CLHEP **Vector** Package
and
ZOOM **PhysicsVector** Package
Space and Lorentz Vector and Transformation Package
Formulas and Definitions

Fermilab "ZOOM" Physics Class Library Task Force

Version 2.0, January 2, 2001

The CLHEP Vector package implements 3-vectors, 4-vectors, rotations, Lorentz transfomations and related concepts. This includes all functionallity in the original CLHEP package, and in the ZOOM PhysicsVectors package. The latter is now implemented as wrapper headers, so that classes from the ZOOM package can be used wherever the corresponding CLHEP class are expected.

This document briefly lists the methods available, then presents relevant mathematical definitions.

Contents

1 Available Classes and Methods

In this document we somewhat abbreviate the signatures of methods, since coders will look at the header files anyway. Any arguments specified without an explicit data type are of the scalar type `HepDouble`, which is just `double` on virtually every system. Unless otherwise indicated, parameter passage for scalars is by value, and parameter passage for other types is by constant reference. This document also does not discuss namespace issues.

1.1 `Hep3Vector` Class — Vector of real quantities in 3-space

Throughout this section, arguments named v, v1, v2, etc., are of type `Hep3Vector`.

Constructors and Accessors

- Hep3Vector( )
- Hep3Vector(x, y, z)

The following accessor methods are used to obtain the named coordinate components, as in `double s = v.r();`.

- x( ) y( ) z( )
- r( ) ▷ see eqn. 2
- theta( ) ▷ see eqn. 2
- eta( ) ▷ see eqn. 4
- phi( ) ▷ see eqn. 2, 3
- rho( ) ▷ see eqn. 3
- getX( ) getY( ) getZ( )
- getR( ) getTheta( ) getPhi( )
- getEta( ) getRho( )

The Cartesian components may also be accessed by the index syntax, using either square braces or parentheses. In either case, the meaning of the index is 0-based, and an enum is provided to help clarify this: `Hep3Vector::X=0`, `Hep3Vector::Y=1`, `Hep3Vector::Z=2`.

There is a family of methods of the form `set`*Component*`()` which may be used to set one component in Cartesian or Polar coordinates, keeping the other two constant. Also available is `setCylTheta()`, which modifies θ (the angle against the Z axis) while keeing ϕ and the radial distance from the Z axis ρ constant.

Finally, there is also a family of `set()` methods that may be used to update all of a `Hep3Vector`'s three coordinates at once. These `set()` methods' signatures are in one-to-one correspondence with the non-default constructors listed above.

Operators

For `Hep3Vectors` v, v1, and v2, and for a scalar c, the following arithmetic operations are provided, in each case resulting in a `Hep3Vector`.

- v1 + v2 and v1 − v2
- v ∗ c and c ∗ v
- v / c
- − v

In addition, the following arithmetic modify-assignment operations are provided.

- v1 += v2 and v1 −= v2
- v ∗= c and v /= c

The usual six relational operations (==, !=, <, <=, >, >=) are provided (see eqn. 25). Further, the following member functions are useful to check for equality within a relative tolerance.

- bool isNear(v, epsilon) ▷ see eqn. 8
- Scalar howNear(v) ▷ see eqn. 16, 19, 21
- Scalar deltaR(v) ▷ see eqn. 20

The tolerance epsilon may be omitted. The following class-wide (static) functions are used to obtain and set the default tolerance for nearness.

- HepDouble setTolerance(tol) ▷ see eqn. 15
- HepDouble getTolerance( )

Methods

- ostream & operator<<(ostream & os, v)
- istream & operator>>(istream & is, Hep3Vector & v)

- HepDouble dot(v) ▷ see eqn. 5
- Hep3Vector cross(v) ▷ see eqn. 6
- HepDouble diff2(v) ▷ see eqn. 7
- bool isParallel(v, epsilon) ▷ see eqn. 10
- bool isOrthogonal(v, epsilon) ▷ see eqn. 11
- HepDouble howParallel(v) ▷ see eqn. 17, 22
- HepDouble howOrthogonal(v) ▷ see eqn. 18, 23

- HepDouble mag2( ) ▷ see eqn. 27
- HepDouble mag( ) ▷ see eqn. 26, 31
- HepDouble beta( ) ▷ see eqn. 31
- HepDouble gamma( ) ▷ see eqn. 32
- HepDouble pseudoRapidity( ) ▷ see eqn. 4
- HepDouble coLinearRapidity( ) ▷ see eqn. 34
- Hep3Vector unit( ) ▷ see eqn. 29
- Hep3Vector orthogonal( ) ▷ see eqn. 30

The following methods depend on a reference direction (specified by a `Hep3Vector u`). Signatures omitting the reference direction are also supplied—$\hat{z}$ is implied in these cases and the methods take advantage of the simpler form.

- HepDouble perp(u) ▷ see eqn. 40
- HepDouble perp2(u) ▷ see eqn. 41
- SpaceVector perpPart(u) ▷ see eqn. 42
- SpaceVector project(u) ▷ see eqn. 43
- HepDouble angle(u) ▷ see eqn. 35
- HepDouble theta(u) ▷ see eqn. 35
- HepDouble cosTheta(u) ▷ see eqn. 36
- HepDouble cos2Theta(u) ▷ see eqn. 37
- HepDouble eta(u) ▷ see eqn. 28, 38, 126, 131
- HepDouble polarAngle(v2, u) ▷ see eqn. 44, 46
- HepDouble azimAngle(v2, u) ▷ see eqn. 47, 48
- HepDouble rapidity(u) ▷ see eqn. 33, 39

Rotations

These methods change the vector:

- Hep3Vector & rotateX(delta) ▷ see eqn. 52
- Hep3Vector & rotateY(delta) ▷ see eqn. 51
- Hep3Vector & rotateZ(delta) ▷ see eqn. 50, 49
- Hep3Vector & rotateUZ(Hep3Vector) ▷ see eqn. 55

- Hep3Vector & rotate(axis, delta) ▷ see eqn. 53
- Hep3Vector & rotate(const AxisAngle & ax) ▷ see eqn. 53
- Hep3Vector & rotate(phi, theta, psi) ▷ see eqn. 54
- Hep3Vector & rotate(const EulerAngles & e) ▷ see eqn. 54

The following methods both do $\vec{v} \Longleftarrow Rv$. (Notice the order of multiplication for `v *= R`.)

- Hep3Vector transform (const HepRotation & R) ▷ see eqn. 56
- Hep3Vector operator *= (const HepRotation & R) ▷ see eqn. 56

These global functions do not change the `Hep3Vector` v:

- Hep3Vector rotationXOf(v, delta) ▷ see eqn. 52
- Hep3Vector rotationYOf(v, delta) ▷ see eqn. 51
- Hep3Vector rotationZOf(v, delta) ▷ see eqn. 50, 49

- Hep3Vector rotationOf(v, axis, delta) ▷ see eqn. 53
- Hep3Vector rotationOf(v, const AxisAngle & ax) ▷ see eqn. 53
- Hep3Vector rotationOf(v, phi, theta, psi) ▷ see eqn. 54
- Hep3Vector rotationOf(v, const EulerAngles & e) ▷ see eqn. 54

1.2 `SpaceVector` Class — Derived from `Hep3Vector`

The `SpaceVector` class provides backward compatibility with the original ZOOM PhysicsVectors package. It is publicly derived from `Hep3Vector`. It would simply be a typedef off `Hep3Vector`, in the appropriate namespace, but for one set of features which were felt to be overkill in the CLHEP context. This is the ability to construct a `SpaceVector` providing spherical or cylindrical coordinates.

Associated with these constructors are a set of defined keywords which allow disambiguation of the various forms of constructors. These keywords are `RADIANS`, `DEGREES`, and `ETA`,

Throughout this section, we will illustrate using `RADIANS`, but this may be replaced by `DEGREES` to indicate the corresponding angle's units. For theta, this may also be replaced by `ETA`, to indicate that the pseudorapidity is being supplied.

- SpaceVector( )
- SpaceVector(x, y, z)
- SpaceVector(r, theta, RADIANS, phi, RADIANS) ▷ see eqn. 2
- SpaceVector(rho, phi, RADIANS, z) ▷ see eqn. 3

1.3 UnitVector Class

It was found useful in the ZOOM package to express the concept of a `UnitVector`, that is, a vector known to be inherently of unit length. CLHEP neither has such a class, and it is felt (at this time) that it should not. In the merged package, `UnitVector` is provided as a header file which appears only in the PhysicsVectors area.

Since `UnitVector` depends on no non-header implementation code, issues of where to place the library containing non-CLHEP code do not arise.

Although the `UnitVector` class is not derived from the `SpaceVector` class, all const methods (except for three relativistic kinematic methods which only make sense for a vector of length less than one) are provided for `UnitVector`. Thos non-const methods of `Hep3Vector` which do not risk violating the unit-length property, such as rotation, are also provided. In this section, therefore, only the differences in the classes are described.

Constructors and Accessors

`UnitVector` constructors (and `set()` methods) have the same signatures as do the corresponding `SpaceVector` methods, but normalize before returning. The default `UnitVector` constructor yields $\hat{z}$. In addition, we have the conversion constructor

- UnitVector(Hep3Vector v)

Unlike the case for `Hep3Vector`, `UnitVector` disallows setting a single Cartesian coordinate. Also, of course, modifying the radius of a `UnitVector` is forbidden.

Operators

`UnitVectors` are treated as `Hep3Vectors` for purposes of arithmetic and comparisons, except that unary minus returns a `UnitVector`. Modify-assignment (e.g., +=) is forbidden on `UnitVectors`.

Methods

Most `UnitVector` methods match those of the `Hep3Vector` class; some are modified, however, to take advantage of the r = 1 property. Forbidden methods include `beta()`, `gamma()`, and `rapidity()`.

Rotations

All methods and functions in this category apply equally to `UnitVectors` as they do to `Hep3Vectors`.

1.4 HepLorentzVector Class — Vector of real quantities in 4-space

Throughout this section, arguments named p, p1, p2, etc., are of type `HepLorentzVector`, arguments named v, v1, v2, etc., are of type `Hep3Vector`, and an argument named t will refer to a scalar meant as a time component.

Constructors and Accessors

- HepLorentzVector( )
- HepLorentzVector(x, y, z, t)
- HepLorentzVector(v, t)
- HepLorentzVector(t, v)
- HepLorentzVector(t)
- HepLorentzVector(x, y, z)
- HepLorentzVector(v)

The following accessor methods are used to obtain the named coordinate components, as in `double s = p.t();`.

- x( ) y( ) z( ) t( )
- px( ) py( ) pz( ) e( )
- getX( ) getY( ) getZ( ) getT( )
- v( ) or getV( ) or vect( )

The Cartesian components may also be accessed by the index syntax, using either square braces or parentheses. In either case, the meaning of the index is 0-based, *with the time component last*. An enum is nested in the `HepLorentzVector` class to help clarify this: `X=0`, `Y=1`, `Z=2`, `T=3`.

The spatial components can also be accessed in spherical coordinates:

- HepDouble theta() const;
- HepDouble cosTheta() const;
- HepDouble phi() const;
- HepDouble rho() const;

There is a family of methods of the form `set`*Component*`()` which may be used to set one component in Cartesian or spherical coordinates, or the ρ cylindrical coordinate, keeping the other components in that system constant.

- setX(HepDouble); setPx(HepDouble);
- setY(HepDouble); setPy(HepDouble);
- setZ(HepDouble); setPz(HepDouble);
- setT(HepDouble); setE(HepDouble);
- setTheta(HepDouble); ▷ see eqn. 2
- setPhi(HepDouble); ▷ see eqn. 2
- setRho(HepDouble); setPerp(HepDouble); ▷ see eqn. 3

The entire spatial component can be set at once, keeping the time component constant.

- setVect(HepDouble); setV(HepDouble);

And there is a family of `set()` methods, corresponding to the constructors, that may be used to update all four of a `HepLorentzVector`'s coordinates at once.

- set (x, y, z, t);
- set (v, t);
- set (t, x);

Assignment from a `Hep3Vector` is supported, as in `p = v;`.

Finally, there are conversion operators to const and non-const Hep3Vector; these were present in the original CLHEP classes. They ignore the time component.

Operators

For `HepLorentzVectors` p, p1, and p2, and for a scalar c, the following arithmetic operations are provided, in each case resulting in a `HepLorentzVector`.

- p1 + p2 and p1 − p2
- p ∗ c and c ∗ p
- p / c
- − p

In addition, the following arithmetic modify-assignment operations are provided.

- p1 += p2 and p1 −= p2
- p ∗= c and p /= c

The usual six relational operations (==, !=, <, <=, >, >=) are provided (see eqn. 112). Further, the following member functions are useful to check for equality, etc., within a relative tolerance.

- bool isNear(p, epsilon) ▷ see eqn. 106
- HepDouble howNear(p) ▷ see eqn. 118, 122
- bool isNearCM(p, epsilon) ▷ see eqn. 110, 111
- HepDouble howNearCM(p) ▷ see eqn. 119, 123
- bool isParallel(p, epsilon) ▷ see eqn. 113, 114
- HepDouble howParallel(p) ▷ see eqn. 120, 124
- HepDouble deltaR(p) ▷ see eqn. 20

The tolerance epsilon may be omitted. The following class-wide (static) functions are used to obtain and set the default tolerance for nearness.

- HepDouble setTolerance(HepDouble tol)
- HepDouble getTolerance( )

Methods

Establishing a metric convention (static):

- ZMpvMetric_t setMetric(ZMpvMetric_t m)
- ZMpvMetric_t getMetric( )

Metric-independent properties and methods of this 4-vector (the results of these methods do not change sign if you go from the TimePositive to the TimeNegative metric):

- ostream & operator<<(ostream & os, p)
- istream & operator>>(istream & is, LorentzVector & p)
- bool isSpacelike( ) ▷ see eqn. 115
- bool isTimelike( ) ▷ see eqn. 116
- bool isLightlike(epsilon) ▷ see eqn. 117
- HepDouble howLightlike( ) ▷ see eqn. 121, 125
- HepDouble plus( ) ▷ see eqn. 65
- HepDouble minus( ) ▷ see eqn. 66
- HepDouble euclideanNorm2( ) ▷ see eqn. 62

- HepDouble euclideanNorm( ) ▷ see eqn. 63
- HepDouble restMass2( ) ▷ see eqn. 69
- HepDouble restMass( ) ▷ see eqn. 70
- HepDouble m2( ) ▷ see eqn. 88
- HepDouble invariantMass2( ) ▷ see eqn. 88
- HepDouble m( ) mag( ) ▷ see eqn. 61
- HepDouble invariantMass( ) ▷ see eqn. 89
- HepDouble mt2( ) ▷ see eqn. 80
- HepDouble mt( ) ▷ see eqn. 81
- HepDouble et2( ) ▷ see eqn. 82
- HepDouble et( ) ▷ see eqn. 83
- LorentzVector rest4Vector( ) ▷ see eqn. 71
- SpaceVector boostVector( ) ▷ see eqn. 72
- HepDouble beta( ) ▷ see eqn. 73
- HepDouble gamma( ) ▷ see eqn. 74
- HepDouble eta( ) ▷ see eqn. 75, 76, 127
- HepDouble rapidity( ) ▷ see eqn. 77, 78
- HepDouble coLinearRapidity( ) ▷ see eqn. 79, 135
- SpaceVector findBoostToCM( ) ▷ see eqn. 90

Metric-dependent properties of this 4-vector: (the results of these methods change sign if you go from the TimePositive to the TimeNegative metric):

- HepDouble mag2( ) ▷ see eqn. 59

Metric-independent methods combining two 4-vectors:

- HepDouble delta2Euclidean(p) ▷ see eqn. 64
- HepDouble plus(p) ▷ see eqn. 67
- HepDouble minus(p) ▷ see eqn. 68
- HepDouble eta(p) ▷ see eqn. 75, 76, 127, 130
- HepDouble rapidity(p) ▷ see eqn. 77, 78, 128, 132
- HepDouble invariantMass2(p) ▷ see eqn. 88
- HepDouble invariantMass(p) ▷ see eqn. 89
- SpaceVector findBoostToCM(p) ▷ see eqn. 90

Metric-dependent methods combining two 4-vectors:

- HepDouble dot(p) ▷ see eqn. 57, 58
- HepDouble operator*(p) ▷ see eqn. 57, 58
- HepDouble diff2(p) ▷ see eqn. 60

Methods involving properties of the spatial part of the 4-vector (these could be invoked as p.v().whatever() but the p.whatever() syntax is shorter). Most of these take a reference direction v; if v is omitted, $\hat{z}$ is implied and the methods take advantage of the simpler form.

- HepDouble perp(v) ▷ see eqn. 40
- HepDouble perp2(v) ▷ see eqn. 41
- HepDouble angle( ) ▷ see eqn. 35
- setVectM(HepDouble); setVectMag(HepDouble); ▷ see eqn. 2
- setRho(HepDouble); setPerp(HepDouble); ▷ see eqn. 3
- HepDouble pseudoRapidity( ) ▷ see eqn. 4

Rotations and Boosts

These methods change the 4-vector.

- LorentzVector & rotateX(delta) ▷ see eqn. 52
- LorentzVector & rotateY(delta) ▷ see eqn. 51
- LorentzVector & rotateZ(delta) ▷ see eqn. 50
- LorentzVector & rotate(v, delta) ▷ see eqn. 53
- LorentzVector & rotate(delta, v) ▷ see eqn. 53
- LorentzVector & rotate(phi, theta, psi) ▷ see eqn. 54
- LorentzVector & rotate(EulerAngles & e) ▷ see eqn. 54
- LorentzVector & rotateUz(v) ▷ see eqn. 55
- LorentzVector & boostX(beta) ▷ see eqn. 100
- LorentzVector & boostY(beta) ▷ see eqn. 101
- LorentzVector & boostZ(beta) ▷ see eqn. 102
- LorentzVector & boost(v) ▷ see eqn. 104
- LorentzVector & boost(v, beta) ▷ see eqn. 103

The following methods all do $\vec{p} \Longleftarrow Rp$ where R is either a `HepRotation`or a `HepLorentzRotation`. (Notice the order of multiplication for `p *= R`.)

- HepLorentzVector transform (const HepRotation & R) ▷ see eqn. 105
- HepLorentzVector operator *= (const HepRotation & R) ▷ see eqn. 105
- HepLorentzVector transform (const HepLorentzRotation & R) ▷ see eqn. 105
- HepLorentzVector operator *= (const HepLorentzRotation & R) ▷ see eqn. 105

These functions return a new 4-vector.

- LorentzVector rotationXOf(p, delta) ▷ see eqn. 52
- LorentzVector rotationYOf(p, delta) ▷ see eqn. 51
- LorentzVector rotationZOf(p, delta) ▷ see eqn. 50
- LorentzVector rotationOf(p, v, delta) ▷ see eqn. 53
- LorentzVector rotationOf(p, phi, theta, psi) ▷ see eqn. 54
- LorentzVector rotationOf(p, const EulerAngles & e) ▷ see eqn. 54
- LorentzVector boostXOf(p, beta) ▷ see eqn. 100
- LorentzVector boostYOf(p, beta) ▷ see eqn. 101
- LorentzVector boostZOf(p, beta) ▷ see eqn. 102
- LorentzVector boostOf(p, betaVector) ▷ see eqn. 104
- LorentzVector boostOf(p, v, beta) ▷ see eqn. 103

1.5 `LorentzVector` Class — Typedefed from `HepLorentzVector`

The `LorentzVector` class provides backward compatibility with the original ZOOM PhysicsVectors package. It is simply a typedef off `HepLorentzVector`, in the appropriate namespace. This is because there were no features or constructors in the ZOOM product which were felt to be overkill in the CLHEP context.

1.6 `Hep2Vector` Class

The `Hep2Vector` class is a simple plane vector. Throughout this section, arguments named s, s1, s2, etc., are of type `Hep2Vector`.

Constructors and Accessors

- Hep2Vector( )
- Hep2Vector(x, y)
- Hep2Vector(Hep3Vector v)

That last constructor will supress the Z component of $\vec{v}$.

Cartesian and polar coordinates may be accessed; these are identical to those of a `Hep3Vector`, with the Z component fixed at zero.

- x( ) y( )
- r( ) ▷ see eqn. 2
- phi( ) ▷ see eqn. 2

The Cartesian components may also be accessed by the index syntax, using either square braces or parentheses. In either case, the meaning of the index is 0-based, and an enum is provided to help clarify this: `Hep2Vector::X=0`, `Hep2Vector::Y=1`.

There is a family of methods of the form `set`*Component*`()` which may be used to set one component in Cartesian or Polar coordinates, keeping the other component constant.

- setX(x) setY(y)
- setR(r) setMag(r) ▷ see eqn. 2
- setPhi(phi) ▷ see eqn. 2

Finally, there is also a family of `set()` methods that may be used to update all of a `Hep3Vector`'s three coordinates at once. These `set()` methods' signatures are in one-to-one correspondence with the non-default constructors listed above.

Operators

For `Hep2Vectors` s, s1, and s2, and for a scalar c, the following arithmetic operations are provided, in each case resulting in a `Hep2Vector`.

- s1 + s2 and s1 − s2
- s ∗ c and c ∗ s
- s / c
- − s

In addition, the following arithmetic modify-assignment operations are provided.

- s1 += s2 and v1 −= s2
- s ∗= c and v /= c

The usual six relational operations (==, !=, <, <=, >, >=) are provided (see eqn. 25). Further, the following member functions are useful to check for equality within a relative tolerance. (Again, these match the corresponding methods for `Hep3Vector`, with Z-component pinned at zero.)

- bool isNear(v, epsilon) ▷ see eqn. 8
- Scalar howNear(v) ▷ see eqn. 16, 19, 21
- Scalar deltaR(v) ▷ see eqn. 20

The tolerance epsilon may be omitted. The following class-wide (static) functions are used to obtain and set the default tolerance for nearness.

- HepDouble setTolerance(tol) ▷ see eqn. 15
- HepDouble getTolerance( )

Methods

The set of methods is somewhat simpler that for `Hep3Vector`, but in all applicable cases the definitions match those for `Hep3Vector`:

- ostream & operator<<(ostream & os, s)

- HepDouble dot(v) ▷ see eqn. 5
- bool isParallel(v, epsilon) ▷ see eqn. 10
- bool isOrthogonal(v, epsilon) ▷ see eqn. 11
- HepDouble howParallel(v) ▷ see eqn. 17, 22
- HepDouble howOrthogonal(v) ▷ see eqn. 18, 23
- HepDouble angle(s2) ▷ see eqn. 35

- HepDouble mag2( ) ▷ see eqn. 27
- HepDouble mag( ) ▷ see eqn. 26, 31
- Hep2Vector unit( ) ▷ see eqn. 29

There is one method which is applicable for `Hep2Vector` but which does not match the definition for `Hep3Vector`:

- Hep2Vector orthogonal( )

`s.orthogonal()` is s rotated clockwise by 90° if $|s_x| < |s_y|$, and counterclockwise by 90° if $|s_x| \geq |s_y|$, and counterclockwise.

In the ZOOM area, the class `PlaneVector` is typedefed to `Hep2Vector`.

1.7 `HepRotation` Classes

Throughout this section, arguments named r, r1, r2, etc., are of type `HepRotation`. Also, rowX, rowY, rowZ, colX, colY, colZ will represent `Hep3Vector` arguments.

Constructors and Accessors

- Rotation( )
- Rotation(r)
- Rotation & operator=(r)
- Rotation & set(r)
- Rotation(phi, theta, psi) ▷ see eqn. 54
- Rotation & set(phi, theta, psi) ▷ see eqn. 54
- Rotation(const EulerAngles & e) ▷ see eqn. 54
- Rotation & set(const EulerAngles & e) ▷ see eqn. 54
- Rotation(const Hep3Vector & axis, delta) ▷ see eqn. 53
- Rotation & set(const Hep3Vector & axis, delta) ▷ see eqn. 53

- Rotation(const AxisAngle & ax) ▷ see eqn. 53
- Rotation & set(const AxisAngle & ax) ▷ see eqn. 53
- Rotation(colX, colY, colZ)
- Rotation & set(colX, colY, colZ)
- Rotation & setRows(rowX, rowY, rowZ)
- Rotation(const ZMpvRep3x3 & rep)
- Rotation & set(const ZMpvRep3x3 & rep)

- phi( ) theta( ) psi( ) ▷ see eqn. 54,146,145
- eulerAngles( ) ▷ see eqn. 54,146,145
- axis( ) delta( )
- axisAngle( ) ▷ see eqn. 53,145
- colX( ) colY( ) colZ( )
- rowX( ) rowY( ) rowZ( )
- xx( ) xy( ) xz( )
- yx( ) yy( ) yz( )
- zx( ) zy( ) zz( )
- HepRep3x3 rep3x3( )
- HepRep4x4 rep4x4( )

And the matrix elements of a `HepRotation` may be accessed using two integer indices in parentheses or square brackets, with indices running from 0 to 2.

The following methods alter one component of a `HepRotation`, in some way of viewing that rotation:

- setPhi(HepDouble) setTheta(HepDouble) setPsi(HepDouble) ▷ see eqn. 137
- setAxis(Hep3Vector) setDelta(Hep3Vector) ▷ see eqn. 136

Use of `HepRotation` as a 4-Rotation

The `HepRotation` may be considered to be a 4-rotation. Thus, other accessors applicable to `HepLorentzRotation` (see section (1.9)) may be applied to `HepRotation` as well.

Operators and Methods

- Rotation & operator*=(r)
- friend Rotation operator*(r1, r2)

- static HepDouble getTolerance( ) ▷ see eqn. 157
- static HepDouble setTolerance(tol)
- bool isNear(r, epsilon) ▷ see eqn. 151
- HepDouble howNear(r) ▷ see eqn. 151
- HepDouble distance2(r) ▷ see eqn. 152

- r1 == r2 r1 != r2
- r1 > r2 r1 >= r2 r1 < r2 r1 <= r2 ▷ see eqn. 159

- Hep3Vector operator( ) (const Hep3Vector & v) ▷ see eqn. 138
- Hep3Vector operator* (const Hep3Vector & v) ▷ see eqn. 138
- LorentzVector operator( ) (const LorentzVector & p) ▷ see eqn. 138
- LorentzVector operator* (const LorentzVector & p) ▷ see eqn. 138

- HepDouble norm2( ) ▷ see eqn. 154
- HepDouble isIdentity( )
- ostream & print (ostream & os)
- ostream & operator<<(ostream & os, const RotationInterface & r)
- rectify( ) ▷ see eqn. 168

The Rotation Group

The following methods use `HepRotation`s as a group, that is, they invert, multiply, and so forth:

- Hep3Rotation operator* (const Hep3Rotation & r) ▷ see eqn. 162
- operator*= (const Hep3Rotation & r) ▷ see eqn. 163
- transform (const Hep3Rotation & r) ▷ see eqn. 165
- RotateX (delta) RotateY (delta) RotateZ (delta) ▷ see eqn. 166
- RotateAxes (newX, newY, newZ) ▷ see eqn. 167
- invert( )
- HepRotation inverse( )
- HepRotation inverseOf(r)

Axial Rotations

There are threee specialized rotation classes, `HepRotationX`, `HepRotationZ`, and `HepRotationZ`. These use substantially less storage than the general `HepRotation`, and for some methods it is quicker to work with a specialized axial rotation rather than the general case.

All information which can be obtained from a general `HepRotation` can be obtained from any of these specialized axial rotations. However, the set of methods which modify the rotations are very restricted for the specialized cases:

- RotationX( )
- RotationX(delta) ▷ see eqn. 52
- RotationX & set(delta) ▷ see eqn. 52
- RotationY( )
- RotationY(delta) ▷ see eqn. 51
- RotationY & set(delta) ▷ see eqn. 51
- RotationZ( )
- RotationZ(delta) ▷ see eqn. 50
- RotationZ & set(delta) ▷ see eqn. 50

1.8 `Rotation` Class — Derived from `HepRotation`

The `Rotation` classes (which also include `RotationX`, `RotationY`, and `RotationZ`) provide backward compatibility with the original ZOOM PhysicsVectors package. These are simply typedefs the corresponding CLHEP classes, in the appropriate namespace. This is because there were no features or constructors in the ZOOM product which were felt to be overkill in the CLHEP context.

The `Rotation.h` header in the ZOOM area also defines `ZMpvRep3x3`, `ZMpvRep3x3`, and `ZMpvRep3x3` which are typedefs for the corresponding CLHEP structs.

1.9 `HepLorentzRotation` Classes

There are three sorts of Lorentz transformation objects suppoprted: General `HepLorentzRotations`, pure Lorentz boosts `HepBoost`, and pure boosts along axes `HepBoostX`, `HepBoostY`, `HepBoostZ`.

(Technically, ordinary `HepRotations` and axial rotations can also be considered as Lorentz transformations, and indeed one of those classes can be used wherever a general Lorentz transformation is called for. The class `Hep4RotationInterface` represents the abstract concept. But the rotation classes have been defined above, so we will restrict this section to discussing transformations involving the time component.)

Throughout this section, arguments named lt, lt1, lt2, etc., are of type `HepLorentzRotation`, arguments named b, b1, b2, etc., are of type `HepBoost`, and arguments named r, r1, r2, etc., are of type `HepRotation`. Arguments named R can be any sort of `Hep4RotationInterface` including `HepLorentzRotations`, boosts, rotations, and axial boosts and rotations. Also, row1, row2, row3, row4, col1, col2, col3, col4 will represent `HepLorentzVector` arguments.

Constructors and Accessors—HepLorentzRotation

- HepLorentzRotation( )
- HepLorentzRotation (lt)
- HepLorentzRotation (r)
- HepLorentzRotation (R)
- HepLorentzRotation & operator=(lt)
- HepLorentzRotation & operator=(r)
- HepLorentzRotation & operator=(R)
- HepLorentzRotation & set(lt)
- HepLorentzRotation & set(r)
- HepLorentzRotation & set(R)
- HepLorentzRotation (Hep3Vector boostVector)
- HepLorentzRotation (HepBoost boost)
- HepLorentzRotation (boostX, boostY, boostZ)
- HepLorentzRotation & set (Hep3Vector boostVector)
- HepLorentzRotation & set (HepBoost boost)
- HepLorentzRotation & set (boostX, boostY, boostZ)
- HepLorentzRotation (b, r) ▷ see eqn. 177
- HepLorentzRotation & set(b, r)
- HepLorentzRotation (r, b) ▷ see eqn. 178
- HepLorentzRotation & set(b, r)
- HepLorentzRotation(col1, col2, col3, col4)
- HepLorentzRotation & set(col1, col2, col3, col4)
- HepLorentzRotation & setRows(row1, row2, row3, row4)
- HepLorentzRotation(const HepRep4x4 & rep)
- HepLorentzRotation & set(const HepRep4x4 & rep)
- print (ostream & os)

Constructors and Accessors—HepBoost

- HepBoost( )
- HepBoost(b)
- HepBoost & set(b)
- HepBoost (const Hep3Vector & direction, beta) ▷ see eqn. 170
- HepBoost & set(const Hep3Vector & direction, beta)

- HepBoost (const Hep3Vector & betaVector) ▷ see eqn. 170
- HepBoost & set(const Hep3Vector & betaVector)
- HepBoost (betaX, betaY, betaZ) ▷ see eqn. 170
- HepBoost & set(betaX, betaY, betaZ)
- HepBoost(const HepRep4x4Symmetric & rep)
- HepBoost & set(const HepRep4x4Symmetric & rep)

Constructors and Accessors—Axial Boosts

- HepBoostX( )
- HepBoostX(beta) ▷ see eqn. 169
- HepBoostX & set(beta)
- HepBoostY( )
- HepBoostY(beta)
- HepBoostY & set(beta)
- HepBoostZ( )
- HepBoostZ(beta)
- HepBoostZ & set(beta)

Components and Decomposition

- decompose(HepBoost & b, HepRotation & r) ▷ see eqn. 177
- decompose(HepRotation & r, HepBoost & b) ▷ see eqn. 178
- col1( ) col2( ) col3( ) col4( )
- row1( ) row2( ) row3( ) row4( )
- xx( ) xy( ) xz( ) xt( )
- yx( ) yy( ) yz( ) yt( )
- zx( ) zy( ) zz( ) zt( )
- tx( ) ty( ) tz( ) tt( )
- HepRep4x4 rep4x4( )

- ostream & operator<<(ostream & os, const Hep4RotationInterface & lt)

Also, components can be accessed by C-style and array-style subscripting:

- lt[i][j] ▷ see eqn. 177
- lt(i,j) ▷ see eqn. 177

The following are applicable to `HepBoost` but not to `HepLorentzRotation`:

- Hep3vector direction( ) beta( ) gamma( )
- Hep3Vector boostVector( )
- HepRep4x4Symmetric rep4x4Symmetric( )

Application to 4-vectors

- LorentzVector operator* (const LorentzVector & w)
- LorentzVector operator( ) (const LorentzVector & w)

Comparisons and Nearness

- lt1 == lt2 lt1 != lt2 ▷ see eqn. 183

- lt1 $>$ lt2 lt1 $>=$ lt2 lt1 $<$ lt2 lt1 $<=$ lt2 ▷ see eqn. 181
- isIdentity()

- getTolerance( ) ▷ see eqn. 157
- setTolerance(tol)
- int compare(lt) ▷ see eqn. 181
- bool isNear(lt, epsilon) ▷ see eqn. 171, 179
- double distance2(lt) ▷ see eqn. 172, 179
- double howNear(lt) ▷ see eqn. 172, 179
- double norm2() ▷ see eqn. 174, 180

Arithmetic in the Lorentz Group

- lt1 * lt2
- HepLorentzRotation & operator*=(lt)
- HepLorentzRotation & operator*=(b)
- HepLorentzRotation & operator*=(r)
- HepLorentzRotation & transform (lt) ▷ see eqn. 186

- invert( )
- HepLorentzRotation inverseOf(lt)

- HepLorentzRotation & rotate (delta, axis) ▷ see eqn. 187
- HepLorentzRotation & rotateX (delta)
- HepLorentzRotation & rotateY (delta)
- HepLorentzRotation & rotateZ (delta)
- HepLorentzRotation & boost (betaX, betaY, betaZ) ▷ see eqn. 188
- HepLorentzRotation & boost (v) ▷ see eqn. 188
- HepLorentzRotation & boostX (beta)
- HepLorentzRotation & boostY (beta)
- HepLorentzRotation & boostZ (beta)

- rectify( ) ▷ see eqn. 189

Arithmetic on Boosts

The pure boosts do not form a group, since the product of two boosts is a Lorentz transformation which in general involves a rotation. The pure boosts along an axial direction do form groups.

- LorentzBoost inverseOf(b)
- LorentzBoostX inverseOf(LorentzBoostX bx)
- LorentzBoostY inverseOf(LorentzBoostY by)
- LorentzBoostZ inverseOf(LorentzBoostZ bz)

- bx = bx1 * bx2
- by = by1 * by2
- bz = bz1 * bz2

1.10 LorentzTransformation Class — Derived from HepLorentzRotation

The HepLorentzRotation classes provides backward compatibility with the original ZOOM PhysicsVectors package. It is defined in LorentzTransformation.h in the ZOOM area, and is a typedef for HepLorentzRotation, in the appropriate namespace. This is because there were no features or constructors in the ZOOM product which were felt to be overkill in the CLHEP context.

Similarly, that files establishes typedefs LorentzBoost for HepBoost, LorentzBoostX for HepBoostX, LorentzBoostY for HepBoostY, and LorentzBoostZ for HepBoostZ.

Also, for any ZOOM users using LorentzTransformationInterface, this is typedefed in LorentzTransformation.h as Hep4RotationInterface.

2 Hep3Vector and SpaceVector Classes

Hep3Vectors may be expressed as Cartesian coordinates, Spherical coordinates, or Cylindrical coordinates.

$$(x, y, z) \tag{1}$$

$$(r, \theta, \phi)$$

$$\begin{cases} x = r\sin\theta\cos\phi \\ y = r\sin\theta\sin\phi \\ z = r\cos\theta \end{cases} \tag{2}$$

$$(\rho, \phi, z)$$

$$\begin{cases} x = \rho\cos\phi \\ y = \rho\sin\phi \\ z = z \end{cases} \tag{3}$$

For Spherical coordinates one may optionally specify the pseudorapidity η instead of θ.

$$\left(\rho, \phi, \eta = -\ln\tan\frac{\theta}{2}\right) \tag{4}$$

2.1 Dot and Cross Products

Let $\vec{v}_1$ and $\vec{v}_2$ be Hep3Vectors. Then:

$$\vec{v}_1.\mathrm{dot}(\vec{v}_2) \equiv \vec{v}_1 \cdot \vec{v}_2 = \sum_i \vec{v}_{1i}\vec{v}_{2i} \tag{5}$$

$$\vec{v}_1.\mathrm{cross}(\vec{v}_2) \equiv \vec{v}_1 \times \vec{v}_2 = \overrightarrow{\left(\sum_{jk} \epsilon_{ijk}\vec{v}_{1j}\vec{v}_{2k}\right)} \tag{6}$$

$$\vec{v}_1.\mathrm{diff2}(\vec{v}_2) = |\vec{v}_1 - \vec{v}_2|^2 \tag{7}$$

Here, ϵ_{ijk} is the three-index anti-symmetric symbol.

2.2 Near Equality and isOrthogonal/isParallel

We structure the definitions of near equality and orthogonal/parallel such that they are commutative (order of vectors makes no difference), rotationally invariant, and scale invariant (multiplying both vectors by the same non-zero constant makes no difference). They also match the definitions of relative equality (or perpendicularity or parallelism) within ϵ for the case where each vector is along or near an axis. This fixes the definitions, up to order ϵ.

Let the method `isNear()` be represented by the symbol $\approx$, and let $\vec{v}_1, \vec{v}_2$ be Hep3Vectors. Then:

$$\vec{v}_1 \approx \vec{v}_2 \text{ if } |\vec{v}_1 - \vec{v}_2|^2 \leq \epsilon^2 \vec{v}_1 \cdot \vec{v}_2 \tag{8}$$

However, if $\vec{v}_1$ and/or $\vec{v}_2$ is known to be a UnitVector (designated as $\hat{u}_1$), this sets a meaningful scale—if the vectors are nearly equal, they are both of magnitude near unity. In that case a simpler absolute formula, which is which is equivalent to order ϵ to the relative criterion 8, is used:

$$\hat{u}_1 \approx \vec{v}_2 \text{ if } |\hat{u}_1 - \vec{v}_2|^2 \leq \epsilon^2 \tag{9}$$

Tests for `v1.isParallel(v2)` and `v1.isOrthogonal(v2)` utilize the dot and cross products:

$$\vec{v}_1 \parallel \vec{v}_2 \text{ if } \left|\frac{\vec{v}_1 \times \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_2}\right|^2 \leq \epsilon^2 \tag{10}$$

$$\vec{v}_1 \perp \vec{v}_2 \text{ if } \left|\frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \times \vec{v}_2}\right|^2 \leq \epsilon^2 \tag{11}$$

Care is taken to avoid taking products of more than two potentially large quantities in evaluating these. Thus these tests can be done on any vectors which could safely be squared. ϵ is assumed to be small; in some cases, short cuts are taken to determine the result without potentially generating large quantities. These techniques are not necessarily faithful to the above formulae when $\epsilon \geq 1$.
The definition of `v1.isParallel(v2)` is equivalent, to order ϵ, to a simple (but computationally more expensive) definition involving normalizing the vectors:

$$\vec{v}_1 \parallel \vec{v}_2 \text{ if } |\hat{v}_1 - \hat{v}_2|^2 \leq \epsilon^2$$

If one of the vectors is the zero vector, the above relations may be ambiguous instead the following hold: ($\vec{v}$ here is any non-zero vector):

$$\vec{v} \approx \vec{0} \text{ iff } \vec{v} = \vec{0} \tag{12}$$

$$\vec{v} \perp \vec{0} \text{ for all } \vec{v} \tag{13}$$

$$\vec{v} \parallel \vec{0} \text{ iff } \vec{v} = \vec{0} \tag{14}$$

The default tolerance (which may be modified by the class static method `setTolerance()`) for vectors and Lorentz vectors is 100 times the double precision epsilon.

$$\epsilon_{\text{default}} \approx 2.2 \cdot 10^{-14} \tag{15}$$

2.3 Measures of Near-ness

Each boolean tolerance comparison method is accompanied by a method returning the measure used to compare to the tolerance ϵ. The formulae can be deduced from those above:

$$\vec{v}_1.\text{howNear}(\vec{v}_2) = \max\left(\sqrt{\frac{|\vec{v}_1 - \vec{v}_2|^2}{\vec{v}_1 \cdot \vec{v}_2}}, 1\right) \tag{16}$$

$$\vec{v}_1.\text{howParallel}(\vec{v}_2) = \max\left(\left|\frac{\vec{v}_1 \times \vec{v}_2}{\vec{v}_1 \cdot \vec{v}_2}\right|, 1\right) \tag{17}$$

$$\vec{v}_1.\text{howOrthogonal}(\vec{v}_2) = \max\left(\left|\frac{\vec{v}_1 \cdot \vec{v}_2}{\vec{v}_1 \times \vec{v}_2}\right|, 1\right) \tag{18}$$

The above relative measures are limited to a maximum of 1; cases where the denominator would be zero may safely be done without overflow. (Of course, `howParallel()` and `howOrthogonal()` are close to zero for nearly parallel and perpendicular vectors, respectively).

Note that since a UnitVector has a natural absolute scale, the `Unitvector::howNear()` method is absolute, not relative. When a comparison method involves absolute tolerance, the measure returned is not truncated at 1:

$$\hat{u}_1.\text{howNear}(\vec{v}_2) = \vec{v}_2.\text{howNear}(\hat{u}_1) = |\hat{u}_1 - \vec{v}_2| \tag{19}$$

In addition, we provide `deltaR()`, a measure of nearness useful in collider physics analysis. It is defined by

$$\vec{v}_1\text{deltaR}(\vec{v}_2) = \sqrt{(\Delta\phi)^2 + (\Delta\eta)^2)} = \sqrt{(\vec{v}_1.\text{phi} - \vec{v}_2.\text{phi})^2 + (\vec{v}_1.\text{eta} - \vec{v}_1.\text{eta})^2} \tag{20}$$

where of course the angular ϕ difference is corrected to lie in the range $(-\pi, \pi]$.

When one vector or the other is zero,

$$\vec{v}.\text{howNear}(\vec{0}) = \vec{0}.\text{howNear}(\vec{v}) = 1 \tag{21}$$
$$\vec{v}.\text{howParallel}(\vec{0}) = \vec{0}.\text{howParallel}(\vec{v}) = 1 \tag{22}$$
$$\vec{v}.\text{howOrthogonal}(\vec{0}) = \vec{0}.\text{howOrthogonal}(\vec{v}) = 0 \tag{23}$$
$$\vec{v}.\text{deltaR}(\vec{0}) = \vec{0}.\text{deltaR}(\vec{v}) = |\Delta\eta| = |\vec{v}.\text{eta}()| \tag{24}$$

When both vectors are zero, all these measures will return zero.

Ordering Comparisons for `Hep3Vectors`

The comparison operators `( >, >=, <, <= )` for `Hep3Vector` (and for `UnitVector`) use a "dictionary ordering", comparing first the Z, then the Y, then the X components:

$$\begin{aligned} \vec{v}_1 > \vec{v}_2 \text{ if} \\ z_1 > z_2 \text{ or} \\ [z_1 = z_2 \text{ and } y_1 > y_2] \text{ or} \\ [z_1 = z_2 \text{ and } y_1 = y_2 \text{ and } x_1 > x_2] \end{aligned} \tag{25}$$

2.4 Intrinsic Properties and Relativistic Quantities

A `Hep3Vector` does not have much in the way of intrinsic properties—just its magnitude. We can also talk about the pseudorapidity, which depends only on the angle against the Z axis.

$$\vec{v1}.\text{mag}() = \sqrt{\vec{v}_1 \cdot \vec{v}_1} \tag{26}$$
$$\vec{v1}.\text{mag2}() = \vec{v}_1 \cdot \vec{v}_1 \tag{27}$$
$$\vec{v1}.\text{eta}() = -\ln\tan\frac{\theta_{\vec{v}_1,\hat{z}}}{2} = -\ln\tan\frac{\hat{v}_1 \cdot \hat{z}}{2} \tag{28}$$

Another intrinsic is the unit vector in the direction of $\vec{v}$:

$$\vec{v}.\text{unit}() \;=\; \frac{\vec{v}}{|v|} \tag{29}$$

A somewhat contrived intrinsic property, useful for Geant4, is a special vector orthognal to $\vec{v}$, lying in a plane defined by two coordinate axes. The plane is chosen so as to supress the smallest component of $\vec{v}$.

$$\begin{aligned}
\min(v_x, v_y, v_z) = v_z \Longrightarrow \vec{v}.\text{orthogonal}() \;&= \{v_y, -v_x, 0\} \\
\min(v_x, v_y, v_z) = v_y \Longrightarrow \vec{v}.\text{orthogonal}() \;&= \{-v_z, 0, v_x\} \\
\min(v_x, v_y, v_z) = v_z \Longrightarrow \vec{v}.\text{orthogonal}() \;&= \{0, v_z, -v_y\}
\end{aligned} \tag{30}$$

(In the case of equal components, v_z is considered smallest, then v_y.)
A `Hep3Vector` of length less than 1 may be considered as defining a Lorentz boost; in that sense, we can discuss β and γ of the `Hep3Vector`, and the rapidity associated with that boost.

$$\vec{v1}.\text{beta}() \;= \vec{v1}\ .\text{mag}() \;= \beta \tag{31}$$

$$\vec{v1}.\text{gamma}() \;= \frac{1}{\sqrt{1-\beta^2}} = \gamma \tag{32}$$

$$\vec{v1}.\text{rapidity}() \;= \tanh^{-1}(\vec{v}_1 \cdot \hat{z}) \tag{33}$$

$$\vec{v1}.\text{coLinearRapidity}() \;= \tanh^{-1}\beta = \tanh^{-1}(\vec{v}_1 \cdot \hat{v}_1) \tag{34}$$

The rapidity and pseudorapidity are discussed further in section §4.

2.5 Properties Involving Vectors and Directions

Let a reference direction be represented by a unit vector $\hat{u}$ in that direction. In all the following methods, the reference direction may be omitted, defaulting to $\hat{\mathbf{z}}$. And in these methods, a *non-zero* second vector may be substituted for $\hat{u}$; the unit vector in the direction of $\vec{v}_2$ will be used in that case. Then the following definitions of methods relative to the reference direction apply:

$$\vec{v}.\text{angle}(\hat{u}) = \theta_{\vec{v},\hat{u}} = \cos^{-1}\left(\frac{\vec{v}\cdot\hat{u}}{|\vec{v}|}\right) \tag{35}$$

$$\vec{v}.\text{cosTheta}(\hat{u}) = \cos\theta_{\vec{v},\hat{u}} \tag{36}$$

$$\vec{v}.\text{cos2Theta}(\hat{u}) = \cos^2\theta_{\vec{v},\hat{u}} \tag{37}$$

$$\vec{v}.\text{eta}(\hat{u}) = -\ln\tan\frac{\theta_{\vec{v},\hat{u}}}{2} \tag{38}$$

$$\vec{v}.\text{rapidity}(\hat{u}) = \tanh^{-1}(\vec{v}_1 \cdot \hat{u}) \tag{39}$$

$$\vec{v}.\text{perp}(\hat{u}) = |\vec{v} - (\vec{v}\cdot\hat{u})\hat{u}| \tag{40}$$

$$\vec{v}.\text{perp2}(\hat{u}) = |\vec{v} - (\vec{v}\cdot\hat{u})\hat{u}|^2 \tag{41}$$

$$\vec{v}.\text{perpPart}(\hat{u}) = \vec{v} - \hat{u}\cos\theta_{\vec{v},\hat{u}}\,|\vec{v}| = \vec{v} - (\vec{v}\cdot\hat{u})\hat{u} \tag{42}$$

$$\vec{v}.\text{project}(\hat{u}) = \hat{u}\cos\theta_{\vec{v},\hat{u}}\,|\vec{v}| = (\vec{v}\cdot\hat{u})\hat{u} \tag{43}$$

$$\vec{v}.\text{polarAngle}(\vec{v_2}) \equiv \vec{v}.\text{polarAngle}(\vec{v_2}, \hat{z}) = \vec{v_2}.\text{theta}() - \vec{v}.\text{theta}() \tag{44}$$

$$\vec{v}.\text{polarEta}(\vec{v_2}) \equiv \vec{v}.\text{polarEta}(\vec{v_2},\hat{z}) = \vec{v_2}.\text{eta}() - \vec{v}.\text{eta}() \tag{45}$$

$$\vec{v}.\text{polarAngle}(\vec{v_2},\hat{u}) = \vec{v_2}.\text{eta}(\hat{u}) - \vec{v}.\text{eta}(\hat{u}) \tag{46}$$

$$\vec{v}.\text{azimAngle}(\vec{v_2}) \equiv \vec{v}.\text{azimAngle}(\vec{v_2},\hat{z}) = \vec{v_2}.\text{phi}() - \vec{v}.\text{phi}() \tag{47}$$

$$\vec{v}.\text{azimAngle}(\vec{v_2},\hat{u}) = \theta_{\vec{v_2}.\text{perpPart}(\hat{u}),\vec{v}.\text{perpPart}(\hat{u})}\ \text{sign}\,(\hat{u}\cdot(\vec{v}\times\vec{v_2})) \tag{48}$$

The azimuthal angle between two vectors, as shown above, is found by projecting them both into the plane defined by the reference direction, and taking the angle between those projections, in the clockwise sense about the reference axis.

2.6 Direct Vector Rotations

Direct vector rotations are methods of `Hep3Vector` or `HepLorentzVector` which modify the vector being acted upon—`v.rotate`(ϕ,θ,ψ)—or global functions which form a new vector—`rotationOf`$(\vec{v},\phi,\theta,\psi)$.

Rotations about the X, Y, or Z axis are defined in the counter-clockwise sense. Thus for rotations about the Z axis, if $\vec{v}$ has representation in polar coordinates (v_r, v_θ, v_ϕ)

$$\vec{v}.\text{rotateZ}(\delta) \text{ is equivalent to } \vec{v}_\phi \Longrightarrow \vec{v}_\phi + \delta \tag{49}$$

The axis rotations are implemented taking advantage of their simple form. When $\vec{v} = (x,y,z)$ is rotated by angle δ,

$$\vec{v}.\text{rotateZ}(\delta) \Longrightarrow (x\cos\delta - y\sin\delta, x\sin\delta + y\cos\delta, z) \tag{50}$$

$$\vec{v}.\text{rotateY}(\delta) \Longrightarrow (z\sin\delta + x\cos\delta, y, z\cos\delta - x\sin\delta) \tag{51}$$

$$\vec{v}.\text{rotateX}(\delta) \Longrightarrow (x, y\cos\delta - z\sin\delta, y\sin\delta + z\cos\delta) \tag{52}$$

More general rotations may be expressed in terms of an angle δ (counter-clockwise) about an axis given as a `UnitVector` $\hat{u}$, or in terms of Euler Angles (ϕ,θ,ψ):

$$\vec{v}.\text{rotate}(\hat{u},\delta) \Longrightarrow$$

$$\begin{pmatrix} \cos\delta + (1-\cos\delta)u_x^2 & (1-\cos\delta)u_xu_y - \sin\delta u_z & (1-\cos\delta)u_xu_z + \sin\delta u_y \\ (1-\cos\delta)u_yu_x + \sin\delta u_z & \cos\delta + (1-\cos\delta)u_y^2 & (1-\cos\delta)u_yu_z - \sin\delta u_x \\ (1-\cos\delta)u_zu_x - \sin\delta u_y & (1-\cos\delta)u_zu_y + \sin\delta u_x & \cos\delta + (1-\cos\delta)u_z^2 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{53}$$

$$\vec{v}.\text{rotate}(\phi,\theta,\psi) \Longrightarrow$$

$$\begin{pmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{54}$$

The Euler angles definition matches that found in found in *Classical Mechanics* (Goldstein), page 109. This treats the Euler angles as a sequence of counter-clockwise **passive** rotations; that is, the vector remains fixed while the coordinate axes are rotated—new vector components are computed in new coordinate frame. It is unnatural (though possible) to view an Euler angles transformation as as sequence of active rotations.

HEP computations ordinarily use the active rotation viewpoint. Therefore, rotations about an axis imply **active** counter-clockwise rotation in this package.

Consequently, a rotation by angle δ around the X axis is equivalent to a rotation with Euler angles $(\phi = \psi = 0,\ \theta = -\delta)$ and a rotation about the Z axis is equivalent to a rotation with Euler angles $(\theta = 0,\ \phi = \psi = -\delta/2)$.

RotateUz

Another way to specify a direct rotation is via `v.rotateUz(u)` where u is required to be a unit `Hep3Vector`. This rotates the reference frame such that the original Z-axis will lie in the direction of $\hat{u}$. Many rotations would accomplish this; the one selected uses u as its third column and is given by:

$$\vec{v}.\mathrm{rotateUz}(u_x, u_y, u_z) \Longrightarrow$$

$$\begin{pmatrix} u_x u_z/u_\perp & -u_y/u_\perp & u_x \\ u_y u_z/u_\perp & u_x/u_\perp & u_y \\ -u_\perp & 0 & u_z \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{55}$$

Here, $u_\perp \equiv \sqrt{u_x^2 + u_y^2}$. Using u as the third column of the rotation removes the ambiguity except if u is parallel to $\hat{z}$. In that case, if $u = \hat{z}$ the vector $\vec{v}$ is left untouched, while if $u = -\hat{z}$, If u is $-\hat{z}$, $\vec{v}$ is rotated by 180 degrees about the Y axis.

Applying `HepRotations` to `Hep3Vectors`

In CLHEP, the `Hep3Vector` class is aware of the existence of the `HepRotation` class. This is reflected in two routines to apply a `HepRotation` to the `Hep3Vector`:

$$\left.\begin{array}{r} \vec{v} \mathrel{*}= R \\ \vec{v}.\mathrm{transform}(R) \end{array}\right\} \Longleftrightarrow \vec{v} \leftarrow R\vec{v} \tag{56}$$

Notice that these are identical. In contrast to the usual `operator *=` semantics, `v *= R` *left* multiplies the matrix representing `v` by the matrix representing `R`.

2.7 Overflow for Large Vectors

When vector components are near the limits of floating point representation, there are cases where operations have mathematical results which can be represented, but intermediate steps give too large a result. An extreme example is that testing whether two vectors are nearly orthogonal should give a result of type `bool`, yet the intermediate steps may involve a dot product adding three huge terms.

Although each of the vector operations described above can be performed protecting against overflow (by judicious re-scaling) this generally leads to unacceptable inefficiency in the overwhelming majority of cases where the vectors are not so large. The compromise used in this package is:

- All vectors are assumed to be "squarable," that is, the algorithms may freely take a dot product of a vector with itself. For `double` numbers, this implies that components are limited to about 10^{153} in magnitude.

- Care is taken that any operation, applied to squarable vectors, gives the proper result. This implies that we never take products of more than two powers of components without checking for size and potentially re-scaling. In particular, `isParallel()` and `isOrthogonal()` do extra work to avoid trouble when vectors have components on the order of 10^{76} to 10^{152}.

3 HepLorentzVector Class

We always take the time component to be the '4' index, and $c = 1$. For these equations, let the sense of the metric be $\mathcal{M} = \pm 1$ and the metric be represented as g_{ij} where $g_{44} = \mathcal{M}$, $g_{ii} = -\mathcal{M}$, and all other $g_{ij} = 0$.
Let $\mathbf{w}_1$ and $\mathbf{w}_2$ be HepLorentzVectors. Then $\mathbf{w}_1.\mathrm{dot}(\mathbf{w}_2)$ is defined by:

$$\mathbf{w}_1 \cdot \mathbf{w}_2 = \sum_{ij} g_i j \mathbf{w}_1^i \mathbf{w}_2^j \tag{57}$$

The sign of this dot product is dependent on $\mathcal{M}$.

For the remainder of these definitions we will take the metric to be $(- - - +)$ and point out any definitions which change sign when the $(+ + + -)$ metric is chosen.

3.1 Combinations and Properties of HepLorentzVectors

The dot product, and Lorentz-invariant magnitude squared and squared norm of the difference between two 4-vectors, are metric-dependent.

Let $w_i = (\vec{v}_i, t_i)$:

$$w_1.\mathrm{dot}(w_2) = w_1 \cdot w_2 = t_1 t_2 - \vec{v}_1 \cdot \vec{v}_2 \tag{58}$$

$$w_1.\mathrm{mag2}() = w_1 \cdot w_1 = t_1^2 - |\vec{v}_1|^2 \tag{59}$$

$$w_1.\mathrm{diff2}(w_2) = (w_1 - w_2) \cdot (w_1 - w_2) = (t_1 - t_2)^2 - |\vec{v}_1 - \vec{v}_2|^2 \tag{60}$$

$$w_1.\mathrm{mag}(\ ) = w_1.\mathrm{m}(\ ) = \mathrm{sign}(t_1^2 - \vec{v}_1^2)\sqrt{|t_1^2 - \vec{v}_1^2|} \tag{61}$$

The Euclidean-norm, and Euclidean-norm difference squared, are given by

$$w_1.\mathrm{EuclideanNorm2}() = t_1^2 + |\vec{v}_1|^2 \tag{62}$$

$$w_1.\mathrm{EuclideanNorm}() = \sqrt{t_1^2 + |\vec{v}_1|^2} \tag{63}$$

$$w_1.\mathrm{delta2Euclidean}(w_2) = t_1 t_2 + \vec{v}_1 \cdot \vec{v}_2 \tag{64}$$

It is convenient to have methods returning $t \pm z$; for completeness we also provide the plus() and minus() methods relative to an arbitrary direction $\hat{u}$:

$$w_1.\mathrm{plus}() = t_1 + z_1 \tag{65}$$

$$w_1.\mathrm{minus}() = t_1 - z_1 \tag{66}$$

$$w_1.\mathrm{plus}(\hat{u}) = t_1 + \vec{v}_1 \cdot \hat{u} \tag{67}$$

$$w_1.\mathrm{minus}(\hat{u}) = t_1 - \vec{v}_1 \cdot \hat{u} \tag{68}$$

3.2 Kinematics of HepLorentzVectors

The rest mass and its square are independent of metric. Note that `restMass2()` differs from `mag2()` in that no matter which metric is selected, `restMass2()` remains $t^2 - v^2$. The sign of the rest mass is set to match the time component; thus the rest mass of $(0, 0, 0, -m)$ will return as $-m$.

The method `rest4Vector()` will return a 4-vector equal to this vector in its rest frame.

$$p.\text{restMass2}() = p.\text{invariantMass2}() = E^2 - |\vec{p}|^2 \tag{69}$$
$$p.\text{restMass}() = p.\text{invariantMass}() = \sqrt{E^2 - |\vec{p}|^2} \times \text{sign}(t_1) \tag{70}$$
$$p.\text{rest4Vector}() = (0, 0, 0, p.\text{restMass}()) \tag{71}$$

Taking the rest mass of a spacelike 4-vector will ZMthrow the exception `ZMxpvSpacelike`. The boost which if applied to `w.rest4Vector()` would give the result `w` is `w.boostVector()`. A pure boost can be expressed as a `Hep3Vector`:

$$w_1.\text{boostVector}() = \frac{\vec{v}_1}{t_1} \tag{72}$$
$$(w_1.\text{rest4Vector}())\,.\text{boosted}\,(w_1..\text{boostVector}()) \equiv w_1$$

The boostVector for a zero 4-vector will return as zero. `boostVector()` will return v/t even if the 4-vector is spacelike, but will ZMthrow a `ZMxpvTachyonic` error. If $t = 0$ it will throw `ZMxpvInfiniteVector` and do the divisions, returning an infinite vector if ignored.

Beta and gamma refer to this boost vector:

$$w_1.\text{beta}() = \beta = |w_1.\text{boostVector}()| = \frac{|\vec{v}_1|}{|t_1|} \tag{73}$$
$$w_1.\text{gamma}() = \gamma = \frac{1}{\sqrt{1-\beta^2}} \tag{74}$$

Psuedorapidity and rapidity are discussed below (§4):

$$w_1.\text{eta}() = -\ln\tan\frac{\theta}{2} \tag{75}$$
$$w_1.\text{eta(u)} = -\ln\tan\frac{\hat{p}\cdot\hat{u}}{2} \tag{76}$$
$$w_1.\text{rapidity}() = \frac{1}{2}\ln\left(\frac{E+p_z}{E-p_z}\right) = \tanh^{-1}\frac{z}{t} \tag{77}$$
$$w_1.\text{rapidity(u)} = \frac{1}{2}\ln\left(\frac{E+\vec{p}\cdot\hat{u}}{E-\vec{p}\cdot\hat{u}}\right) = \tanh^{-1}\frac{\vec{v}\cdot\hat{u}}{t} \tag{78}$$
$$w_1.\text{coLinearRapidity}() = \frac{1}{2}\ln\left(\frac{E+|p|}{E-|p|}\right) = \tanh^{-1}\frac{|v|}{t} \tag{79}$$

The "transverse mass" is found by neglecting the x- and y-components of the 4-vector. This is the square root of the sum of the rest mass squared and the transverse momentum squared:

$$p.\text{mt2}() = E^2 - p_z^2 \tag{80}$$
$$p.\text{mt}() = \sqrt{|E^2 - |\vec{p}|^2|} \times \text{sign}(E^2 - |\vec{p}|^2) \tag{81}$$

Note that $m_t \geq m$. *Warning:* Although this definition for m_t matches that of equation (34.36) in the Review of Particle Physics and this definition has always been in CLHEP, some experimenters have indicated that various experiments may use different definitions, which are of more use to their applications.

The "transverse energy" is defined as $E\sin\theta$. This quantity is invariant, to order m/E, under boosts in the Z direction. It is most easily expressed in terms of the energy, momentum, and transverse momentum:

$$p.\mathrm{et2}() = \frac{E^2 p_\perp^2}{|p|^2} \tag{82}$$

$$p.\mathrm{et}() = \sqrt{\left|\frac{E^2 p_\perp^2}{|p|^2}\right|} \times \mathrm{sign}(E) \tag{83}$$

For completeness, the transverse mass and energy are also defined with respect to a direction specified by a `Hep3Vector`$\vec{v}$:

$$p.\mathrm{mt2}(\vec{v}) = E^2 - (\vec{p}\cdot\hat{v})^2 \tag{84}$$

$$p.\mathrm{mt}(\vec{v}) = \sqrt{|E^2 - (\vec{p}\cdot\hat{v})^2|} \times \mathrm{sign}(E^2 - (\vec{p}\cdot\hat{v})^2) \tag{85}$$

$$p.\mathrm{et2}(\vec{v}) = \frac{E^2 p.\mathrm{perp}(\hat{v})^2}{|p|^2} \tag{86}$$

$$p.\mathrm{et}(\vec{v}) = \sqrt{\left|\frac{E^2 p.\mathrm{perp}(\hat{v})^2}{|p|^2}\right|} \times \mathrm{sign}(E) \tag{87}$$

3.3 Invariant Mass and the Center-of-Mass Frame

The invariant mass of a pair of `HepLorentzVectors` is given by:

$$w_1.\mathrm{invariantMass2}(w_2) = (t_1 + t_2)^2 - |\vec{v}_1 - \vec{v}_2|^2 \tag{88}$$

$$w_1.\mathrm{invariantMass}(w_2) = (t_1 + t_2)^2 - |\vec{v}_1 - \vec{v}_2|^2 \times \mathrm{sign}(t_1 + t_2) \tag{89}$$

The boost necessary top bring a pair of vectors into their Center-of-Mass frame is given by

$$w_1.\mathrm{findBoostToCM}(w_2) = -\frac{\vec{v}_1 + \vec{v}_2}{t_1 + t_2} \tag{90}$$

If the sum of the two 4-vectors is spacelike, this makes analytic sense but is physically meaningless; a `ZMxpvTachyonic` error will be ZMthrown. If the sum of the time components is zero, a `ZMxpvInfiniteVector` error will be ZMthrown.

3.4 Various Forms of Masses and Magnitudes

The `HepLorentzVector` class combines the features of the CLHEP and ZOOM 4-vectors. Each of these deal with several concepts related to the magnitude or "mass" associated with the 4-vector.

This is further complicated by the possibility that the metric, normally taken to be $(- - - +)$ which we designate as $\mathcal{M} = +1$, can be set to $(+ + + -)$ which we will designate as $\mathcal{M} = -1$. So some of the definitions below will involve $\mathcal{M}$. In the CLHEP original package, of course, $\mathcal{M}$ would always be $+1$.

These methods are defined above, but perhaps it will be helpful to provide them all in one place. In the below definitions, since we are conceptually dealing with energy-momentum 4-vectors, we will use $(\vec{p}, e)$ for the 4-vector components.

$$w.\text{mag2}() = w.\text{dot(w)} = \mathcal{M}(E^2 - \vec{p}^2) \tag{91}$$
$$w.\text{m2}(\ ) = E^2 - \vec{p}^2 \tag{92}$$
$$w.\text{mag}(\ ) = w.\text{m}(\ ) = \text{sign}(E^2 - \vec{p}^2)\sqrt{|E^2 - |\vec{p}|^2|} \tag{93}$$
$$w.\text{invariantMass2}(\ ) = w.\text{restMass2}(\ ) = E^2 - |\vec{p}|^2 \tag{94}$$
$$w.\text{w.invariantMass}(\ ) = w.\text{restMass}(\ ) = \text{sign}(E)\sqrt{E^2 - |\vec{p}|^2} \tag{95}$$
$$w.\text{mt2}(\ ) = E^2 - p_z^2 \tag{96}$$
$$w.\text{mt}(\ ) = \text{sign}(E^2 - p_z^2)\sqrt{|E^2 - p_z^2|} \tag{97}$$
$$w.\text{et2}() = \frac{E^2 p_\perp^2}{|p|^2} \tag{98}$$
$$w.\text{et}() = \sqrt{\left|\frac{E^2 p_\perp^2}{|p|^2}\right|} \times \text{sign}(E) \tag{99}$$

Thus a tachyonic particle (for which $t^2 - v^2 < 0$) is assigned a negative mass m(), but a positive restMass().

3.5 Direct `HepLorentzVector` Boosts and Rotations

Direct boosts and rotations are methods of `HepLorentzVector` which modify the 4-vector being acted upon— *e.g.*, `w.boost`$(\hat{u}, \beta)$—or global functions which form a new vector— *e.g.*, `boostOf`$(w, \hat{u}, \beta)$.

Rotations act in the obvious manner, affecting only the $\vec{v}$ component of the 4-vector—see §2.6.

In analogy with our "active" rotation viewpoint, boosts are treated as "active" transformations rather than transformations of the coordinate system. That is, if you take a 4-vector at rest, with positive mass (t), and boost it by a positive amount in the X direction, the resulting 4-vector will have positive x.

Boosts along the X, Y, or Z axis are simpler than the general case. Let $w = (\vec{v}, t) = (x, y, z, t)$:

$$w.\text{boostX}(\beta) \Longrightarrow (\gamma x + \beta\gamma t, y, z, \gamma t + \beta\gamma x) \tag{100}$$
$$w.\text{boostY}(\beta) \Longrightarrow (x, \gamma y + \beta\gamma t, z, \gamma t + \beta\gamma y) \tag{101}$$
$$w.\text{boostZ}(\beta) \Longrightarrow (x, y, \gamma z + \beta\gamma t, \gamma t + \beta\gamma z) \tag{102}$$
$$\gamma \equiv \frac{1}{\sqrt{1 - \beta^2}}$$

More general rotations boosts may be expressed in terms of β along an axis given as a `Hep3Vector` $\hat{u}$ (which will be normalized), or in terms of a `Hep3Vector` boost $\vec{\beta}$, which must obey $|\vec{\beta}| < 1$. (Boosts beyond the speed of light ZMthrow a `ZMxpvTachyonic` error, and leave the 4-vector unchanged if this is ignored.)

For the axis $(\hat{u}, \beta)$ form,

$$\left\{ \begin{array}{lcl} t & \longleftarrow & \gamma t + \beta\gamma\vec{v}\cdot\hat{u} \\ \vec{v} & \longleftarrow & \vec{v} + \left[\frac{\gamma-1}{\beta^2}\beta\vec{v}\cdot\hat{u} + \beta\gamma t\right]\hat{u} \end{array} \right. \tag{103}$$

$$\gamma \equiv \frac{1}{\sqrt{1-\beta^2}}$$

For the boost vector ($\vec{\beta}$) form,

$$\begin{cases} t & \longleftarrow & \gamma t + \gamma \vec{v} \cdot \vec{\beta} \\ \vec{v} & \longleftarrow & \vec{v} + \left[\frac{\gamma-1}{|\vec{\beta}|^2} \vec{v} \cdot \vec{\beta} + \gamma t\right] \vec{\beta} \end{cases} \tag{104}$$

$$\gamma \equiv \frac{1}{\sqrt{1-|\vec{\beta}|^2}}$$

Applying `HepRotations` and `HepLorentzRotations` to `Hep3Vectors`

In CLHEP, the `HepLorentzVector` class is aware of the existence of the `HepRotation` and `HepLorentzRotation` classes. This is reflected in routines to apply these the `HepLorentzVector`:

$$\left.\begin{array}{r} p \mathrel{*}= R \\ p.\mathrm{transform}(R) \end{array}\right\} \Longleftrightarrow p \leftarrow Rp \tag{105}$$

Notice that these are identical. In contrast to the usual `operator *=` semantics, `p *= R` *left* multiplies the matrix representing `p` by the matrix representing `R`.

3.6 Near-equality of `HepLorentzVectors`

We keep in mind that the prime utility of `isNear()` and related methods is to see whether two vectors, which have been created along two computational paths or from two sets of fuzzy quantities, ought mathematically to be taken as equal. *Relative* tolerance is needed, and this is more involved than just checking for approximate equality in the time and space sectors respectively. For example, though no non-zero vector can be near the zero vector, $(\vec{\epsilon}, t=1)$ with $\vec{\epsilon}$ a very small vector should be considered close to $(\vec{0}, t=1)$.

The temptation is to use, as the normaliztion to determine relative nearness, the length $|\vec{v}|^2 + t^2$. Let us call this the Euclidean norm, since it is the norm in complex 4-space $(\vec{v}, it)$. This is not Lorentz invariant, but for many purposes is a good criteria for calling two 4-vectors close. We use the Euclidean norm to define `isNear()` for `LorentzVectors`; it has the virtue of simplicity and can be applied to any 4-vectors. Let the method `isNear()` be represented by the symbol $\approx$:

$$w_1 \approx w_2 \Longleftrightarrow |\vec{v}_1 - \vec{v}_2|^2 + (t_1 - t_2)^2 \le \epsilon^2 \left[|\vec{v}_1 \cdot \vec{v}_2| + \left(\frac{t_1 + t_2}{2}\right)^2 \right] \tag{106}$$

This definition not Lorentz invariant, but it is (to order $\epsilon \times \gamma$) independent of frame within the space of *small* Lorentz transformations. It turns out to be impossible to find a definition which is Lorentz invariant for all possible 4-vectors, under arbitrarily large boosts, and still behaves like a measure of near-ness.

A second useful definition, which is Lorentz invariant, but is sensibly applicable only to timelike 4-vectors, is to look at the Euclidean norm of the difference of two vectors *in their Center-of-Mass frame*. This is intuitively appealing to HEP practitioners; we make this method available as well, calling it `isNearCM()`.

Let the method `isNear()` be represented by the symbol $\approx$, and the method `isNearCM()` be represented by the symbol $\stackrel{\mathrm{CM}}{\approx}$. Let $\mathbf{w}_1$, $\mathbf{w}_2$ be LorentzVectors. Also, let $\vec{v}_i, t_i$ be the space vector and time components of $\mathbf{w}_i$. Then using the boost vector to the joint center of mass frame

$$\vec{b} = -\frac{|\vec{v}_1 + \vec{v}_2|}{t_1 + t_2} \tag{107}$$

and assuming that $|\vec{b}| < 1$ so we can take

$$\beta = |\vec{b}| \tag{108}$$
$$\gamma = \frac{1}{\sqrt{1-\beta^2}} \tag{109}$$

we define the condition

$$w_1 \overset{\text{CM}}{\approx} w_2 \Longleftrightarrow \left(w_1.\text{boost}(\vec{b})\right) \approx \left(w_2.\text{boost}(\vec{b})\right) \tag{110}$$

As mentioned earlier, this criterion makes sense only for timelike 4-vectors. If applied to 4-vectors whose sum is not timelike, the `isNearCM()` method will return a test for exact equality.

$$\text{if } |\vec{v}_1 + \vec{v}_1| \geq |t_1 + t_2| \text{ then } w_1 \overset{\text{CM}}{\approx} w|2 \Longleftrightarrow w_1 = v_2 \tag{111}$$

Ordering Comparisons for HepLorentzVectors

The comparison operators (>, >=, <, <=) for `HepLorentzVector` act by comparing first the time component, then the `Hep3Vector` part. The latter comparison is done using definition (25).

$$w_1 > w_2 \text{ if } t_1 > t_2 \text{ or } [t_1 = t_2 \text{ and } \vec{v}_1 > \vec{v}_2] \tag{112}$$

3.7 Other Boolean Methods for HepLorentzVectors

The `w1.isParallel(w2)` method works with a relative tolerance. If the difference of the normalized 4-vectors is small, then those 4-vectors are considered nearly parallel. For this purpose, we use the Euclidean norm to define the normalization and size of difference.

Let

$$\overline{w}_1 \equiv \frac{w_1}{|\vec{v}_1|^2 + t_1^2}$$
$$\overline{w}_2 \equiv \frac{w_1}{|\vec{v}_2|^2 + t_2^2}$$

and

$$\overline{w}_1 - \overline{w_2} \equiv (\overline{v}_{12}, \overline{t}_{12})$$

then

$$w_1 \parallel w_2 \text{ iff } |\overline{v}_{12}|^2 + \overline{t}_{12}^2 \leq \epsilon^2 \tag{113}$$

As in the case of `Hep3Vectors`, only the zero 4-vector is considered parallel to the zero 4-vector.

$$w \parallel (0,0,0,0) \text{ iff } w = (0,0,0,0) \tag{114}$$

The `w1.howParallel(w2)` method, applied to two non-zero `HepLorentzVectors`, returns the Euclidean norm of the differnce. This can range from zero (if `w2` is a positive multiple of `w1`) to 2 (if `w2` is a negative multiple of `w1`). If both `HepLorentzVectors` are zero, `w1.howParallel(w2)` returns zero; if one is zero, it returns 1.

The boolean tests `isSpacelike()`, `isTimelike()`, and `isLightlike()`, work with the `restMass2()` function, which returns $t^2 - |\vec{v}|^2$.

$$w\text{.isSpacelike()} \iff t^2 - |\vec{v}|^2 < 0 \tag{115}$$
$$w\text{.isTimelike()} \iff t^2 - |\vec{v}|^2 > 0 \tag{116}$$

The test for `isLightlike()` uses a tolerance relative to the time component of the vector. It determines if, starting from an exactly lightlike vector, you can perturb t and v by a small relative amount to reach the actual vector.

$$w\text{.isLightlike()} \iff \left|t^2 - |\vec{v}|^2\right| <= 2\epsilon t^2 \tag{117}$$

The $2\epsilon t^2$ limit is chosen such that $(0, 0, (1 \pm \epsilon)t, t)$ is just on the boundary of being considered lightlike.

Since the `isLightlike()` method is tolerant of small perturbations, these three methods are not mutually exclusive: A 4-vector can test true for `isLightlike()` and either `isSpacelike()` or `isTimelike()` as well.

By these definitions, the zero 4-vector is considered lightlike.

3.8 Nearness measures for `HepLorentzVectors`

Since both `isNear()` and `isNearCM()` for `HepLorentzVectors` use relative tolerance, the corresponding nearness measures are truncated at a maximum of 1:

$$w_1\text{.howNear}(w_2) = \max\left(\sqrt{\frac{\left|\vec{v}_1 - \vec{v}_2\right|^2 + (t_1 - t_2)^2}{|\vec{v}_1 \cdot \vec{v}_2| + \left(\frac{t_1+t_2}{2}\right)^2}}\ ,\ 1\right) \tag{118}$$

$$w_1\text{.howNearCM}(w_2) = \left(w_1\text{.boost}(\vec{b})\right)\text{.howNear}\left(w_1\text{.boost}(\vec{b})\right) \tag{119}$$

with, as before, the boost vector $\vec{b}$ given by

$$\vec{b} = -\frac{|\vec{v}_1 + \vec{v}_2|}{t_1 + t_2}$$

For two unequal `HepLorentzVectors`, `w1.howNearCM(w2)` will return 1 if the boost to the rest frame is tachyonic.

The `w1.isParallel(w2)` and `w.isLightlike()` methods also work with a relative tolerance. The corresponding measures are defined by:

$$w_1\text{.howParallel}(w_2) = \max\left(\sqrt{\left|\overline{v}_{12}\right|^2 + \overline{t}_{12}^2}\ ,\ 1\right) \tag{120}$$

(where the notation used is that use for equation 113).
If $w = (v, t)$

$$w\text{.howLightlike()} = \max\left(\frac{\left|t^2 - |\vec{v}|^2\right| 2t^2}{},\ 1\right) \tag{121}$$

As before, if this measure is very nearly zero, the 4-vectors are nearly parallel or the 4-vector is nearly lightlike.

When one of the 4-vectors (but not the other) is zero,

$$w.\mathrm{howNear}(\mathit{0}) = \mathit{0}.\mathrm{howNear}(w) = 1 \tag{122}$$

$$w.\mathrm{howNearCM}(\mathit{0}) = \mathit{0}.\mathrm{howNearCM}(w) = 1 \tag{123}$$

$$w.\mathrm{howParallel}(\mathit{0}) = \mathit{0}.\mathrm{howParallel}(w) = 1 \tag{124}$$

$$\mathit{0}.\mathrm{howLightlike}() = 0 \tag{125}$$

When both 4-vectors are zero, the nearness measures will all return zero.

4 Pseudorapidity, Rapidity and CoLinearRapidity

Pseudorapidity (conventionally labelled η) and rapidity are properties which apply to both `Hep3Vectors` and `HepLorentzVectors`. Pseudorapidity and rapidity are defined relative to the z direction.

The pseudorapidity `eta` of either a `HepLorentzVector` or a `Hep3Vector` is defined in terms of the angle the 3-vector part forms with the Z axis: This can be found knowing nothing of the mass of a particle, and in the limit of large momentum is approximately the same as the true rapidity. Unlike the case for rapidity(), eta() makes mathematical sense for any vector—it a simple function of the tangent of the angle between the vector and the Z axis. The pseudorapidity of a zero vector will be assigned the value zero.

Let $\vec{v_1}$ be a `Hep3Vector`, and w_1 be a `HepLorentzVector` with decomposition $(t, \vec{v})$. And let the angle formed between the Z axis and $\vec{v_1}$ or $\vec{v}$ be θ. Then:

$$v_1.\mathrm{eta}() = -\ln\tan\frac{\theta}{2} \tag{126}$$

$$w_1.\mathrm{eta}() = -\ln\tan\frac{\theta}{2} \tag{127}$$

The true rapidity of a LorentzVector is defined (see the Kinematics section of the Review of Particle Properties, page 177 in the 1997 version) such that the rapidity transforms under a boost *along the Z axis* by adding the rapidity of the boost: This treats the LorentzVector as a timelike 4-momentum (using the t and z components as E and p_z).

In analogy with the familiar definition for `HepLorentzVector`, the rapidity of a `Hep3Vector` is defined with respect to the z direction. Mathematically, this is the same as the rapidity of a 4-vector $(\vec{v}, 1)$.

Letting w_1 be (x, y, z, t) or (p_x, p_y, p_z, E),

$$w_1.\mathrm{rapidity}() = \frac{1}{2}\ln\left(\frac{E+p_z}{E-p_z}\right) = \tanh^{-1}\frac{z}{t} \tag{128}$$

$$v_1.\mathrm{rapidity}() = \tanh^{-1}(\vec{v}_1 \cdot \hat{z}) \tag{129}$$

The shape of a rapidity distribution is a invariant under boosts in the z direction. Pseudorapidity can always be determined from direction information alone, and when $p^2 \gg m^2$ *and the time component is positive* pseudorapidity matches rapidity to order m^2/p^2.

Although the z direction is special for many HEP uses, rapidity and pseudorapidity can be defined with respect to an arbitrary direction $\hat{u}$. Letting w_1 be $(\vec{v}, t)$:

$$w_1.\mathrm{eta(u)} = -\ln\tan\frac{\hat{p}\cdot u}{2} \tag{130}$$

$$v_1.\mathrm{eta(u)} = -\ln\tan\frac{\hat{v}_1\cdot\hat{u}}{2} \tag{131}$$

$$w_1.\mathrm{rapidity(u)} \;= \frac{1}{2}\ln\left(\frac{E+\vec{p}\cdot\hat{u}}{E-\vec{p}\cdot\hat{u}}\right) = \tanh^{-1}\frac{\vec{v}\cdot\hat{u}}{t} \tag{132}$$

$$v_1.\mathrm{rapidity(u)} \;= \tanh^{-1}(\vec{v}_1\cdot\hat{u}) \tag{133}$$

Relativity texts discuss rapidity along the direction of the vector. This concept can apply to `Hep3Vector`s or `HepLorentzVector`s. This function adds when you compbine two boosts in the same direction.

Such a method may not be needed for typical HEP calculations but is provided for completeness. To distinguish this from the rapidity with respect to a specific direction—or a default rapidity, which is with respect to $\hat{z}$—the package names the rapidity along the direction of the vector `coLinearRapidity()`.

$$v_1.\mathrm{coLinearRapidity()} \;= \tanh^{-1}\beta = \tanh^{-1}|\vec{v_1}| \tag{134}$$

$$w_1.\mathrm{coLinearRapidity()} \;= \frac{1}{2}\ln\left(\frac{E+|p|}{E-|p|}\right) = \tanh^{-1}\frac{|v|}{t} \tag{135}$$

The co-linear rapidity of a 3-vector is inherently non-negative; for a 4-vector it will have the same sign as the time component of the 4-vector.

5 `HepRotation` Class

`HepRotation`s may be expressed in terms of an axis $\hat{u}$ and angle δ of counter-clockwise rotation, or as a set of three Euler Angles (ϕ, θ, ψ). Definitions and conventions for these classes match those described in §2.6 for directly rotating a `Hep3Vector`:

$$\mathrm{HepRotation}(\hat{u},\delta) \Longrightarrow$$

$$\begin{pmatrix} \cos\delta + (1-\cos\delta)u_x^2 & (1-\cos\delta)u_xu_y - \sin\delta u_z & (1-\cos\delta)u_xu_z + \sin\delta u_y \\ (1-\cos\delta)u_yu_x + \sin\delta u_z & \cos\delta + (1-\cos\delta)u_y^2 & (1-\cos\delta)u_yu_z - \sin\delta u_x \\ (1-\cos\delta)u_zu_x - \sin\delta u_y & (1-\cos\delta)u_zu_y + \sin\delta u_x & \cos\delta + (1-\cos\delta)u_z^2 \end{pmatrix} \tag{136}$$

$$\mathrm{HepRotation}(\phi,\theta,\psi) \Longrightarrow$$

$$\begin{pmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{pmatrix} \tag{137}$$

The Euler angles definition matches that found in found in *Classical Mechanics* (Goldstein), page 109. This treats the Euler angles as a sequence of counter-clockwise **passive** rotations; that is, the vector remains fixed while the coordinate axes are rotated—new vector components are computed in new coordinate frame.

HEP computations ordinarily use the active rotation viewpoint. Therefore, rotations about an axis imply **active** counter-clockwise rotation in this package.

Consequently, a rotation by angle δ around the X axis is equivalent to a rotation with Euler angles ($\phi = \psi = 0,\ \theta = -\delta$) and a rotation about the Z axis is equivalent to a rotation with Euler angles ($\theta = 0,\ \phi = \psi = -\delta/2$).

5.1 Applying Rotations to Vectors and 4-Vectors

A `HepRotation`may be applied to a `Hep3Vector`using either of two notations:

$$\mathrm{R} * \mathrm{v} \equiv \mathrm{R}(\mathrm{v}) \equiv R\vec{v} \tag{138}$$

where R is the matrix representing R, as in equation 137 or 136. The same syntaxes may be used to apply a `HepRotation`to a `HepLorentzVector`—the rotation matrix acts on the space components of the 4-vector.

Note that `R *= v` is meaningless and not supported. Also note the warning given earlier: $\vec{v}$*=$R \Longleftrightarrow \vec{v} = R$*$\vec{v}$

5.2 Axial Rotations

The special case rotations along the X, Y, and Z axes will often be specified by just the rotation angle δ. Thus a `HepRotationX`, `HepRotationY`, or `HepRotationZ` with angle δ would be represented by the matrix

$$\mathrm{HepRotationX}(\delta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\delta & -\sin\delta \\ 0 & \sin\delta & \cos\delta \end{pmatrix} \tag{139}$$

$$\mathrm{HepRotationY}(\delta) = \begin{pmatrix} \cos\delta & 0 & \sin\delta \\ 0 & 1 & 0 \\ -\sin\delta & 0 & \cos\delta \end{pmatrix} \tag{140}$$

$$\mathrm{HepRotationZ}(\delta) = \begin{pmatrix} \cos\delta & -\sin\delta & 0 \\ \sin\delta & \cos\delta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{141}$$

$$\tag{142}$$

A `HepRotation` is an object in its own right; two `HepRotations` may be multiplied, tested for near equality, and so forth. Multiplication is done by multiplying the matrix representations of two `HepRotations` (though for matching axis rotations, simplifications are done to effciently compute this product).

The matrix representing a `HepRotation` is orthonormal: Each row, considered as a vector, has length 1, and the dot product of any two distinct rows is zero. This leads to a trivial inversion method—simply transpose the matrix.

When dealing with structures holding an axis and angle, or Euler angles, we do not provide direct analogues of all rotation methods. For example, we do not provide for multiplication of two `HepAxisAngle`objects to form a third `HepAxisAngle`. We do provide comparison and nearness methods for these classes, and this document specifies whether the criteria match the corresponding `Rotation` criteria.

5.3 Expressing a `HepRotation` as `HepAxisAngle` or `HepEulerAngles`

It is worth noting that equations (53) and (54) are not single-valued when extracting $(\hat{u}, \delta)$ or (ϕ, θ, ψ) from a `Rotation` matrix. We adhere to the following conventions to resolve that ambiguity whenever forming an `HepAxisAngle` or `HepEulerAngles` from a `Rotation`R:

$$R = \mathbf{I} \Longrightarrow \delta = 0,\ \hat{u} = \hat{z} \tag{143}$$
$$R \longrightarrow (\hat{u}, \delta) \Longrightarrow 0 \leq \delta \leq \pi \tag{144}$$
$$R \longrightarrow (\phi, \theta, \psi) \Longrightarrow 0 \leq \theta \leq \pi \tag{145}$$
$$R \longrightarrow (\phi,\ \theta = 0,\ \psi) \Longrightarrow -\pi/2 < \phi = \psi \leq \pi/2 \tag{146}$$
$$R \longrightarrow (\phi,\ \theta = \pi,\ \psi) \Longrightarrow -\pi/2 < \phi = -\psi \leq \pi/2 \tag{147}$$

Thus when supplying values for Euler angles, we return $(0, 0, 0)$ whenever the rotation is equivalent to the identity, and return $(0, \pi, 0)$ in preference to the equivalent (π, π, π).

In particular, special case rotations such as `HepRotationX` with δ supplied as zero or π obey these conventions: even though for $\delta = \pi - |\epsilon|$ a `RotationX` would have Euler angles $(\pi, \delta, \pi,)$, when $\delta = \pi$ exactly, the Euler angles are $(0, \delta = \pi, 0)$.

These conventions for how methods return values for Euler angles do not affect the user's right to supply explicit values for $(\hat{u}, \delta)$ or (ϕ, θ, ψ) when defining one of those structures. Similarly, the conventions for reading axis and angle do not affect the user's ability to supply arbitrary values for the angle δ.

Obeying the above rules, Euler angles returned for rotations about coordinate axes behave as follows:

$$\text{HepRotationX}(\delta) \longrightarrow \begin{cases} (0, -\delta, 0) & \text{if} \quad \delta \leq 0 \\ (\pi, \delta, \pi) & \text{if} \quad 0 < \delta < \pi \\ (0, \pi, 0) & \text{if} \quad \delta = \pi \end{cases} \tag{148}$$

$$\text{HepRotationY}(\delta) \longrightarrow \begin{cases} (+\pi/2, -\delta, -\pi/2) & \text{if} \quad \delta < 0 \\ (0, 0, 0) & \text{if} \quad \delta = 0 \\ (-\pi/2, \delta, +\pi/2) & \text{if} \quad 0 < \delta < \pi \\ (\pi/2, \delta, -\pi/2) & \text{if} \quad \delta = \pi \end{cases} \tag{149}$$

$$\text{HepRotationZ}(\delta) \longrightarrow (-\delta/2, 0, -\delta/2) \tag{150}$$

always assuming that δ represents an angle of active rotation between pi and π.

5.4 Nearness Measure for `HepRotations`, `HepAxisAngles`, and `HepEulerAngles`

The definition used for `isNear()` and `howNear()` on `Rotations` has the following properties:

1. `isNear()` and `howNear()` use the same measure: Two rotations are considered near if their `howNear()` measure is less that ϵ.

2. Transitivity: $r_1 \approx r_2$ and `r1.howNear(r2)` are exactly equivalent to $r_2 \approx r_1$ and `r2.howNear(r1)`.

3. Rotational invariance to order ϵ: If $x =$`r1.howNear(r2)` is small, then for any third rotation `r3` `(r3*r1).howNear(r3*r2)` $x + O(x^2)$.

4. For the special case of rotations that happen to be around the same axis, the measure agrees with a natural definition for measure of rotation about an axis (§5.4).

Although it is possible to formulate a measure definition with exact rotational invariance—based on $\sup_{\hat{u}} \{|R_1(\hat{u}) - R_2(\hat{u})|\}$—this definition would require finding the 2-norm (or the largest eigenvalue) of the matrix $r_1 r_2^{-1}$, which is computationally difficult. The measure we use is the same for small answers, and has the above desirable properties.

$$r_1.\mathrm{howNear}(r_2) = \sqrt{3 - \mathrm{Tr}(r_1 r_2^{-1})} = \sqrt{3 - \sum_{ij} r_{1_{ij}} r_{2_{ij}}} \tag{151}$$

$$r_1.\mathrm{distance2}(r_2) = 3 - \mathrm{Tr}(r_1 r_2^{-1}) = 3 - \sum_{ij} r_{1_{ij}} r_{2_{ij}} \tag{152}$$

$$r_1 \approx r_2 \Longleftrightarrow 3 - \mathrm{Tr}(r_1 r_2^{-1}) \leq \epsilon^2 \tag{153}$$

And a norm is provided:

$$r_1.\mathrm{norm2}(\ ) = 3 - \mathrm{Tr}(r_1) \tag{154}$$

`HepAxisAngle`s have the property that two apparently unequal forms may be equivalent, for example if the axes are in opposite directions and one angle is the negative of the other. Similarly, two different-looking `HepEulerAngles` can represent the same rotation. To avoid a whole spectrum of special cases, we adapt the rule that, letting Υ be an `HepAxisAngle`and Ξ be an `HepEulerAngles` structure, and $R(\Upsilon), R(\Xi)$ be the corresponding `HepRotations`,

$$\Upsilon_1.\mathrm{howNear}(\Upsilon_2) \equiv R(\Upsilon_1).\mathrm{howNear}(R(\Upsilon_2)) \tag{155}$$

$$\Xi_1.\mathrm{howNear}(\Xi_2) \equiv R(\Xi_1).\mathrm{howNear}(R(\Xi_2)) \tag{156}$$

Since for double precision computations these nearness measures cannot count on precision to better than 10^{-8}, the default tolerance for Rotations (which may be modified by the class static method `setTolerance()`) is set to 100 times that.

$$\epsilon_{\mathrm{default}} = 10^{-6} \tag{157}$$

Nearness for `HepRotations` About the Same Axis

The above definition of `howNear()` for two general `Rotations` reduces, when both `Rotations` are around the same axis by angles δ_1 and δ_2, to

$$r_1.\mathrm{howNear}(r_2) = \sqrt{2 - 2\cos(\delta_1 - \delta_2)} = |\delta_1 - \delta_2| + O\left((\delta_1 - \delta_2)^3\right) \tag{158}$$

(Note that when the two rotations are special-case coordinate axis rotations, computing the cosine of that angle difference is trivial, given that the structures already hold the sine and cosine of δ_i.)

Although equivalent for most small angular differences to the simpler concept of $|\delta_1 - \delta_2|$, the definition above also properly handles the case where one δ is near π and the other near $-\pi$.

5.5 Comparison for `HepRotations`, `HepAxisAngles`, and `HepEulerAngles`

Of the three classes, `HepAxisAngle` has a natural meaning for ordering comparisons, taking advantage of the ordering relation already available for the `UnitVector` axes:

$$(\hat{u}_1, \delta_1) > (\hat{u}_2, \delta_2) \text{ if } \hat{u}_1 > \hat{u}_2 \text{ or } [\hat{u}_1 = \hat{u}_2 \text{ and } \delta_1 > \delta_2] \tag{159}$$

For `Rotation`, we use the ordering induced by its `HepAxisAngle` expression:

$$r_1 > r_2 \text{ iff } r_1.\text{axisAngle}() > r_1.\text{axisAngle}() \tag{160}$$

Extracting the `HepAxisAngle` corresponding to a `HepRotation` is a fairly simple task. However, for testing equality or inequality, the package uses the even easier method of element-by-element equality checking.

For `HepEulerAngles`, rather than laboriously going over to `HepRotations` and then using the induced `HepAxisAngle` comparison, we adapt simple dictionary ordering:

$$\begin{aligned} (\phi_1, \theta_1, \psi_1) > (\phi_2, \theta_2, \psi_2) &\text{ if} \\ \phi_1 > \phi_2 &\text{ or} \\ [\phi_1 = \phi_2 \text{ and } \theta_1 > \theta_2] &\text{ or} \\ [\phi_1 = \phi_2 \text{ and } \theta_1 = \theta_2 \text{ and } \psi_1 = \psi_2] & \end{aligned} \tag{161}$$

5.6 The Rotation Group

Inversion of a `HepRotation` is supported. Since the matrix is orthogonal, the inverse matches the transpose:

$$R.\text{inverse}() \equiv R^{-1} = R^T$$

Multiplication is available in three syntaxes:

$$\tag{162}$$

$$\text{R} = \text{R1 * R2} \Longrightarrow R = R_1 R_2 \tag{163}$$

$$\text{R *= R1} \Longrightarrow R = R R_1 \tag{164}$$

$$\text{R.transform(R1)} \Longrightarrow R = R_1 R \tag{165}$$

To complete the group concept, the identity `HepRotation` can easily be obtained; the default constructor for `HepRotation` gives the identity.

Two sorts of specialized transformations are available. The first transforms by a rotation around an axis:

$$\begin{aligned} R.\text{rotateX}(\delta) &\Longrightarrow R = \text{RotationX}(\delta) R \\ R.\text{rotateY}(\delta) &\Longrightarrow R = \text{RotationY}(\delta) R \\ R.\text{rotateZ}(\delta) &\Longrightarrow R = \text{RotationZ}(\delta) R \end{aligned} \tag{166}$$

The other specialized transformation rotates such that the original X axis becomes a specified new X axis, and similarly for the Y and Z axes. In the specified new axes are labeled $\vec{X}'$, $\vec{Y}'$, $\vec{Z}'$, this transformation is equivalent to:

$$\text{R.rotateAxes}(X', Y', Z') \Longrightarrow R = \begin{pmatrix} X'_x & Y'_x & Z'_x \\ X'_y & Y'_y & Z'_y \\ X'_z & Y'_z & Z'_z \end{pmatrix} R \tag{167}$$

The supplied $\vec{X}'$, $\vec{Y}'$ and $\vec{Z}'$ must be orthonormal; no checking is done, and if the supplied new axes are not orthonormal, the result will be an ill-formed (non-orthogonal) rotation matrix.

5.7 Rectifying Rotations

The operations on HepRotations are such that mathematically, the orthonormality of the representation is always preserved. And methods take advantage of this property. However, a long series of operations could, due to round-off, produce a HepRotation object with a representation that slightly deviates from the mathematical ideal. This deviation can be repaired by extracting the axis and delta for the HepRotation, and freshly setting the axis and delta to those values.

$$\text{R.rectify()} \rightarrow \text{R.set (R.axis(), R.delta())} \tag{168}$$

A technical point: If the rotation has strayed significantly from a true orthonormal matrix, then extracting the axis is not necessarily an accurate process. To minimze such effects, before performing the formal algorithm to extract the axis, the rectify() method averages the purported rotation with the transpose of its inverse. (A true rotaion is identical to the transpose of its inverse). This in principle eliminates errors to lowest order.

6 HepLorentzRotation Class

A HepLorentzRotation may be expressed in terms of a Rotation in the space sector, followed by a pure HepBoost along some direction. Alternatively, it may be expressed as a pure boost followed by a rotation.

In any event, just as for rotations, we use the convention of active transformations changing 4-vectors (rather than transformations of a reference frame). So a boost by β along the in the X direction, for example, would be represented by the matrix

$$\text{LorentzBoost}(\beta) = \begin{pmatrix} \gamma & 0 & 0 & \beta\gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta\gamma & 0 & 0 & \gamma \end{pmatrix} \tag{169}$$

A HepLorentzRotation is an object in its own right; two HepLorentzRotations may be multiplied, tested for near equality, and so forth. Multiplication is done by multiplying the matrix representations of two HepLorentzRotations.

The matrix representing a HepLorentzRotation is orthosymplectic: The last (T) row, constdered as a 4-vector, has $t^2 - |\vec{v}|^2 = 1$, each of rows X, Y, and Z have $t^2 - |\vec{v}|^2 = -1$, and the Minkowski space dot product of any two distinct rows is zero. This leads to a trivial inversion method—simply transpose the matrix and negate any elment with just one of its two indices refering to a space direction.

6.1 Pure Lorentz Boosts

Definitions and and conventions for a pure boost classes match those described in equation (103). Here we will write out the matrix in detail, for a boost specified by $\vec{b}$, a Hep3Vector with magnitude $0 < \beta < 1$ (here we write $\vec{b} = \beta\hat{u}$ and $\gamma \equiv \frac{1}{\sqrt{1-\beta^2}}$):

$$\text{LorentzBoost}(\beta\hat{u}) = \begin{pmatrix} (\gamma-1)u_x^2+1 & (\gamma-1)u_xu_y & (\gamma-1)u_xu_z & \beta\gamma u_x \\ (\gamma-1)u_yu_x & (\gamma-1)u_y^2+1 & (\gamma-1)u_yu_z & \beta\gamma u_y \\ (\gamma-1)u_zu_x & (\gamma-1)u_zu_y & (\gamma-1)u_z^2+1 & \beta\gamma u_z \\ \beta\gamma u_x & \beta\gamma u_y & \beta\gamma u_z & \gamma \end{pmatrix} \tag{170}$$

isNear() and howNear() for LorentzBoost

Though a pure boosts may be specified by a `Hep3Vector` of magnitude less than one, we do not define a nearness measure as the (relative) measure that induced by those `Hep3Vectors`. This is because two `HepBoosts` in the same direction with large but quite different values of γ should be viewed as quite different transformations. The `Hep3Vectors` representing two such boosts would both be very close to the same unit vector. That is, (.9999, 0, 0) and (.999999, 0, 0) are not very similar: The former would boost a muon to 10 GeV, while the latter would boost it to 200 GeV; yet as `Hep3Vectors`, (.9999, 0, 0) is equal to (.999999, 0, 0) within a relative tolerance 10^{-4}.

LorentzBoosts have a natural scale, set by the speed of light, so absolute rather than relative tolerances are appropriate. The correct way to measure nearness is to find the difference between the values of γb. If $\vec{\beta}_1$ and $\vec{\beta}_2$ specify two pure boosts B_1 and B_2 and $\gamma_i \equiv \frac{1}{\sqrt{1-|\vec{\beta}_i|^2}}$ then

$$B_1.\mathrm{isNear}(B_2) \Longleftrightarrow \left|\gamma_1\vec{\beta}_1 - \gamma_2\vec{\beta}_2\right|^2 \leq \epsilon^2 \tag{171}$$

$$B_1.\mathrm{howNear}(B_2) = \left|\gamma_1\vec{\beta}_1 - \gamma_2\vec{\beta}_2\right| \tag{172}$$

$$B_1.\mathrm{distance2}(B_2) = \left|\gamma_1\vec{\beta}_1 - \gamma_2\vec{\beta}_2\right|^2 \tag{173}$$

$$B.\mathrm{norm2}() \equiv B.\mathrm{distance2}(I) = \gamma^2\beta^2 = 1 - \gamma^2 \tag{174}$$

Ordering Comparisons of LorentzBoosts

Any pure boost can be viewed as having a non-negative β, and s direction. For pure boosts in the same direction both with positive β, we wish to order LorentzBoosts according β:

$$\mathrm{LorentzBoost}(\beta_1 > 0, \hat{u}) > \mathrm{LorentzBoost}(\beta_2 > 0, \hat{u}) \Longleftrightarrow \beta_1 > \beta_2 \tag{175}$$

For pure boosts which may be in different directions, we use a comparison condition induced by the `Hep3Vectors` specifying the two boosts. But in order to match the above ordering for identical directions, we must reverse the sense of the naive induced ordering when both vectors are negative. That is, we want to say that the boost (-.2, -.2, -.2) is "more than" the boost (-.1, -.1, -.1).

$$\mathrm{LorentzBoost}(\vec{\beta}_1) > \mathrm{LorentzBoost}(\vec{\beta}_2) \text{ if } \begin{cases} (\vec{\beta}_1 \geq \vec{0} & \text{and} & \vec{\beta}_1 > \vec{\beta}_2) \\ & \text{or} & \\ (\vec{\beta}_2 < \vec{0} & \text{and} & \vec{\beta}_1 < \vec{\beta}_2) \end{cases} \tag{176}$$

6.2 Components of HepLorentzRotations

Beyond the obvious methods returning single components, such as `lt.yt()` there is also the method `rep4x4()`, which returns a struct of type `HepRep4x4`—this has publicly visible data members `xx_`, `xy_`, ..., `tz_`, `tt_`. These methods may also be used for pure `HepBoost` boosts. For those classes, there is also a method `rep4x4Symmetric()`, which returns a ten-element struct `HepRep4x4Symmetric`.

For `HepLorentzRotation`, there are also indexing methods with syntax `lt[i][j]` and `lt(i,j)`. In this syntax, the indices range from 0 to 3, with the time component last. That is, 0 refers to an X component, 1 to Y, 2 to Z, and 3 to T. If R is the four by four representation of a `HepLorentzRotation` L, then:

$$\begin{array}{llll} L[0][0] = R_{xx} & L[0][1] = R_{xy} & L[0][2] = R_{xz} & L[0][3] = R_{xt} \\ L[1][0] = R_{yx} & L[1][1] = R_{yy} & L[1][2] = R_{yz} & L[1][3] = R_{yt} \\ L[2][0] = R_{zx} & L[2][1] = R_{zy} & L[2][2] = R_{zz} & L[2][3] = R_{zt} \\ L[3][0] = R_{tx} & L[3][1] = R_{ty} & L[3][2] = R_{tz} & L[3][3] = R_{tt} \end{array}$$

6.3 Decomposition of Transformations into Boost and Rotation

A `HepLorentzRotation` T may be decomposed either into the form BR or the form RB. Here we will refer to the first form as $T = \acute{B}(T)\grave{R}(T)$ and the second as $T = \acute{R}(T)\grave{B}(T)$.

When decomposing into the product BR, the boost will have the same last column as the `HepLorentzRotation` T. Using the value of γ read off T_{tt}, one can apply equation (170) to find the matrix $\acute{B}(T)$ for that boost. Then

$$\grave{R}(T) = \left[\acute{B}(T)\right]^{-1} T \tag{177}$$

When decomposing into the product RB, the boost will have the same last row as the `HepLorentzRotation` T. Again one can apply equation (170) to find the matrix (this time $\grave{B}(T)$) for that boost. Then

$$\acute{R}(T) = T\left[\grave{B}(T)\right]^{-1} \tag{178}$$

Naively applying the above equations leads to non-neglible round-off errors in the components of the Rotation—of order $3 \cdot 10^{-14}$ if the boost has $\beta = .95$. Since a Rotation representation which is non-orthogonal on that scale would lead to errors in distance measures of more than one part in 10^{-7}, the decompose() method rectifies the Rotation before returning.

6.4 isNear() and howNear() for `HepLorentzRotations`

For `HepLorentzRotations`, the analog of the the `Rotation`measure (151) would be

$$T_1.\text{howNear}(T_2) = \sqrt{4 - \text{Tr}(T_1 T_2^{-1})}$$

This, however, would be a horrible definitition of distance since that trace can equal 4 for very different T_1 and T_2. Instead, we use a definition which matches the definitions (151) and (172) for `HepLorentzRotations` which happen to be pure boosts or pure rotations:

$$\begin{aligned} \text{if } T_1 = B_1 R_1 &\text{ and } T_2 = B_2 R_2 \\ \text{then } T_1.\text{distance2}(T_2) &= B_1.\text{distance2}(B_2) + R_1.\text{distance2}(R_2) \\ T_1.\text{howNear}(T_2) &= \sqrt{T_1.\text{distance2}(T_2)} \end{aligned} \tag{179}$$

The `isNear()` relationship uses this same measure, but when the transformations have boosts that differ by more that ϵ, `isNear()` will be quite a bit quicker since the boost portion of this formula is trivial to compute.

And, as usual, `norm2()` is the squared distance from the identity:

$$T = BR \Longrightarrow T.\text{norm2}() = B..\text{norm2}() + R.\text{norm2}() \tag{180}$$

6.5 Ordering Comparisons of `HepLorentzRotations`

The `( >, >=, <, <= )` comparisons of `HepLorentzRotations` are defined as those induced by the decomposition $T = \acute{B}\grave{R}$, comparing the pure boost part (using (175)) and then, if those are equal, comparing the rotation part (using (159)).

$$
\begin{aligned}
T_1.\mathrm{compare}(T_2) = 1 \text{ if } \acute{B}_1 > \acute{B}_2 \text{ or } \left[\acute{B}_1 = \acute{B}_2 \text{ and } \grave{R}_1 > \grave{R}_2\right] \\
T_1.\mathrm{compare}(T_2) = -1 \text{ if } \acute{B}_1 < \acute{B}_2 \text{ or } \left[\acute{B}_1 = \acute{B}_2 \text{ and } \grave{R}_1 < \grave{R}_2\right] \\
T_1.\mathrm{compare}(T_2) = 0 \text{ if } \acute{B}_1 = \acute{B}_2 \text{ and } \grave{R}_1 = \grave{R}_2
\end{aligned}
\tag{181}
$$

$$
\begin{aligned}
T_1 > T_2 \text{ if } T_1.\mathrm{compare}(T_2) = 1 \\
T_1 < T_2 \text{ if } T_1.\mathrm{compare}(T_2) = -1
\end{aligned}
\tag{182}
$$

Although `==` and `!=` comparisons could use this same comparison algorithm, decomposition is expensive and unnecessary. Instead,

$$
T == S \Longleftrightarrow \begin{cases}
T_{xx} = S_{xx} \wedge & T_{xy} = S_{xy} \wedge & T_{xz} = S_{xz} \wedge & T_{xt} = S_{xt} \wedge \\
T_{yx} = S_{yx} \wedge & T_{yy} = S_{yy} \wedge & T_{yz} = S_{yz} \wedge & T_{yt} = S_{yt} \wedge \\
T_{zx} = S_{zx} \wedge & T_{zy} = S_{zy} \wedge & T_{zz} = S_{zz} \wedge & T_{zt} = S_{zt} \wedge \\
T_{tx} = S_{tx} \wedge & T_{ty} = S_{ty} \wedge & T_{tz} = S_{tz} \wedge & T_{tt} = S_{tt}
\end{cases}
\tag{183}
$$

6.6 The Lorentz Group

Inversion of a `HepLorentzRotation` is supported. Since the matrix is orthosymplectic, the inverse matches the transpose, but with the signs of all components with mixed space and time indices reversed.

Multiplication is available in three syntaxes:

$$
\mathrm{T = T1 * T2} \Longrightarrow T = T_1 T_2 \tag{184}
$$

$$
\mathrm{T *= T1} \Longrightarrow T = T T_1 \tag{185}
$$

$$
\mathrm{T.transform(T1)} \Longrightarrow T = T_1 T \tag{186}
$$

To complete the group concept, the identity `HepLorentzRotation` can easily be obtained; the default constructor for `HepLorentzRotation` gives the identity.

Specialized transformations based on pure rotations and pure boosts are available.

$$
\begin{aligned}
T.\mathrm{rotate}(\delta, \vec{v}) \Longrightarrow T = \mathrm{Rotation}(\delta, \vec{v}) T \\
T.\mathrm{rotateX}(\delta) \Longrightarrow T = \mathrm{RotationX}(\delta) T \\
T.\mathrm{rotateY}(\delta) \Longrightarrow T = \mathrm{RotationY}(\delta) T \\
T.\mathrm{rotateZ}(\delta) \Longrightarrow T = \mathrm{RotationZ}(\delta) T
\end{aligned}
\tag{187}
$$

$$
\begin{aligned}
T.\mathrm{boost}(\vec{\beta}) \Longrightarrow T = \mathrm{HepBoost}(\vec{\beta}) T \\
T.\mathrm{boost}(\beta_x, \beta_y, \beta_z) \Longrightarrow T = \mathrm{HepBoost}(\beta_x, \beta_y, \beta_z) T
\end{aligned}
\tag{188}
$$

$$
\begin{aligned}
T.\text{boostX}(\beta) &\Longrightarrow T = \text{HepBoostX}(\beta)T \\
T.\text{boostY}(\beta) &\Longrightarrow T = \text{HepBoostY}(\beta)T \\
T.\text{boostZ}(\beta) &\Longrightarrow T = \text{HepBoostZ}(\beta)T
\end{aligned}
$$

6.7 Rectifying HepLorentzRotations

The operations on `HepLorentzRotation`s are such that mathematically, the ortosymplectic property of the representation is always preserved. And methods take advantage of this property. However, a long series of operations could, due to round-off, produce a `HepLorentzRotation` object with a representation that slightly deviates from the mathematical ideal. This deviation can be repaired by extracting the boost vector based on row 4, multiplying by the inverse of that boost to form what should ideally be a rotation, rectifying that rotation, and setting the `HepLorentzRotation` ,based on that boost and rotation.

$$
\begin{aligned}
\text{T.rectify()} &\rightarrow \\
\vec{\beta} &= \text{T.row4()} \\
R &= T\ \text{Boost}(-\vec{\beta}) \\
\text{drop time components of R} & \\
&\text{R.rectify()} \\
T &= R\ \text{Boost}(\beta)
\end{aligned}
\tag{189}
$$

7 Names and Keywords

7.1 Symbols in the CLHEP Vector Package

Here we list the keywords that the user may need to be aware are used by the CLHEP Vectors package. Some symbols not listed here are:

- All symbols particular to the ZOOM PhysicsVectors package are listed in the next section. Pure CLHEP users who do not include files fro a ZOOM area need not be concerned with these.
- Method names in class scope, and names accessible only in the scope of a class (such as Rotation::IDENTITY) cannot clash with user names.

Names of functions defined at global scope do not actually pollute the user namespace, as long as one or more of their arguments involves a class defined in this package. These are therefore listed separately here.

```
File Defines                 Class Names
------------                 --------------------------------------------
HEP_THREEVEVCTOR_H       Hep3Vector
HEP_LORENTZVECTOR_H          HepLorentzVector
HEP_ROTATION_H               HepRotation  HepRotationX
                             HepRotationY HepRotationZ
HEP_LORENTZ_ROTATION_H       HepLorentzRotation HepBoost
                             HepBoostX  HepBoostY  HepBoostZ
HEP_ROTATION_INTERFACES_H    Hep4RotationInterface  Hep3RotationInterface
                             HepRep3x3  HepRep4x4  HepRep4x4Symmetric
HEP_EULERANGLES_H            HepEulerAngles
HEP_AXISANGLE_H              HepAxisAngle
```

```
Keywords and Types    Constants      Global Functions
------------------    ---------      ----------------
Tcomponent            X_HAT4    rotationOf
TimePositive          Y_HAT4       rotationXOf
TimeNegative          Z_HAT4       rotationYOf
      T_HAT4      rotationZOf
      X_HAT2    boostOf
      Y_HAT2    boostXOf
                      HepXHat      boostYOf
                      HepYHat      boostZOf
                      HepZHat      inverseOf
```

7.2 Further Symbols Defined in ZOOM Headers

For backward compatibility, ZOOM headers are made available. These typedef classes to the corresponding CLHEP versions, and in the two cases of features which were in ZOOM but are not in CLHEP—`UnitVector` and construction of vectors from spherical coordinates—provide the appropriate features.

These ZOOM features are implemented purely at the header level; there is no separate ZOOM PhysicsVectors library beyond the CLHEP library.

Note that for users including these headers, all symbols starting with `ZMpv` are to be considered as reserved. These therefore do not appear in this list of keywords.

```
File Defines                Class Names
------------                --------------------------------------------
PHYSICSVECTORS_H
SPACEVECTOR_H               SpaceVector
UNITVECTOR_H                UnitVector
LORENTZVECTOR_H             LorentzVector
ROTATION_H                  Rotation RotationX  RotationY RotationZ
LORENTZ_TRANSFORMATION_H    LorentzTransformation  LorentzBoost
                            LorentzBoostX  LorentzBoostY  LorentzBoostZ
EULERANGLES_H               EulerAngles
AXISANGLE_H                 AxisAngle
```

```
Keywords and Types    Constants
------------------    ---------
DEGREES               X_HAT
RADIANS               Y_HAT
ETA                   Z_HAT
```