx, 2)

                if S == S3:
                    flag = True

                v = (-I * (e3 ^ a_12)).normal()  # Cross product in geometric algebra, I is the pseudoscalar, v = e3 x c_12
                R3 = e3 ^ v  # (pq) degree of freedom, rotation by pi in the plane e3-v

                S4 = rotate(S3, R3)
                S5 = swap(S4, 0, 1)

                if S == S5:
                    flag = True

                flag_tmp_1 = permutation_boolean(M, E, 3, 4)  # This checks if (bc) is available

                if flag_tmp_1:
                    S6 = swap(S3, 3, 4)
                    if S == S6:
                        flag = True

                    S7 = rotate(S6, R3)
                    S8 = swap(S7, 0, 1)

                    if S == S8:
                        flag = True

                flag_tmp_2 = permutation_boolean(M, E, 3, 5)  # This checks if (bd) is available

                if flag_tmp_2:
                    S9 = swap(S3, 3, 5)
                    if S == S9:
                        flag = True
                    S10 = rotate(S9, R3)
                    S11 = swap(S10, 0, 1)

                    if S == S11:
                        flag = True

                if flag_tmp_1 and flag_tmp_2:  # If we have (bc) and (bd) then we have (bcd)
                    # The following achieves b->c->d->b
                    S12 = swap(S3, 3, 4)  # (bc)
                    S13 = swap(S12, 3, 5)  # (bd), here in the 3 index lies c but we name it b still, notice we use S6
                    if S == S13:
                        flag = True
                    S14 = rotate(S13, R3)
                    S15 = swap(S14, 0, 1)

                    if S == S15:
                        flag = True
                    # The following achieves b->d->c->b
                    S16 = swap(S3, 3, 5)  # (bd)
                    S17 = swap(S16, 3, 4)  # (bc), here in the 3 index lies d but we name it b still, notice we use S7
                    if S == S17:
                        flag = True
                    S18 = rotate(S17, R3)
                    S19 = swap(S18, 0, 1)
```

```python

                    if S == S19:
                        flag = True

                if permutation_boolean(M, E, 4, 5):  # If we have (cd)

                    S20 = swap(S3, 4, 5)
                    if S == S20:
                        flag = True
                    S21 = rotate(S20, R3)
                    S22 = swap(S21, 0, 1)

                    if S == S22:
                        flag = True

    if flag:
        non_chiral_states.append([S, E])
    else:
        chiral_states.append([S, E])

    return non_chiral_states, chiral_states

# chiral_states = []
# non_chiral_states = []
#
# for _ in range(10):
#
#     non_chiral_states_tmp, chiral_states_tmp = chirality_test()
#     non_chiral_states += non_chiral_states_tmp
#     chiral_states += chiral_states_tmp
#
# # The first slot is incremented for every true and the second for every false
# non_chiral_evaluation_on_logic_statement = [0, 0]
# chiral_evaluation_on_logic_statement = [0, 0]
#
# for non_chiral_state in non_chiral_states:
#     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
#     if flag:
#         non_chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         non_chiral_evaluation_on_logic_statement[1] += 1
#
# for chiral_state in chiral_states:
#     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
#     if flag:
#         chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         chiral_evaluation_on_logic_statement[1] += 1
#
# x = ['True', 'False']
# height_non_chiral = [non_chiral_evaluation_on_logic_statement[0], non_chiral_evaluation_on_logic_statement[1]]
# height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement[1]]
#
# plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
# plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# plt.show()
# # plt.savefig('non_chiral_collision_logic_statement_test.pdf', bbox_inches='tight')
# #
# plt.bar(x, height_chiral, color = 'k', width = 0.1)
# plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# plt.show()
# plt.savefig('chiral_collision_logic_statement_test.pdf', bbox_inches='tight')

```

```
# ------------------------------------------
```

Code file: non_collision_one_massive_algorithm.py

```
import time
from numpy import pi, cos, sin, e, tan, arctan
from clifford.g3 import blades
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from random import uniform, seed, randint
from sympy import LeviCivita as eps
from main import parity, rotate, energy, epsilon, Gram_det_2
from pytest import approx

# Works only for a,b,c,d != 0 (it is very unlikely that any one will be randomly generated in
    momentum 0)

e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
I = e1^e2^e3

def dot(a,b):
    dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
    return dot

def multivec_to_vec(a):
    return np.array([a[1], a[2], a[3]])

def swap(S,idx_1,idx_2):

    tmp = S[idx_1]
    S[idx_1] = S[idx_2]
    S[idx_2] = tmp

    return S

def logic_statement_true_for_non_chiral(S, E, mp, mq):

    p = np.array([mp, 0, 0, 0])
    q = np.array([mq, 0, 0, 0])
    a = multivec_to_vec(S[0])
    a = np.insert(a, 0, E[0])
    b = multivec_to_vec(S[1])
    b = np.insert(b, 0, E[1])
    c = multivec_to_vec(S[2])
    c = np.insert(c, 0, E[2])
    d = multivec_to_vec(S[3])
    d = np.insert(d, 0, E[3])
    RF = p + q

    case_1 = ((epsilon(a, b, c, RF) == approx(0)) and (epsilon(a, b, d, RF) == approx(0)) and (
    epsilon(b, c, d, RF) == approx(0)) and (
                 epsilon(a, c, d, RF) == approx(0)))

    case_2 = ((dot(a,a) == approx(dot(b,b))) and (not (a == b).all()) and (Gram_det_2(a,RF,a,RF)
     == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(a,RF,c,RF) == approx((Gram_det_2(b,RF,c,
    RF)))) and (Gram_det_2(a,RF,d,RF) == approx(Gram_det_2(b,RF,d,RF))))

    def case_2_symmetry(a,b,c,d):
        return ((dot(a,a) == approx(dot(b,b))) and (not (a == b).all()) and (Gram_det_2(a,RF,a,
    RF) == approx(Gram_det_2(b,RF,b,RF))) and (Gram_det_2(a,RF,c,RF) == approx((Gram_det_2(b,RF
    ,c,RF) == 0))) and (Gram_det_2(a,RF,d,RF) == approx(Gram_det_2(b,RF,d,RF))))

    case_3 = case_2_symmetry(a,c,b,d)
    case_4 = case_2_symmetry(a,d,c,b)
    case_5 = case_2_symmetry(b,c,a,d)
    case_6 = case_2_symmetry(b,d,c,a)
    case_7 = case_2_symmetry(c,d,a,b)

```

```
    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()))))

    def case_8_symmetry(a,b,c,d):
        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()))))

    case_9 = case_8_symmetry(a,c,b,d)
    case_10 = case_8_symmetry(a,d,b,c)

    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)))

    def case_19_symmetry(a,b,c,d):
        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)))

    case_20 = case_19_symmetry(a,b,d,c)
    case_21 = case_19_symmetry(a,c,b,d)
    case_22 = case_19_symmetry(a,c,d,b)
    case_23 = case_19_symmetry(a,d,c,b)
    case_24 = case_19_symmetry(a,d,b,c)

    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
            or case_19 or case_20 or case_21 or case_22 or case_23 or case_24)

def construct_state():

    B = (uniform(-10, 10) * (e1 ^ e2) + uniform(-10, 10) * (e1 ^ e3) + uniform(-10, 10) * (e2 ^
     e3)).normal()
    R = (e ^ (uniform(0, 2 * pi) * B)).normal()

    rdm = randint(0, 4)

    if rdm == 0:
        ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
        a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        d = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3

    # Type of non-chiral (checked) (ab)
    if rdm == 1:
        ma, mb, mc, md = 1, 1, 2, 3
        a = e1
        b = e2
        c = e3
        d = (-e3)

    # Type of non-chiral (checked) (ab)(cd)
    if rdm == 2:
        ma, mb, mc, md = 1, 1, 2, 2
        n = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
        n = n.normal()
        a = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
        b = -n*a*n
        c = uniform(-10, 10)*e1 + uniform(-10, 10)*e2 + uniform(-10, 10)*e3
        d = -n*c*n

    # Type of non-chiral (checked) (abc)
```

```
    if rdm == 3:
        ma, mb, mc, md = 1, 1, 1, 3
        mag = uniform(-10, 10)
        a = mag*e1 + mag*e2
        b = mag*e1 - mag*e2
        c = -mag*e1 + mag*e2
        d = mag*e1 - mag*e2

    # Type of non-chiral (checked) (abcd)
    if rdm == 4:
        ma, mb, mc, md = 1, 1, 1, 1
        a = R*(-e1 + e3)*~R
        b = R*(e1 + e2)*~R
        c = R*(-e1 - e3)*~R
        d = R*(e1 - e2)*~R

    Ea, Eb, Ec, Ed = energy(ma, a), energy(mb, b), energy(mc, c), energy(md, d)

    M = [ma, mb, mc, md]
    E = [Ea, Eb, Ec, Ed]
    S = [a, b, c, d]

    print(rdm)

    return S, E, M

def permute_with_idx(M, E, idx_to_permute):

    same_mass_with_idx = [idx for idx in range(len(M)) if M[idx] == M[idx_to_permute] and idx != idx_to_permute]
    same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute]) and idx != idx_to_permute]

    return list(set(same_mass_with_idx) and set(same_energy_with_idx))

def permutation_boolean(M, E, idx_1, idx_2):

    if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
        return True
    else:
        return False

def chirality_test():

    chiral_states = []
    non_chiral_states = []

    S, E, M = construct_state()
    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 list

    # These lists hold indices (as they appear in S) of the particles that can be permuted with a,b,c and d respectively
    permute_with_a = permute_with_idx(M, E, 0)
    permute_with_b = permute_with_idx(M, E, 1)
    permute_with_c = permute_with_idx(M, E, 2)
    permute_with_d = permute_with_idx(M, E, 3)

    S_parity = parity(S)  # Perform parity on the set of momenta

    flag = False # The flag is set to true if the state is non-chiral

    for idx in permute_with_a+[0]: # For every x that can be permuted with a, map x to a and perform (ax)

        x = S_parity[idx]
        n = a+x
        if n == 0: # If a and x are collinear we construct any v perpendicular to a and do a pi rotation in the plane av
```

```
        if a[2] != 0 or a[3] != 0:
            v = -I*(a^e1) # Cross product a x e1 in geometric algebra, I is the pseudoscalar I = e1e2e3
        else:
            v = -I*(a^e2)
        R1 = (a^v).normal()
    else:
        R1 = a.normal()*n.normal()

    S1 = rotate(S_parity, R1)
    S2 = swap(S1, 0, idx) # Performs (ax) where the index 0 corresponds to a in the S list

    # At this point a is fixed to its original state

    # 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

    if a^b != 0: # If b is not collinear to a then it has components in the plane perpendicular to a

        for idx_plane in permute_with_b + [1]:

            if idx_plane == 0: # Since a is already fixed
                continue

            # If we have a plane P with its perpendicular being a and we have a vector y, then the component of y
            # in the plane P is given in geometric algebra by the rejection y_plane = a*(a^y)

            b_plane = a*(a^b)
            y = S2[idx_plane]
            y_plane = a*(a^y)

            # If y is collinear with a then it has no component in the plane perpendicular to a, so even if it can
            # be permuted with b, we cannot map y_plane to b_plane and then swap because y_plane is 0

            if y_plane == 0:
                continue
            else: # Here we map y_plane to b_plane and perform (yb)
                m = y_plane + b_plane
                if m == 0: # In this case we need a pi rotation in this plane we are working in
                    # We construct another vector in this plane with the cross product a x b_plane
                    w_plane = -I*(a^b_plane)
                    R2 = (w_plane^b_plane).normal()
                else:
                    R2 = b_plane.normal()*m.normal()

            S3 = rotate(S2, R2) # Map y_plane to b_plane
            S4 = swap(S3, 1, idx_plane) # Perform (yb)

            if S == S4:
                flag = True

            if permutation_boolean(M, E, 2, 3): # If we can perform (cd)
                S5 = swap(S4, 2, 3)
                if S == S5:
                    flag = True


    elif a^c != 0:

        for idx_plane in permute_with_c + [2]:

            # Since a and b are already fixed, b is fixed because we fixed a and to get into
```

```
this elif we need
                # b to be collinear with a and hence when we fixed a we automatically fixed b
                if idx_plane == 0 or idx_plane == 1:
                    continue

                # If we have a plane P with its perpendicular being a and we have a vector y, then the component of y
                # in the plane P is given in geometric algebra by the rejection y_plane = a*(a^y)

                c_plane = a * (a ^ c)
                y = S2[idx_plane]
                y_plane = a * (a ^ y)

                # If y is collinear with a then it has no component in the plane perpendicular to a, so even if it can
                # be permuted with c, we cannot map y_plane to c_plane and then swap because y_plane is 0

                if y_plane == 0:
                    continue
                else:  # Here we map y_plane to c_plane and perform (yc)
                    m = y_plane + c_plane
                    if m == 0:  # In this case we need a pi rotation in this plane we are working in
                        # We construct another vector in this plane with the cross product a x c_plane
                        w_plane = -I * (a ^ c_plane)
                        R2 = (w_plane ^ c_plane).normal()
                    else:
                        R2 = c_plane.normal() * m.normal()

                S6 = rotate(S2, R2)  # Map y_plane to c_plane
                S7 = swap(S6, 2, idx_plane)  # Perform (yc)

                if S == S7:
                    flag = True

                if permutation_boolean(M, E, 1, 3): # If we can perform (bd)
                    S8 = swap(S7, 1, 3)
                    if S == S8:
                        flag = True

        else:
            flag = True # If a,b,c are collinear then the state is non-chiral

    if flag:
        non_chiral_states.append([S, E])
    else:
        chiral_states.append([S, E])

    return non_chiral_states, chiral_states

# non_chiral_states_list = []
# chiral_states_list = []
# for iterations in range(1000):
#     S, E, M = construct_state()
#     non_chiral_states, chiral_states = chirality_test()
#     non_chiral_states_list += non_chiral_states
#     chiral_states_list += chiral_states
#
# print(len(non_chiral_states_list), len(chiral_states_list))
#
# non_chiral_evaluation_on_logic_statement = [0, 0]
# for non_chiral_state in non_chiral_states_list:
#     mp, mq = uniform(1, 10), uniform(1, 10)
#     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1], mp, mq)
#     if flag:
```

```
#         non_chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         non_chiral_evaluation_on_logic_statement[1] += 1
#
# chiral_evaluation_on_logic_statement = [0, 0]
# for chiral_state in chiral_states_list:
#     mp, mq = uniform(1, 10), uniform(1, 10)
#     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1], mp, mq)
#     if flag:
#         chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         chiral_evaluation_on_logic_statement[1] += 1
#
# x = ['True', 'False']
# height_non_chiral = [non_chiral_evaluation_on_logic_statement[0], non_chiral_evaluation_on_logic_statement[1]]
# height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement[1]]
#
# plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
# plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# #plt.show()
# #plt.savefig('non_chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
#
# plt.bar(x, height_chiral, color = 'k', width = 0.1)
# plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# #plt.show()
# plt.savefig('chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
```

Code file: non_collision_massless_algorithm.py

```
import time
from numpy import pi, cos, sin, e, tan, arctan
from clifford.g3 import blades
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from random import uniform, seed, randint
from sympy import LeviCivita as eps
from main import parity, rotate, energy, epsilon, Gram_det_2
from pytest import approx

e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
I = e1^e2^e3

def dot(a,b):
    dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
    return dot

def multivec_to_vec(a):
    return np.array([a[1], a[2], a[3]])

def swap(S,idx_1,idx_2):

    tmp = S[idx_1]
    S[idx_1] = S[idx_2]
    S[idx_2] = tmp

    return S

def sym_2_Gram_det(a,b):
    return Gram_det_2(a,b,a,b)

def logic_statement_true_for_non_chiral(S, E):

```

```
    p = multivec_to_vec(S[0])
    p = np.insert(p, 0, E[0])
    q = multivec_to_vec(S[1])
    q = np.insert(q, 0, E[1])
    a = multivec_to_vec(S[2])
    a = np.insert(a, 0, E[2])
    b = multivec_to_vec(S[3])
    b = np.insert(b, 0, E[3])
    c = multivec_to_vec(S[4])
    c = np.insert(c, 0, E[4])
    d = multivec_to_vec(S[5])
    d = np.insert(d, 0, E[5])
    RF = a + b + c + d

    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)

    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))))

    def case_2_symmetry(a,b,c,d):
        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)))

    case_3 = case_2_symmetry(a, c, b, d)
    case_4 = case_2_symmetry(a, d, c, b)
    case_5 = case_2_symmetry(b, c, a, d)
    case_6 = case_2_symmetry(b, d, a, c)
    case_7 = case_2_symmetry(c, d, a, b)

    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)))

    def case_8_symmetry(a,b,c,d):
        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)))

    case_9 = case_8_symmetry(a,c,b,d)
    case_10 = case_8_symmetry(a,d,b,c)

    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)

def permute_with_idx(M, E, idx_to_permute):

    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]
    same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute]) and idx != idx_to_permute]

    return list(set(same_mass_with_idx) and set(same_energy_with_idx))

def permutation_boolean(M, E, idx_1, idx_2):

    if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
        return True
    else:
        return False

def construct_state():

    rdm = randint(0, 0)

    if rdm == 0:
        mp, mq, ma, mb, mc, md = 0, 0, randint(0, 10), randint(0, 10), randint(0, 10), randint
```

```
(0, 10)
        p = uniform(-10, 10)*e3
        q = uniform(-10, 10)*e3
        a = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        d = - a - b - c

    if rdm == 1:
        # non chiral
        mp, mq, ma, mb, mc, md = 0, 0, randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
        p = uniform(-10, 10)*e3
        q = uniform(-10, 10)*e3
        a = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
        b = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
        c = uniform(-10, 10) * e1 + uniform(-10, 10) * e3
        d = - a - b - c

    if rdm == 2:
        # (ab) non chiral
        mp, mq, ma, mb, mc, md = 0, 0, 1, 1, randint(0, 10), randint(0, 10)
        p = uniform(-10, 10)*e3
        q = uniform(-10, 10)*e3
        a = 4 * e1 + 5 * e2 + 3 * e3
        B = (uniform(-10, 10) * (e1 ^ e2) + uniform(-10, 10) * (e1 ^ e3) + uniform(-10, 10) * (e2 ^ e3)).normal()
        R = e ^ (uniform(0, 2 * pi) * B)
        b = -5 * e1 + 4 * e2 + 3 * e3
        c = a+b
        d = -c

    if rdm == 3:
        # (ab)(cd) non chiral
        mp, mq, ma, mb, mc, md = 0, 0, 1, 1, 2, 2
        p = uniform(-10, 10)*e3
        q = uniform(-10, 10)*e3
        angle_a = uniform(0,pi/5)
        angle_c = uniform(0, pi / 5)
        a = cos(angle_a)*e1 + sin(angle_a)*e2 + uniform(-10, 10)*e3
        b = sin(angle_a)*e1 + cos(angle_a)*e2 + a[3]*e3
        c = -cos(angle_c)*e1 - sin(angle_c)*e2 - uniform(-10, 10)*e3
        d = -sin(angle_c)*e1 - cos(angle_c)*e2 + c[3]*e3

    Ep, Eq, Ea, Eb, Ec, Ed = energy(mp, p), energy(mq, q), energy(ma, a), energy(mb, b), energy(mc, c), energy(md, d)

    M = [mp, mq, ma, mb, mc, md]
    E = [Ep, Eq, Ea, Eb, Ec, Ed]
    S = [p, q, a, b, c, d]

    return S, E, M

def chirality_test():

    chiral_states = []
    non_chiral_states = []

    S, E, M = construct_state()
    p, q, a, b, c, d = S[0], S[1], S[2], S[3], S[4], S[5]

    # These lists hold indices (as they appear in S) of the particles that can be permuted with a,b,c and d respectively
    permute_with_a = permute_with_idx(M, E, 2)
    permute_with_b = permute_with_idx(M, E, 3)
    permute_with_c = permute_with_idx(M, E, 4)
    permute_with_d = permute_with_idx(M, E, 5)

    S_parity = parity(S)  # Perform parity on the set of momenta
```

```python

    flag = False  # The flag is set to true if the state is non-chiral

    R1 = e1*e3
    S1 = rotate(S_parity, R1) # Now p and q are fixed back to their original state

    if (a[1] != 0) or (a[2] != 0): # If a has components in the 1-2 plane

        for idx in permute_with_a + [2]: # For every x that can be permuted with a, map x to a and perform (ax)

            x = S1[idx]
            x_12 = x - x[3]*e3
            a_12 = a - a[3]*e3

            n = a_12 + x_12

            if n == 0:

                R2 = e1*e2

            else:

                R2 = a_12.normal()*n.normal()

            S2 = rotate(S1, R2)
            S3 = swap(S2, 2, idx)

            if S == S3:
                flag = True

            # The degrees of freedom left now that a is fixed to its original state are permutations between b,c,d

            flag_tmp_1 = permutation_boolean(M, E, 3, 4)  # This checks if (bc) is available

            if flag_tmp_1:
                S4 = swap(S3, 3, 4)
                if S == S4:
                    flag = True

            flag_tmp_2 = permutation_boolean(M, E, 3, 5)  # This checks if (bd) is available

            if flag_tmp_2:
                S5 = swap(S3, 3, 5)
                if S == S5:
                    flag = True

            flag_tmp_3 = permutation_boolean(M, E, 4, 5)  # If we have (cd)

            if flag_tmp_3:
                S6 = swap(S3, 4, 5)
                if S == S6:
                    flag = True

            if flag_tmp_1 and flag_tmp_2:  # If we have (bc) and (bd) then we have (bcd)
                # The following achieves b->c->d->b
                S7 = swap(S3, 3, 4)  # (bc)
                S8 = swap(S7, 3, 5)  # (bd), here in the 3 index lies c but we name it b still, notice we use S6
                if S == S8:
                    flag = True

                # The following achieves b->d->c->b
                S9 = swap(S3, 3, 5)  # (bd)
                S10 = swap(S9, 3, 4)  # (bc), here in the 3 index lies d but we name it b still, notice we use S7
                if S == S10:
                    flag = True
```

```
    elif (b[1] != 0) or (b[2] != 0): # To get here we asserted that a is collinear with e3 so fixed by R1

        for idx in permute_with_b + [3]:  # For every x that can be permuted with b, map x to b and perform (bx)

            if idx == 2: # We do not want permutations with a since a is fixed by R1 when collinear with e3
                continue

            x = S1[idx]
            x_12 = x - x[3] * e3
            b_12 = b - b[3] * e3

            n = b_12 + x_12

            if n == 0:

                R2 = e1 * e2

            else:

                R2 = b_12.normal() * n.normal()

            S2 = rotate(S1, R2)
            S3 = swap(S2, 3, idx)

            if S == S3:
                flag = True

            # The degrees of freedom left now that b is fixed to its original state are permutations between c,d

            flag_tmp_1 = permutation_boolean(M, E, 4, 5)  # This checks if (cd) is available

            if flag_tmp_1:
                S4 = swap(S3, 4, 5)
                if S == S4:
                    flag = True

    elif (c[1] != 0) or (c[2] != 0): # To get here we asserted that a,b are collinear with e3 so fixed by R1

        for idx in permute_with_c + [4]:  # For every x that can be permuted with c, map x to c and perform (cx)

            if (idx == 2) or (idx == 3):  # We do not want permutations with a,b since a,b are fixed by R1
                continue

            x = S1[idx]
            x_12 = x - x[3] * e3
            c_12 = c - c[3] * e3

            n = c_12 + x_12

            if n == 0:

                R2 = e1 * e2

            else:

                R2 = c_12.normal() * n.normal()

            S2 = rotate(S1, R2)
            S3 = swap(S2, 4, idx)

            if S == S3:
```

```python
                flag = True

    if flag:
        non_chiral_states.append([S, E])
    else:
        chiral_states.append([S, E])

    return non_chiral_states, chiral_states

# non_chiral_states_list = []
# chiral_states_list = []
# for iterations in range(1000):
#     S, E, M = construct_state()
#     non_chiral_states, chiral_states = chirality_test()
#     non_chiral_states_list += non_chiral_states
#     chiral_states_list += chiral_states
#
# print(len(non_chiral_states_list), len(chiral_states_list))
#
# non_chiral_evaluation_on_logic_statement = [0, 0]
# for non_chiral_state in non_chiral_states_list:
#     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
#     if flag:
#         non_chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         non_chiral_evaluation_on_logic_statement[1] += 1
#
# chiral_evaluation_on_logic_statement = [0, 0]
# for chiral_state in chiral_states_list:
#     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
#     if flag:
#         chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         chiral_evaluation_on_logic_statement[1] += 1
#
# x = ['True', 'False']
# height_non_chiral = [non_chiral_evaluation_on_logic_statement[0], non_chiral_evaluation_on_logic_statement[1]]
# height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement[1]]
#
# plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
# plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# #plt.show()
# plt.savefig('non_chiral_non_collision_massless_logic_statement_test.pdf', bbox_inches='tight')
#
# plt.bar(x, height_chiral, color = 'k', width = 0.1)
# plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# plt.show()
# plt.savefig('chiral_non_collision_massless_logic_statement_test.pdf', bbox_inches='tight')

print(chirality_test())
```

Code file: non_collision_zeros_algorithm.py

```python
import time
from numpy import pi, cos, sin, e, tan, arctan
from clifford.g3 import blades
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from random import uniform, seed, randint
from sympy import LeviCivita as eps
from main import parity, rotate, energy, epsilon, Gram_det_2
from pytest import approx
```

```python

# Works only for a,b,c,d != 0 (it is very unlikely that any one will be randomly generated in
    momentum 0)

e1, e2, e3 = blades['e1'], blades['e2'], blades['e3']
I = e1^e2^e3

def dot(a,b):
    dot = a[1]*b[1] + a[2]*b[2] + a[3]*b[3]
    return dot

def multivec_to_vec(a):
    return np.array([a[1], a[2], a[3]])

def swap(S,idx_1,idx_2):

    tmp = S[idx_1]
    S[idx_1] = S[idx_2]
    S[idx_2] = tmp

    return S

def sym_2_Gram_det(a,b):
    return Gram_det_2(a,b,a,b)

def logic_statement_true_for_non_chiral(S, E):

    a = multivec_to_vec(S[0])
    a = np.insert(a, 0, E[0])
    b = multivec_to_vec(S[1])
    b = np.insert(b, 0, E[1])
    c = multivec_to_vec(S[2])
    c = np.insert(c, 0, E[2])
    d = multivec_to_vec(S[3])
    d = np.insert(d, 0, E[3])
    RF = a+b+c+d

    case_1 = (Gram_det_2(a,RF,a,RF) == approx(0))

    case_2 = (dot(a+b+c+d, a+b+c+d) == approx(0))

    case_3 = (epsilon(b,c,d,RF) == approx(0))

    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)))

    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)))

    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)))

    return (case_1 or case_2 or case_3 or case_4 or case_5 or case_6)

def permute_with_idx(M, E, idx_to_permute):

    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]
    same_energy_with_idx = [idx for idx in range(len(E)) if E[idx] == approx(E[idx_to_permute])
    and idx != idx_to_permute]

    return list(set(same_mass_with_idx) and set(same_energy_with_idx))

def permutation_boolean(M, E, idx_1, idx_2):

    if (M[idx_1] == M[idx_2]) and (E[idx_1] == E[idx_2]):
        return True
    else:
        return False
```

```python

def construct_state():

    rdm = randint(0, 2)

    if rdm == 0:
        ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
        a = uniform(-10, 10) * e3
        b = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        c = uniform(-10, 10) * e1 + uniform(-10, 10) * e2 + uniform(-10, 10) * e3
        d = -a-b-c

    if rdm == 1:
        # This is the no permutation non-chiral case
        ma, mb, mc, md = randint(0, 10), randint(0, 10), randint(0, 10), randint(0, 10)
        a = uniform(-10, 10) * e3
        b = e1 + e2 + uniform(-10, 10)*e3
        c = -e1 - e2 + uniform(-10, 10)*e3
        d = -a-b-c

    if rdm == 2:
        # Permute 2 non chiral case
        ma, mb, mc, md = 1, 1, 1, 1
        a = uniform(-10, 10) * e3
        angle = uniform(0, pi / 5)
        b = sin(angle)*e1 + cos(angle)*e2 + uniform(-10, 10)*e3
        c = cos(angle)*e1 + sin(angle)*e2 + b[3]*e3
        d = -a-b-c

    Ea, Eb, Ec, Ed = energy(ma, a), energy(mb, b), energy(mc, c), energy(md, d)

    M = [ma, mb, mc, md]
    E = [Ea, Eb, Ec, Ed]
    S = [a, b, c, d]

    return S, E, M

def chirality_test():

    chiral_states = []
    non_chiral_states = []

    S, E, M = construct_state()
    a, b, c, d = S[0], S[1], S[2], S[3]

    permute_with_b = permute_with_idx(M, E, 1)
    permute_with_c = permute_with_idx(M, E, 2)

    S_parity = parity(S)  # Perform parity on the set of momenta

    flag = False  # The flag is set to true if the state is non-chiral

    R1 = e1 * e3
    S1 = rotate(S_parity, R1)  # Now a is fixed back to their original state

    if (b[1] != 0) or (b[2] != 0): # if b has components in the 1-2 plane

        for idx in permute_with_b + [1]:

            if idx == 0:
                continue

            x = S1[idx]
            x_12 = x - x[3]*e3
            b_12 = b - b[3]*e3

            n = b_12 + x_12

            if n == 0:
```

```python

                R2 = e1*e2

            else:

                R2 = b_12.normal()*n.normal()

            S2 = rotate(S1, R2)
            S3 = swap(S2, 1, idx) # Index 1 corresponds to b

            if S == S3:
                flag = True

            if permutation_boolean(M, E, 2, 3): # If we can permute (cd)
                S4 = swap(S3, 2, 3)
                if S == S4:
                    flag = True

    elif (c[1] != 0) or (c[2] != 0):

        for idx in permute_with_c + [2]:

            if idx == 0 or idx == 1:
                continue

            x = S1[idx]
            x_12 = x - x[3] * e3
            c_12 = c - c[3] * e3

            n = c_12 + x_12

            if n == 0:

                R2 = e1 * e2

            else:

                R2 = c_12.normal() * n.normal()

            S2 = rotate(S1, R2)
            S3 = swap(S2, 2, idx) # Index 2 corresponds to c

            if S == S3:
                flag = True

    if flag:
        non_chiral_states.append([S, E])
    else:
        chiral_states.append([S, E])

    return non_chiral_states, chiral_states

# non_chiral_states_list = []
# chiral_states_list = []
# for iterations in range(1000):
#     S, E, M = construct_state()
#     non_chiral_states, chiral_states = chirality_test()
#     non_chiral_states_list += non_chiral_states
#     chiral_states_list += chiral_states
#
# print(len(non_chiral_states_list), len(chiral_states_list))
#
# non_chiral_evaluation_on_logic_statement = [0, 0]
# for non_chiral_state in non_chiral_states_list:
#     mp, mq = uniform(1, 10), uniform(1, 10)
#     flag = logic_statement_true_for_non_chiral(non_chiral_state[0], non_chiral_state[1])
#     if flag:
#         non_chiral_evaluation_on_logic_statement[0] += 1
#     else:
```

```
#         non_chiral_evaluation_on_logic_statement[1] += 1
#
# chiral_evaluation_on_logic_statement = [0, 0]
# for chiral_state in chiral_states_list:
#     mp, mq = uniform(1, 10), uniform(1, 10)
#     flag = logic_statement_true_for_non_chiral(chiral_state[0], chiral_state[1])
#     if flag:
#         chiral_evaluation_on_logic_statement[0] += 1
#     else:
#         chiral_evaluation_on_logic_statement[1] += 1
#
# x = ['True', 'False']
# height_non_chiral = [non_chiral_evaluation_on_logic_statement[0], non_chiral_evaluation_on_logic_statement[1]]
# height_chiral = [chiral_evaluation_on_logic_statement[0], chiral_evaluation_on_logic_statement[1]]
#
# plt.bar(x, height_non_chiral, color = 'k', width = 0.1)
# plt.title('Non-chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# #plt.show()
# plt.savefig('non_chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')
#
# plt.bar(x, height_chiral, color = 'k', width = 0.1)
# plt.title('Chiral states evaluated on the logic statement\nwhich is true iff the input is non-chiral')
# plt.ylabel('Frequency')
# plt.show()
# plt.savefig('chiral_non_collision_logic_statement_test.pdf', bbox_inches='tight')

print(chirality_test())
```