The Classes in the GenVector Package

M. Fischler

December 13, 2005

Abstract

The GenVector package is intended to represent physicals vectors and transformations (such as rotations and Lorentz transformations) in 2, 3, and (space-time) 4 dimensions, providing models and capabilities attuned to the needs of High Energy Physics codes.

Here we describe each class present, along with all the member functions and associated functions which users can count on.

Contents

1 General Information

The GenVector package provides class templates for classes modeling vectors and transformations.

There is a user-controlled freedom as to how the vector is internally represented. This is expressed by a choice of coordinate system; the coordinate system is supplied as a template parameter when the vector is constructed. A coordinate system can be one of several choices (Cartesian, Polar, and so forth) in 2, 3, or 4 dimensions–thus the choice of coordinate system simultaneously dictates the dimensionallity of the vector.

There is a further degree of control: Each coordinate system is itself a template so that the user can specify the underlying scalar type. For example, `Cartesian3D<double>` is a coordinate system.

The transformations are modeled simple (non-template) classes, using double as the scalar-type. For the purposes of understanding the classes available, the transformations are grouped: Rotations (in two and three dimensions), Lorentz transformations, and Poincare transformations, which are Translation/Rotation combinations. Each group has several members, which may model physically equivalent transformations but with different internal representations. For example, a Rotation may be kept as a 3x3 matrix (`Rotation3D`) or as an axis and angle of rotation (`AxisAngle`).

Some instantiations of the templates are given special names for convenience and brevity. For example, `XYZVector` is typedef-ed to `DisplacementVector< Cartesian3D<double> >`

1.1 Classes (and Class Templates)

The user of the GenVector package normally instantiates and invokes member and other functions involving objects of the following classes:

DisplacementVector3D – Class template (the template parameter is the coordinate system to use) modeling an abstract 3-component direction-and-magnitude vector, not rooted at any particular point.

PositionVector3D – Class template modeling a point in 3-space.

LorentzVector – Class template (the template parameter is the coordinate system to use) modeling a 4-vector in Minkowski space, intended to represent a momentum-energy 4-vector.

DisplacementVector2D – Class template modeling an abstract 2-component direction-and-magnitude vector, not rooted at any particular point.

PositionVector2D – Class template modeling a point in 2-space.

Rotation3D – Class representing a 3-D rotation by a 3×3 orthogonal matrix.

AxisAngle – Class representing a 3-D rotation by a unit vector axis and an angle of rotation.

EulerAngles – Class representing a 3-D rotation by a its three Euler angles.

Quaternion – Class representing a 3-D rotation by the four components of the corresponding quaternion. The Quaternion representation is significantly more efficient when multiplying several rotations, but less effecient when applying a rotation to a vector.

RotationX – Class representing a 3-D rotation about the X axis by the angle of rotation. For efficiency, the sine and cosine of the angle of rotation are held.

RotationY – Class representing a 3-D rotation about the Y axis by the angle of rotation. For efficiency, the sine and cosine of the angle of rotation are held.

RotationZ – Class representing a 3-D rotation about the Z axis by the angle of rotation. For efficiency, the sine and cosine of the angle of rotation are held.

LorentzRotation – Class representing a 4-D rotation (in Minkowski space) by a 4×4 orthosimplectic matrix.

Boost – Class representing a pure Lorentz Boost, which is a special case of LorentzRotation, by the components $(\beta_x, \beta_y, \beta_z)$. For efficiency, γ is held as well.

BoostX – Class representing a Lorentz Boost along the X axis, by β_x. For efficiency, γ is held as well.

BoostY – Class representing a Lorentz Boost along the Y axis, by β_y. For efficiency, γ is held as well.

BoostZ – Class representing a Lorentz Boost along the Z axis, by β_z. For efficiency, γ is held as well.

Rotation2D – Class representing a 2-D rotation by the angle of rotation. For efficiency, the sine and cosine of the angle of rotation are held.

In addition, the following classes are provided so as to allow user control of the coordinate systems used internally by vectors. The headers for these classes are of necessity publicly available, but the only normal usage will be to supply a coordinate system class as a template parameter to a vector class template to specify the coordinates (and Scalar type) to use.

Cartesian3D The usual Cartesian (x, y, z).

Polar3D Spherical polar coordinates (r, θ, ϕ).

CylindricalEta3D Cylindrical coordinates (ρ, η, ϕ), where $\eta \equiv \log(\frac{z+\sqrt{\rho^2+z^2}}{\rho})$. Special handling is done for vectors along the Z axis ($\rho = 0$) to be able to recover the vector in other coordinate systems.

PxPyPzE4D The usual "Cartesian" Minkowski coordinates, with time-like component last. Although the name implies this models a momentum-energy 4-vector, it can equally well be thought of as (x, y, z, t).

PtEtaPhiE4D $(p_\perp, \eta, \phi, E)$. The transverse momentum acts as ρ, and the rapidity, azimuthal angle, and energy complete the description of a 4-Vector. This system has been called 4-D CylindricalEta coordinates.

PtEtaPhiM4D $(p_\perp, \eta, \phi, m)$. The mass is used in place of the energy. This system can be thought of as the outer product of 3-D CylindricalEta coordinates, with an arbitrary mass value.

EEtaPhiM4D (E, η, ϕ, m). The energy is used in place of the transverse momentum. This completes a set of descriptions involving (η, ϕ) to specify the direction of the momentum: On can specify any two of energy, mass, or magnitude of momentum to specify the momentu-energy 4-vector.

Cartesian2D The usual Cartesian (x, y).

Polar2D Polar (r, ϕ).

Note that the 4-D coordinate systems make use of the relativistic relation among energy, mass and momentum, taking $c = 1$.

1.2 Properties Common to Every Class

By "every class in this package" we include, of course, every class obtained by instantiating some template in the GenVector package. These include coordinate systems, vectors, and transformations.

Every class and free method is in the namespace `ROOT::Math::`.

Every class is default constructible, copy constructible, and copy assignable.

Every class is dependent on some real type to be used for scalars, and contains a typedef `Scalar` to publish that type.

Every class is equality comparable (for equals and not-equals) against objects of the identical type. The usual warnings about comparing two floating point numbers for equality apply. In addition, if the two objects have different component values which represent equivalent physical properties (for example, two `AxisAngle`s with both the axes and angles negative of one another), the objects will compare as being unequal.

Every class has operators `<<` and `>>` to ostreams and from istreams respectively, which produce and read back human-readable output representing the vectors or transformations. In addition, every class responds to the one-shot manipulator `ROOT::Math::machine_readable` to produce (and read back) data such that the result after read-back is portably identical to the instance output.

```
using namespace ROOT::Math;
{std::ofstream os ("filename");
 XYZVector v (1, 2, 3);
 v /= 3;
 os << v;
 os << machine_readable << v;
}
{std::ifstream is ("filename");
 XYZVector v;
 is >> v;
 assert(v != XYZVector(1,2,3)/3;
 is >> machine_readable >> v;
 assert(v == XYZVector(1,2,3)/3;
}
```

In the example above, the first input of `v` would not give an exact reproduction because the human readable output was rounded at 6 digits (or whatever precision had been set for floating point output). The second input of `v` would give an exact reproduction.

1.3 Naming Conventions

As part of the LCG core math library, the GenVector package adheres to the prescribed naming and namespace conventions, with (approved) exceptions as described here:

- Every class, function, manipulator, or other symbol defined in the package is in the `ROOT::Math::` namespace.
- Member function names start with an upper-case letter, as dictated by the ROOT naming conventions.
- Data members of classes are kept private, and start with the letter `f` followed by an upper-case letter.

1.4 Behavior in Exceptional Circumstances

As dictated by the core math library architects, the GenVector package avoids throwing exceptions in cases where sensible behavior can be framed in terms of returning or producing infinities, NAN's, or other special values. For example, when dividing a vector by a scalar, the implementation does not check for a zero divisor; the rules of IEEE arithmetic provide for sensible infinities to be placed in the answer. As a more complex example, if a vector in clyndrical eta coordinates is constructed from a vector lying on the Z axis, the value of eta produced will be a special value, impossible to produce from any non-Z-axis vector, allowing for recovery of the correct Z value.

When sensible behavior cannot be framed, under some circumstances the package suggests throwing a `GenVector_exception`. This exception inherits from `std::runtime_error` and holds an explanatory string which can be queried by the `what()` method. However, the throw is done by calling `Throw(GenVector_exception&)`, which by default does nothing and returns. Then some non-catastrophic continuing action is taken. To enable actual throws in these circumstances, the user invokes `GenVector_exception::EnableThrow()`.

Implementation of member functions may detect certain operations which are logically erroneous. Cases where violation of the proper conditions does not necessarily lead to catastrophic states are dealt with by checking via an `assert`; these checks can be removed in non-debug builds.

1.5 GenVector Utilities

There are provided separate functions which perform less fundamental actions on the various GenVector classes. These utilities are provided via separate headers, and are described in a separate document.

1.6 Headers and Dependencies

Every class has a pair of corresponding headers (with extension `.h`): A header containing the full class or class template definition, matching the name of the class (for example, `Rotation3D.h`) and a "forwarding header" with the letters `fwd`, containing only declarations of the class or class template (for example, `Rotation3Dfwd.h`). Headers freely include each other's forwarding headers, but inclusion of the full headers is organized so as to minimize coupling and avoid circular dependencies.

- The various coordinate system classes include no other user-relevant headers.
- The various vector classes include the Cartesian coordinate system class of the appropriate dimensions. If a user wishes to use any other specific coordinate system to instantiate a vector class, the relevant coordintate system header must be included in the user code.
- `PositionVector3D` includes the header for `DisplacementVector3D` because operations involving mixtures of the two are expressed in the `PositionVector3D` header. This choice of dependency is not arbitrary: The difference between two position vectors is a displacement vector, thus the dependency in this direction is necessary, whilst the difference between two displacement vectors remains a displacement vector.
- The various rotation and transformation classes include each of the vector classes in their respective dimensions and all higher dimensions, so that the operator of application of the rotation to each type of vector can be expressed.

2 Coordinate Systems

The coordinate system class templates are provided primarily to allow the user to specify which coordinates should be used by a vector class for its internal representation.

These are templates with the type of real variable to use for the coordinates specified as a parameter. This real type is published as a typedef `Scalar`, and defaults to double.

There are three groupings of coordinate systems: 2-D coordinates, 3-D coordinates, and 4-D coordinates.

Should the user wish to use these classes independantly, they have common member functions as described below.

Every coordinate system class in the N-D group is constructible from N `Scalar`s representing the N coordinates to be used.

Every coordinate system class in the N-D group has member function `SetCoordinates()` and `GetCoordinates()`. These are present in two signatures: A form taking N `Scalars` representing the N coordinates to be used, and a form taking a an array which is assumed (without checking) to contain N `Scalar`s.

Every coordinate system class is constructible and assignable from an instance of any other coordinate system in its group. This is implemented without introducing cross-dependencies other than forwarding headers.

Every coordinate system class has a member function `Scale(Scalar a)` which scales the vector by multiplying by `a`. For example, in polar coordinates, scaling by two would mean doubling the value of r. Every coordinate system class also has a member function `Negate( )` which turns the vector into its additive inverse (this is not, except in Cartisian coordnates, the same as negating each coordinate!).

2.1 3-D Coordinate Systems

The 3-D coordinate systems in the GenVector package are:

Cartesian3D (x, y, z) Cartesian coordinates.

Polar3D (r, θ, ϕ) spherical coordinates.

CylindricalEta3D (ρ, η, ϕ) cylindrical eta coordinates ($\eta \equiv \tan^{-1}(\rho/z)$).

Each coordinate system class in the 3-D group contains the following methods returning a single real of the `Scalar` type established for that class, to obtain information about a component or property of a vector:

X( )	
Y( )	
Z( )	
R( )	
Theta( )	
Phi( )	
Rho( )	
Eta( )	
Mag2( )	$\equiv r^2$
Perp2( )	$\equiv \rho^2$
x( )	$\equiv$ X( )
y( )	$\equiv$ Y( )
z( )	$\equiv$ Z( )

Each coordinate system class in the 3-D group contains three member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other two unchanged:

Cartesian3D	has `SetX( )`, `SetY( )`, `SetZ( )`.
Polar3D	has `SetR( )`, `SetTheta( )`, `SetPhi( )`.
CylindricalEta3D	has `SetRho( )`, `SetEta( )`, `SetPhi( )`.

2.2 4-D Coordinate Systems

The 4-D coordinate systems in the GenVector package always represent vectors in Minkowski space. They are intended to model momentum-energy 4-vectors, and depending on the choice of coordinate system, the components will represent properties (such as angles and transverse magnitudes) of the momentum, energy or transverse energy, and/or mass. (Of course the properties are chosen such that a vector in any 4-D coordinate system represents one unique (p_x, p_y, p_z, E) 4-vector.)

The 4-D coordinate systems in the GenVector package are:

PxPyPzE4D	The standard "Cartesian" (p_x, p_y, P_z, E) relativistic 4-vector coordinates.
PtEtaPhiE4D	$(p_\perp, \eta, \phi, E)$ "4-D cylindrical eta" coordinates ($\eta \equiv \tan^{-1}(p_\perp/z)$).
PtEtaPhiM4D	$(p_\perp, \eta, \phi, m)$ coordinates.
EEtaPhiM4D	(E, η, ϕ, m) coordinates.

Each coordinate system class in the 4-D group has a constructor and `SetCoordinates( )` and `GetCoordinates( )` method taking a 3-vector for the spatial momentum coordinates followed by a Scalar energy. The 3-vector object have implement member functions $x(), y()$, and $z()$; it would normally be a `DisplacementVector`. In this form of constructor component access, if the timelike component in the 4-D coordinate system is kept as something other than the energy (for example, as the transverse energy or the mass) the appropriate physics is applied to do the conversion.

Each coordinate system class in the 4-D group contains the following methods returning a single real of the `Scalar` type established for that class, to obtain information about a component or property of a momentum-energy 4-vector:

Px( )	
Py( )	
Pz( )	
E( )	
M( )	$\equiv \sqrt{m^2}$
M2( )	$m^2 \equiv E^2 - p^2$
P( )	$\lvert p\rvert \equiv \sqrt{p_x^2 + p_y^2 + p_z^2}$
P2( )	$\equiv p^2$
Pt( )	transverse momentum $p_\perp \equiv \sqrt{p_x^2 + p_y^2}$
Pt2( )	$\equiv p_\perp^2$
Mt( )	transverse mass $m_t \equiv \sqrt{m^2 + p_\perp^2}$ with sign matching m
Mt2( )	$\equiv m_t^2$
Et( )	transverse energy $E_t \equiv p_\perp \sqrt{1 + m^2/p^2}$
Et2( )	$\equiv E_t^2$
Phi( )	
Theta( )	
Eta( )	$\equiv \tan^{-1}(p_\perp/p_z)$

The following are alternative names for these accessors:

X( )	$\equiv p_x$

Y( )	$\equiv p_y$
Z( )	$\equiv p_z$
T( )	$\equiv E$
R( )	$\equiv \lvert p\rvert$
Mag( )	$\equiv m$
Mag2( )	$\equiv m^2$
Rho( )	$\equiv p_\perp$
Perp( )	$\equiv p_\perp$
Perp2( )	$\equiv p_\perp^2$
x( )	$\equiv p_x$
y( )	$\equiv p_y$
z( )	$\equiv p_z$
t( )	$\equiv E$

Each coordinate system class in the 4-D group contains three member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other two unchchanged.
`PxPyPzE4D` has `SetPx( )`, `SetPy( )`, `SetPz( )`, `SetE( )`. `PtEtaPhiE4D` has `SetPt( )` `SetEta( )`, `SetPhi( )` `SetE( )`. `PtEtaPhiM4D` has `SetPt( )` `SetEta( )`, `SetPhi( )` `SetM( )`. `EEtaPhiM4D` has `SetEta( )`, `SetPhi( )` `SetM( )`, `SetE( )`.

2.3 2-D Coordinate Systems

The 2-D coordinate systems in the GenVector package are:

Cartesian2D	(x, y, z) Cartesian coordinates.
Polar2D	(r, ϕ) spherical coordinates.

Each coordinate system class in the 2-D group contains the following methods returning a single real of the `Scalar` type established for that class, to obtain information about a component or property of a vector:

X( )	
Y( )	
R( )	
Phi( )	
Mag2( )	$\equiv r^2$
x( )	$\equiv x$
y( )	$\equiv y$

Each coordinate system class in the 2-D group contains three member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other one unchanged. `Cartesian2D` has `SetX( )`, `SetY( )`. `Polar2D` has `SetR( )`, `SetPhi( )`.

3 Vectors

The vector class templates are provided to represent vectors (in the physics sense) in 2, 3, and Minkowski space 4 dimensions. Each template takes, as its sole template parameter, the coordinate system which the user desired be used to represent vectors. As a side effect, since coordinate systems themselves are tempated off the type of real number to be used as scalars, the user is also in control of whether the vector class instantiated uses floats, doubles, or long doubles.

In two and three dimensions, the package makes a distinction between vectors representing positions relative to some given origin, and vectors representing displacements. The distinction comes in what operations are meaningful. For example, it is physically meaningless to add a position to another position, yet it is absolutely useful to add two displacements.

In four dimensions, since the important use of the vector is to represent a momentum-energy 4-vector, operations which make sense in that context are provided.

Every vector class publishes a typedef for `CoordinateType` which allows user code to determine the coordinate system specified, as well as a typedef for `Scalar` (which will match the `Scalar` type for the coordinate system).

There are various ways of constructing each vector class; in all cases, the set of assignment operators matches the set of constructors taking single arguments.

The vector classes are each equality comparable (`operator ==` and `operator !=` but only with objects or the identical class (including coordinate system used). Checks for equivalence of two vectors in different coordinate systems can be done by converting one to the system of the other.

None of the vector classes have methods to rotate the vector. To rotate a vector, one instantiates the desired rotation object, and applies that to the vector.

3.1 DisplacementVector3D and PositionVector3D

The `DisplacementVector3D` class template models an abstract 3-component direction-and-magnitude displacement vector. The `PositionVector3D` class template models a point or position in 3-space.

Headers XYZVector.h and XYZPoint.h are provided. These simply include DisplacementVector3D.h and PositionVector3D.h, respectively, along with Cartesian3D.h, and provide typedef aliases:

```
  typedef DisplacementVector3D< Cartesian3D<double> > XYZVector;
  typedef PositionVector3D    < Cartesian3D<double> > XYZPoint;
```

This gives managable names to these commonly used forms (double precision, in Cartesian coordinates).

3.1.1 Construction and Coordinate Access

Instances of either of these classes are default constructible (the zero vector) and constructible from 3 `Scalar`s representing the coordinates to be used; the meaning of the coordinates depends, of course, on the coordinate system specified.

Templated constructors are provided to construct instances of either of the 3-D vector classes from an arbitrary "foreign" vector. The precondition imposed on this foriegn vector is that it have `const` methods `x()`, `y()`, and `z()`. In particular, the CLHEP `ThreeVector` class meets this condition, so these vector classes can be constructed from the CLHEP class.

Explicit constructors for `DisplacementVector3D` from an arbitrary `DisplacementVector3D`, and for `PositionVector3D` from an arbitrary `PositionVector3D` or `DisplacementVector3D`, are provided (the other vector may use different coordinate system). While these cases would be handled properly by the constructors from arbitrary foreign vectors, that route would always involved conversion to Cartesian coordinates and then to the desired coordinates. These constructors can take advantage of ways to go directly from one coordinate system to another.

The 3-D vector classes have three signatures of SetCoordinates methods, allowing the user to either directly provide the three coordinates, to provide a C-style array of Scalars, or provide iterators that behave like an array of coordinates.

```
void SetCoordinates(Scalar a, Scalar b, Scalar c);
void SetCoordinates(Scalar src[]);
template<class IT>
void SetCoordinates(IT begin, IT end);
```

As is the case for constructors taking three coordinates: the meaning of each coordinate supplied depends on which coordinate system was specified.

Corresponding GetCoordinates methods are present:

```
void GetCoordinates(Scalar& a, Scalar& b, Scalar& c) const;
template<class IT>
void GetCoordinates(Scalar dst[]) const;
void GetCoordinates(IT begin, IT end) const;
```

A method to set the vector given X, Y, and Z coordinates (which will be transformed into the coordinate system used for that vector) is present:

```
void SetXYZ(Scalar& x, Scalar& y, Scalar& z);
```

Each of these classes has the coordinate access methods shared by all the 3-D coordinate systems (repeated here for convenience):

X( )	
Y( )	
Z( )	
R( )	
Theta( )	
Phi( )	
Rho( )	
Eta( )	
Mag2( )	$\equiv r^2$
Perp2( )	$\equiv \rho^2$
x( )	$\equiv x$
y( )	$\equiv y$
z( )	$\equiv z$

In addition, lower-case spellings of all the above member functions are provided, to allow code that expects a CLEHP ThreeVector to use these classes in its place.

r( )

theta( )
phi( )
rho( )
eta( )
mag2( )
perp2( )

Each 3-D vector class contains three member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other two unchanged. The set of such functions applicable depends, of course, on the coordinate system in use:

Cartesian3D	has `SetX( )`, `SetY( )` and `SetZ( )`.
Polar3D	has `SetR( )`, `SetTheta( )` and `SetP{hi( )`.
CylindricalEta3D	has `SetRho( )`, `SetEta( )` and `SetPhi( )`.

Attempts to use a set method which is not present in the specified coordinate system will not compile.

3.1.2 DisplacementVector3D Manipulation

`DisplacementVector3D` has a const member function `Unit()` (along with the compatibility spelling `unit()` which returns a unit vector in the direction of this vector.

The dot and cross products of two `DisplacementVector3D`s, or indeed of a `DisplacementVector3D` with any object having methods `x(), y(),` and `z()`, are provided:

```
template< class OtherVector >
   Scalar Dot( const  OtherVector & v) const;
template <class OtherVector>
   DisplacementVector3D Cross( const OtherVector & v) const;
```

Because there is ambiguity over what the overload of `operator*` of two vectors ought to mean, this package provides the unambiguous `Dot()` and `Cross()` methods and does not provide `operator*(const DisplacementVector3D&)` as an alias for either.

Two `DisplacementVector3D`s may be added or subtracted, resulting in a third `DisplacementVector3D`. Operators `+=` and `-=` have the obvious meanings.

A `DisplacementVector3D` may be multiplied or divided by a `Scalar`, with the `Scalar` appearing on the left or right for multiplication. Operators `*=` and `/=` have the obvious meanings. The unary minus operator has the same effect as multiplication by -1.

3.1.3 PositionVector3D Manipulation

The dot and cross products of a `PositionVector3D` with a verb$DisplacementVector3D$, or of almost and any object having methods `x(), y(),` and `z()`, are provided:

```
template< class OtherVector >
   Scalar Dot( const  OtherVector & v) const;
template <class OtherVector>
   DisplacementVector3D Cross( const OtherVector & v) const;
```

However, the notion of the dot or cross products of two points is meaningless, so the dot and cross products of two `PositionVector3D`s are made private (thus forbidden).

A `DisplacementVector3D`s may be added to a `PositionVector3D` (in either order) or subtracted from a `PositionVector3D`; the result is a `PositionVector3D`. Operators `+=` and `-=` taking a `DisplacementVector3D` as the added or subtracted vector have the obvious meanings.

Subtracting one `PositionVector3D` from another `PositionVector3D` is valid: The result is a `DisplacementVector3D`.

All other vector additions and subtractions involving at least one `PositionVector3D` are meaningless and hence not supplied. In particular, two `PositionVector3D`s cannot be added together, and one cannot subtract a `PositionVector3D` from a verb$DisplacementVector3D$.

The notion of multiplying by a scalar is not meaningful, so such methods as scalar multiplication and division, `Unit()`, and the unary minus operator are not present for `PositionVector3D`.

3.2 LorentzVector

The `LorentzVector` class template models an abstract 4-component vector in Minkowski space. The vector is intended to meet the important use-case of modeling a momentum-energy 4-vector.

The header XYZTVector.h is provided. This simply includes LorentzVector.h and Px-PyPzE4D.h, and provided a typedef alias:

```
typedef LorentzVector< Cartesian3D<double> > XYZTVector;
```

This gives a managable name to this commonly used form (double precision, in rectilinear coordinates).

3.2.1 Construction and Coordinate Access

Instances of `LorentzVector` classes are default constructible (the zero vector) and constructible from 4 `Scalar`s representing the coordinates to be used; the meaning of the coordinates depends, of course, on the coordinate system specified.

Templated constructors are provided to construct instances from an arbitrary "foreign" 4-vector. The precondition imposed on this foriegn vector is that it have `const` methods `x()`, `y()`, `z()`, and `t()`. In particular, the CLHEP `HepLorentzVector` class meets this condition, so `LorentzVector` can be constructed from an object of that CLHEP class.

Explicit constructors for `LorentzVector` from an arbitrary `LorentzVector` are provided (the other vector may use a different coordinate system). While these cases would be handled properly by the constructors from arbitrary foreign vectors, that route would always involved conversion to rectilinear coordinates and then to the desired coordinates. These constructors can take advantage of ways to go directly from one coordinate system to another.

`LorentzVector` has three signatures of SetCoordinates methods, allowing the user to either directly provide the four coordinates, to provide a C-style array of 4 Scalars, or provide iterators that behave like an array of coordinates.

```
void SetCoordinates(Scalar a, Scalar b, Scalar c, Scalar d);
void SetCoordinates(Scalar src[]);
template<class IT>
void SetCoordinates(IT begin, IT end);
```

As is the case for constructors taking four coordinates: the meaning of each coordinate supplied depends on which coordinate system was specified.

Corresponding GetCoordinates methods are present:

```
void GetCoordinates(Scalar& a, Scalar& b, Scalar& c, Scalar& d) const;
template<class IT>
void GetCoordinates(Scalar dst[]) const;
void GetCoordinates(IT begin, IT end) const;
```

A method to set the `LorentzVector` given X, Y, Z, and T coordinates (which will be transformed into the coordinate system used for that vector) is present:

```
void SetXYZT(Scalar& x, Scalar& y, Scalar& z);
```

Each of these classes has the coordinate access methods shared by all the 4-D coordinate systems (repeated here for convenience):

Px( )	
Py( )	
Pz( )	
E( )	
P( )	$\equiv \sqrt{p_x^2 + p_y^2 + p_z^2}$
M( )	$\equiv \sqrt{E^2 - P^2}$. The positive quare root is used unless the coordinate system has a mass as one component, in which case M may explicitly be negative.
M2( )	$\equiv m^2$
P2( )	$\equiv p^2$
Pt( )	transverse momentum $p_\perp \equiv \sqrt{p_x^2 + p_y^2}$
Perp2( )	$\equiv p_\perp^2$
Mt( )	transverse mass $m_t \equiv \sqrt{m^2 + p_\perp^2}$ with sign matching m
Mt2( )	$\equiv m_t^2$
Et( )	transverse energy $E_\perp \equiv E p_\perp / \vert p \vert$
Et2( )	$\equiv E_t^2 =$
Phi( )	
Theta( )	
Eta( )	$\equiv \tan^{-1}(p_\perp / p_z)$

An additional coordinate access method is:

Vect()	returns a `DisplacementVector3D< Cartesian3D<Scalar> >` containing the spatial components (p_x, p_y, p_z).

Each `LorentzVector` class contains three member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other three unchanged. The set of such functions applicable depends, of course, on the coordinate system in use:

PxPyPzE4D	has `SetPx( )`, `SetPy( )`, `SetPz( )`, and `SetE( )`.
EEtaPhiM4D	has `SetEta( )`, `SetPhi( ) SetM( )`, and `SetE( )`.

PtEtaPhiM4D	has `SetPt( )` `SetEta( )`, `SetPhi( )` and `SetM( )`.
PtEtaPhiE4D	has `SetPt( )` `SetEta( )`, `SetPhi( )` and `SetE( )`.

Attempts to use a set method which is not present in the specified coordinate system will not compile.

In addition, certain alternative and lower-case spellings of the above member functions are provided, to allow code that expects a CLEHP LorentzVector to use `XYZTVector` in its place.

X( )	$\equiv P_x$
Y( )	$\equiv P_y$
Z( )	$\equiv P_z$
T( )	$\equiv E$
R( )	$\equiv P$
Rho( )	$\equiv P_\perp$
x( )	$\equiv P_x$
y( )	$\equiv P_y$
z( )	$\equiv P_z$
t( )	$\equiv E$
px( )	$\equiv P_x$
py( )	$\equiv P_y$
pz( )	$\equiv P_z$
e( )	
r( )	
theta( )	
phi( )	
rho( )	
eta( )	
perp2( )	
mag2( )	≡ M2( )
mag( )	≡ M( )

3.2.2 LorentzVector Manipulation

The dot product of two `LorentzVector`s, or of a verb$LorentzVector$ with any object having methods `x(), y(), z(),` and `t()`, is provided:

```
template< class Other4Vector >
   Scalar Dot( const  Other4Vector & v) const;
```

This dot product is taken using the $(---+)$ metric. Every physical (timelike or lightlike) momentum-energy vector will have a non-negative dot product with itself (which, in fact, will be equal to `M2( )`).

Also provided are the invariant mass squared and invariant mass of the combination of this 4-vector and another:

```
template< class Other4Vector >
   Scalar InvariantMass2( const  Other4Vector & v) const;
template< class Other4Vector >
   Scalar InvariantMass2( const  Other4Vector & v) const;
```

Two `LorentzVector`s may be added or subtracted, resulting in a third `LorentzVector`. Operators `+=` and `-=` have the obvious meanings.

A `LorentzVector` may be multiplied or divided by a `Scalar`, with the `Scalar` appearing on the left or right for multiplication. Operators `*=` and `/=` have the obvious meanings. The unary minus operator has the same effect as multiplication by -1. Note that this is not the same as multiplying each coordinate by -1.

3.2.3 Relativistic Properties of LorentzVectors

A limited set of common relativistic properties is provided. The first two return Scalars involving rapidity:

Rapidity( )	The classical rapidity $\frac{1}{2}\log\frac{E+p_z}{E-p_z}$. This is less useful than the pseudorapidity η but still sometimes required.
ColinearRapidity( )	The rapidity in the direction of motion $\tanh^{-1}(\|\vec{p}\|/E)$.

Then there are three functions returning `bool` answers about the nature of the momentum-energy 4-vector:

isTimelike( )	The momentum-energy 4-vector describing an ordinary particle is timelike.
isSpacelike( )	The momentum-energy 4-vector describing a tachyonic particle is spacelike.
isLightlike( )	The momentum-energy 4-vector describing a photon is lightlike. Because roundoff will often prevent exact equality between the energy and the square of the momentum, this member function accepts an optional argument to specify tolerance (an acceptable difference level, relative to the value of the energy). The tolerance defaults to 100 times the machine epsilon for the Scalr type used.

Then there are methods to learn about boosts to various center-of-mass frames. Since the `LorentzVector` class does not know about the `Boost` class, any boost answer is provided as a cartesian displacement 3-vector (using the same Scalar type as the `LorentzVector`) representing $\vec{\beta}$. Such a vector is well-suited for constructing a `Boost`. In the list below, we will use the alias `BetaVector` for that type.

```
typedef DisplacementVector3D< Cartesian3D<Scalar> > BetaVector;
BetaVector BoostToCM ( ) const;
template <class Other4Vector>
BetaVector BoostToCM ( const Other4Vector& v ) const;
```

For example, `BoostToCM()` provides a beta vector which, if supplied to construct a `Boost`, would yield a `Boost` which would bring this 4-vector into its rest frame. Asuming the 4-vector is timelike, β-vectors will be of length less than 1.

Finally, there are provided global functions taking a std::vector of `LorentzVector`s (necessarily using the same coordinate systems since they are all together in a std::vector), and returning properties of the aggregate of those 4-vectors:

```
template <class L>
typename L::Scalar InvariantMass2 ( std:vector<L> & V ) const;
template <class L>
BetaVector BoostToCM ( std:vector<L> & V ) const;
```

3.2.4 Peculiar LorentzVectors

Because of the nature of Minkowski coordinates (with a non-positive-definite metric) and the fact that some coordinate systems specify a mass directly, it is possible to have "pathological" `LorentzVectors`. These can have the following peculiar features:

Tachyonic 4-vectors In the (p_x, p_y, p_z, E) coordinate system, one can easily specify a 4-vector whose invariant mass squared $m^2 \equiv E^2 - |p|^2$ is negative. Such a 4-vector is perfectly fine when viewed as a point in spacetime, but when taken as a momentum-energy 4-vector it represents a tachyonic particle. Similarly, in the $(p_\perp, \eta, \phi, E)$ coordinate system a 4-vector can be constructed supplying a $(p_\perp, \eta)$ combination such that the momentum is greater than its energy. Such a 4-vector again represents a tachyonic particle. If the mass of a tachyonic 4-vector is requested (member function `M()`) the package attempts to throw; if this is disabled (as will be the case by default) then it reverses sign but returns the negative square root value. A similar action is taken if the transverse mass (`Mt()`) of a tachyonic 4-vector is requested and that happens to be imaginary.

Negative Mass In either the (E, η, ϕ, m) or the $(p_\perp, \eta, \phi, m)$ coordinate system a 4-vector can be constructed supplying a negative mass. This causes no serious problems, but sequences of operations which might be expected to propagate that negative mass to other negative masses may not behave in that way. Two such situations come to mind: If a negative-mass 4-vector is converted to (p_x, p_y, p_z, E) coordinates and that is converted back to the original coordinate system, the mass will now be positive. And if the invariant mass function is taken, supplying 4-vectors with negative masses, the result will still be non-negative.

Also, for $(p_\perp, \eta, \phi, m)$ coordinates, if a 4-vector has a negative mass, its transverse mass, energy, and transverse energy will be reported as negative. (The energy must report as negative, otherwise if $\vec{v_2}$ is expressed in $(p_\perp, \eta, \phi, m)$ coordinates, then $\vec{v_1} - \vec{v_2} \neq \vec{v_1} + (-\vec{v_2})$).

Negative Energy In the (E, η, ϕ, m) coordinate system a 4-vector can be constructed supplying a negative energy. As is the case for negative mass, If such a 4-vector is converted to other coordinates and back, the energy will now be positive. And if energy is explicitly set to be negative, transverse energy will be reported as negative.

Imaginary Momentum(!) In the (E, η, ϕ, m) coordinate system a 4-vector can be constructed supplying mass greater than its energy. The constructor for such an object will attempt to throw; the imaginary-momentum 4-vector will come into existance only if GenVector throws are disabled. In such a case, p, $p_\perp$ and possibly E_t are formally imaginary, and if any of those are requested, a throw is attemped and calls to `std::sqrt()` on a negative argument can occurs. In addition, `P2()` and possibly `Et2()` are capable of returning negative values. And if such a 4-vector is converted to either of the other coordinate systems (where momentum values are needed) a throw is attempted and calls to `std::sqrt()` on a negative arguments can occur.

3.3 DisplacementVector2D and PositionVector2D

The `DisplacementVector2D` class template models an abstract 3-component direction-and-magnitude displacement vector.

The header XYVector.h is provided to include DisplacementVector2D.h along with Cartesian2D.h, and provide a typedef alias:

```
typedef DisplacementVector2D<Cartesian2D> XYVector;
```

3.3.1 Construction and Coordinate Access

Instances of `DisplacementVector2D` are default constructible (the zero vector) and constructible from 2 `Scalar`s representing the values of the coordinates.

Templated constructors are provided to construct instances of either of the 2-D vector classes from an arbitrary "foreign" 2-vector. The precondition imposed on this foriegn vector is that it have `const` methods `x()` and `y()`.

There are three signatures of SetCoordinates methods, allowing the user to either directly provide the three coordinates, to provide a C-style array of Scalars, or provide iterators that behave like an array of coordinates.

```
void SetCoordinates(Scalar a, Scalar b);
void SetCoordinates(Scalar src[]);
template<class IT>
void SetCoordinates(IT begin, IT end);
```

As is the case for constructors taking two coordinates: the meaning of each coordinate supplied depends on which coordinate system was specified.

Corresponding GetCoordinates methods are present:

```
void GetCoordinates(Scalar& a, Scalar& b) const;
template<class IT>
void GetCoordinates(Scalar dst[]) const;
void GetCoordinates(IT begin, IT end) const;
```

A method to set the vector given X, Y coordinates (which will be transformed into polar coordinates if the vector uses that coordinate system) is present:

```
void SetXY(Scalar& x, Scalar& y);
```

`DisplacementVector2D` has the coordinate access methods shared by the 2-D coordinate systems (repeated here for convenience):

X( )	
Y( )	
R( )	
Phi( )	
Mag2( )	$\equiv r^2$
x( )	$\equiv x$
y( )	$\equiv y$

In addition, lower-case and laternate spellings of all the above member functions are provided, to allow code that expects a CLEHP ThreeVector to use these classes in its place.

r( )
phi( )
mag2( )

Each `DisplacementVector2D` class contains two member functions taking a `Scalar` agument, to set the value of one individual component used in the coordinate representation, leaving the other one unchanged. The set of such functions applicable depends, of course, on the coordinate system in use:

Cartesian2D	has `SetX( )`, and `SetY( )`.
Polar2D	has `SetR( )`, and `SetP{hi( )`.

Attempts to use a set method which is not present in the specified coordinate system will not compile.

3.3.2 DisplacementVector2D Manipulation

`DisplacementVector2D` has a const member function `Unit()` (along with the compatibility spelling `unit()` which returns a unit vector in the direction of this vector.

The dot product of two `DisplacementVector2D`s is provided:

```
template< class OtherVector >
   Scalar Dot( const  OtherVector & v) const;
template <class OtherVector>
   DisplacementVector2D Cross( const OtherVector & v) const;
```

The cross product of two `DisplacementVector2D`s is not a meaningful operation unless we allow for knowledge of a third dimension. Nonetheless, the scalar magnitude of that cross product, which is $|a||b|\sin\theta_{ab}$ is often useful:

```
template <class OtherVector>
   DisplacementVector2D Cross( const OtherVector & v) const;
```

Two `DisplacementVector2D`s may be added or subtracted, resulting in a third `DisplacementVector3D`. Operators `+=` and `-=` have the obvious meanings.

A `DisplacementVector2D` may be multiplied or divided by a `Scalar`, with the `Scalar` appearing on the left or right for multiplication. Operators `*=` and `/=` have the obvious meanings. The unary minus operator has the same effect as multiplication by -1.

4 Transformations

The transformation classes are provided to represent active transformations which can be applied to vectors of the appropriate dimension. The classes include various flavors of 3-D and 2-D rotations, and of "Lorentz rotations" transforming 4-vectors in Minkowski space. In 3 dimensions, there are also transformations that allow for translations and/or inhomogeneous scaling.

We will refer to transformations as being in the same group if the represent the same notion in the same number of dimensions. For example, `Rotation3D` and `AxisAngle` are in the same group, but neither `Transformation3D` nor `LorentzRotation` are in that group.

The transformation classes are not class templates: Different ways to represent a rotation (for example) are expressed as different classes.

The Scalar type used for the component data in all representations is double. Every transformation class publishes a typedef for `Scalar` which will be `double`.

There are various ways of constructing each transformation class. In all cases, the set of assignment operators matches the set of constructors taking single arguments. There is a collection of `SetComponents()` methods matching the ways to construct the matrix, and `GetComponents()` methods matching those.

The transformation classes are each equality comparable (`operator ==` and `operator !=`) but only with objects or the identical class. Checks for equivalence of two transformations of the same group can be done by explicitly converting one to the type of the other. Also, in some cases there are multiple possible representations for equivalent rotations (for example, an `AxisAngle` has the same physical meaning if both the axis and the angle are negated) – in such cases, the equivalent rotations may fail to indicate equality.

4.1 Rotations

A (3D) Rotation object is an object which can be applied to any 3-vector of Lorentz vector to yield another vector of the same type representing the result of a homogenous, length-preserving linear transformation (a rotation). Two syntaxes are supported:

```
v2 = R * v1;
v2 = R(v1);
```

Note that “left multiplication” of a vector times a rotation is not supported. Thus to get the conjugate product of a pair of vectors with respect to some rotation one can do:

```
double a = v1 * (R * v2);
double b = v1 * R(v2);
double c = v1 * R * v2;  // error - v1*R is not supported
```

All the rotation objects represent active rotations, that is, they may be applied to a vector to change it into a new vector in the same coordinate system.

There are four 3D rotation classes, representing general rotations in 3 dimensions in four different ways:

Rotation3D	The familiar 3x3 orthogonal matrix representation. This form has nine components, and is the optimal form for applying a rotation to a vector.
AxisAngle	The rotation is represented by a unit-vector axis of rotation, and an angle of rotation about that axis (a total of four components). This form is a good match with physicists' mental model of a rotation, but both application to vectors and multiplication of rotations involve computing trigonometric functions.
EulerAngles	The three-angle expression for a rotation is the minimal amount of data (three components) to describe a general rotation. Operations involving Euler angles tend to be more costly than those involving other representations.
Quaternion	The quaternion representation of a rotation involves four components (which must add in quadrature to unity) which are coefficients of four 2×2 complex matrices labelled (u, i, j, k). (These are familiar as the Pauli spin matrices, plus the unit matrix.) The quaternion representation is optimal for multiplying rotations, and ideal for expressing the “distance” between two rotations, but is slower than the matrix representation for applying a rotation to a vector.

There are 3 additional classes, representing specialized rotations about the X, Y, and Z axes.

RotationX
RotationY
RotationZ

These are very efficient to apply to vectors. Although they have just one formal component (the angle of rotation) they also hold the cosine and sine of the angle of rotation for optimum speed in application. In discussions below, we will call these "axial rotations."

`RotationZ` has the special property that it can be used as a 2-D rotation, that is, you can apply a `RotationZ` to a `DisplacementVector2D` or a `PositionVector2D`.

4.1.1 Construction and Component Access

All rotations are default constructible (giving the identity or trival rotation) and copy constructible.

All rotations are constructible taking a number of Scalar arguments matching the number (and order) of components. In addition, all rotations other than axial rotations are constructible from (begin, end) iterators that behave like iterators (or pointers) to an array of the appropriate number of Scalar components. These two constructors are "raw", in the sense that the components are used unchecked, and the user is responsible for ensuring that they meet whatever conditions are necessary for the given type of rotation representation.

All rotations *except* the specialized axis rotations are constructible from any other type of rotation (including the specialized ones). These single-argument constructors are declared explicit.

All rotations are capable of application (either operator* or operator()), on any `DisplacementVector3D`, `PositionVector3D`, or `LorentzVector3D`. Rotations are also capable of application on an arbitary foriegn vector, as long as that vector has const methods $x(), y()$, and $z()$ to provide its Cartesian coordinates, and a constructor taking three Cartesian coordinates. The result of applying a rotation to a vector is a vector of the same type.

Each rotation class has a `Rectify()` method, to take an instance whose components may have drifted from perfectly satisfying the conditions needed for a true rotation (orthonormality in the case of `Rotation3D`, for example) and correct it to a true rotation in that representation.

Each rotation class has additional constructors and component access methods particular to that representation. In the list below, arguments are always Scalars passed by value, except in the case of the axis for `AxisAngle` which is passed by const reference, and in the case of all `GetComponents()` methods, which use non-const references.

Rotation3D Construction, assignment, `SetComponents` and `GetComponents` using an abritrary foreign linear algebra matrix object. The requirements on the matrix object's class is that it have const and non-const `operator()(int i, int j)` where i and j run from 0 to 2. The `operator()(int i, int j)` methods must return a Scalar or const reference to a Scalar, or for `GetComponents`, a non-const reference to a Scalar. For `SetComponents`, the constructor is "raw": The compnents are accepted as is, without any rectification step.

Construction, assignment, `SetComponents` and `GetComponents` taking three references to vectors. For `SetComponents`, the class used for the

VECTOR OBJECTS MUST HAVE METHODS `x(), y(),` AND `z()` RETURNING VALUES (OR REFERNCES) WHICH CAN BE USED AS SCALARS. FOR `GetComponents`, THE CLASS USED FOR THE VECTOR OBJECTS MUST HAVE A CONSTRUCTOR ACCEPTING THREE SCALARS. For constructing or setting a `Rotation3D`, the vectors should be at least roughly orthonormal; the rotation will be rectified to correct for deviations from orthonormality.

AxisAngle Construction, assignment, `SetComponents` and `GetComponents` taking a vector to be used as the axis followed by a Scalar to be used as the angle of rotation(in radians). The class used for the vector object must have methods `x(), y(),` and $z()$ returning Scalars or const references to Scalars, or for `GetComponents`, non-const references to Scalars. For constructing or setting an `AxisAngle`, the angle is given in radians, and the vector must be non-zero; the rotation will be rectified to hold a unit vector.

Const methods `XYZVector Axis()` and `Scalar Angle()`.

EulerAngle Construction, assignment, `SetComponents` and `GetComponents` taking three Euler angles (ϕ, θ, ψ). The Euler angle conventions used match those of CLHEP Vector.

Const methods `Phi()`, `Theta()` and `Psi()` returning Scalars.

Quaternion Construction, assignment, `SetComponents` and `GetComponents` taking four quaternion components which we label `(U, I, J, K)`. These are the coefficients of the identity matrix and the three Pauli matrices in this quaternion. For constructing or setting a `Quaternion`, the sum of the squares of the components should be (at least roughly) unity; the components will be rectified to leave a unit 4-vector. The relation between the quaternion representation and the Axis/angle representation of the equvalent rotation is that the axis is a unit vector in the direction of $\vec{q} = (i, j, k)$ and the rotation angle is given by $|\vec{q}| = \sin(\theta/2)$ and $u = \cos(\theta/2)$.

Const methods `U()`, `I()`, `J()` and `K()` returning Scalars.

RotationX Construction, assignment, `SetComponents` and `GetComponents` taking the angle of rotation about the X axis.

Const method `Angle()` returning a Scalar in the range $(-\pi, pi)$.

Const methods `SinAngle()` and `CosAngle()` returning Scalars.

RotationY Construction, assignment, `SetComponents` and `GetComponents` taking the angle of rotation about the Y axis.

Const method `Angle()` returning a Scalar in the range $(-\pi, pi)$.

Const methods `SinAngle()` and `CosAngle()` returning Scalars.

RotationZ Construction, assignment, `SetComponents` and `GetComponents` taking the angle of rotation about the Z axis.

Const method `Angle()` returning a Scalar in the range $(-\pi, pi)$.

Const methods `SinAngle()` and `CosAngle()` returning Scalars.

4.1.2 Manipulation of Rotations

Each class representing rotations has member functions `Invert()` for in-place inversion and `Inverse()` to return an object of the same kind which is the inverse of the original.

There is a free function `double Distance( r1, r2)` where `r1` and `r1` are instances of (not necessarily the same) rotation classes. This returns the minimal opening angle between the 3-sphere representations of the two rotations (the (i, j, k) parts of their quaternion representations). The distance is zero for equivalent rotations and can be no greater than $\pi/2$.

Two rotation objects of the identical type can be equality compared. This does a simple test for equal components. (If some sequence of operations have brought the two into some state where they use different component values but represent equivalent rotations, the two rotations may compare as being unequal.)

Any two rotations R and S can be multiplied. The meaning is that if $\vec{v}$ is a vector, $(R * S)(\vec{v}) = R(S(\vec{v}))$. The product of two rotations of identical type is expressed as a third rotation of that same type.

Multiplication of two rotations of different types, `R * S`, returns:

- A rotation of the same type as `R` if R is not an axial rotation.
- A rotation of the same type as `S` if R is an axial rotation but `S` is not.
- The product of two axial rotations will return a `Rotation3D`.

Any rotation other than an axial rotation may be post-multiplied (`operator *=()`) by any other rotation. Axial rotations may be post-multiplied only by a rotation of the identical type.

4.1.3 Euler Angles Convention

A note about conventions for `EulerAngles`:

There are quite a few possible conventions for the meanings (and orderings) of the three Euler angles (ϕ, θ, ψ). The choice made for CLHEP, which the GenVector package will follow, is that of Goldstein in *Classical Mechanics.* Unfortunately, Goldstein uses passive rotations (that is, considers the unrotated vector, expressed in rotated coordinate axes). This is a potential source of confusion since the preference of HEP software–adhered to in the GenVector package–is for active rotations.

We here unambiguously define what we mean by a rotation `E = EulerAngles(phi,theta,psi)`: If we apply `E` to a vector $\vec{v}$ then the resulting vector is given by $A\vec{v}$ where A is the matrix

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

This is the definition given in Goldstein, equation (4-46), but in this package the meaning is that of forming a new vector in the existing coordinate system, rather than that of rotating the coordinate system by some angles and expressing the original vector in the new coordinates.

4.2 LorentzTransformations

A LorentzRotation is an object which can be applied to any `LorentzVector` to yield another `LorentzVector` of the same type representing the result of a homogenous, norm-preserving linear transformation, using a norm with a $(---+)$ metric:

$$s^2 \equiv (\delta t)^2 - (\delta x)^2 - (\delta y)^2 - (\delta z)^2$$

Such transformations are general Lorentz transformations. (Note that we take $c = 1$ in these discussions.)

Every physicist is familiar with Lorentz boosts, which are "rotations" involving time and one space direction. There is time dilation by a factor of γ and length contraction in the direction of motion by that same factor. These are a special case of the general Lorentz transformation, and are represented in this package by the class `Boost`. The package also includes further-specialized axial boosts `BoostX`, `BoostY`, and `BoostZ`.

Another special type of Lorentz transformation is a 3-D rotation. The GenVector package does not provide separate classes to represent a pure space rotation as applied to 4-vectors; the existing 3-D Rotation classes serve that purpose.

The most general form of a Lorentz transformation can be thought of as a combination of a rotation and a boost (in either order, though the precise values of the two transformations will depend on which order you choose).

Two syntaxes for applying a `LorentzRotation` to a `LorentzVector` (OR ANY ARBITRARY FOREIGN VECTOR WITH METHODS `x(), y(), z()` AND `t()` AND A CONSTRUCTOR TAKING `(x,y,z,t)`) are supported:

```
p2 = L * p1;
p2 = L(p1);
```

Note that "left multiplication" of a Lorentz vector times a Lorentz rotation is not supported.

There is one class representing general Lorentz rotations:

LorentzRotation	The 4x4 orthosymplectic matrix representation of a Lorentz transformation. This form has sixteen components, and applying this to 4-vector is the same as (right-) multiplication by that 4x4 matrix.

There is an additional class, representing the special case of pure boosts along arbitrary directions:

Boost	A pure boost, which in principle can be characterised by just 3 components of a vector $\vec{\beta} = \vec{v}/c$, is nonetheless kept as a 4x4 orthosymplectic matrix, because this form is optimal for application to a 4-vector and for combination with other boosts, rotations, and Lorentz rotations.

There are 3 additional classes, representing specialized boosts along the X, Y, and Z axes.

BoostX
BoostY
BoostZ

These are very efficient to apply to 4-vectors. Although they have just one formal component, they hold both β and γ for optimum speed in application. In discussions below, we will call these "axial boosts."

4.2.1 Construction and Component Access

All Lorentz transformations are default constructible (giving the identity or trival transformation) and copy constructible.

All Lorentz transformations are constructible taking a number of Scalar arguments matching the number (and order) of components: The sixteen matrix components for a general transformation, the three $\vec{\beta}$ components for a pure boost, and just β for an axial boost. In addition, all Lorentz rotations other than axial rotations are constructible from (begin, end) iterators that behave like iterators (or pointers) to an array of the appropriate number of Scalar components. In the case of the general `LorentzRotation`, these two constructors are "raw", in the sense that the components are used unchecked, and the user is responsible for ensuring that they meet the orthosymlectic condition: The last (T) row, considered as a 4-vector, must have has $t^2 - |\vec{v}|^2 = 1$, each of other rows must have $t^2 - |\vec{v}|^2 = -1$, and the Minkowski space dot product of any two distinct rows must be zero.

The general `LorentzRotation` is constructible from any Lorentz transformation class, or any rotation class. `Boost` is constructible from any axial boost class. These single-argument constructors are declared explicit.

All Lorentz transfomations are capable of application (either `operator*` or `operator()`), on any `LorentzVector` regardless of which 4-D coordinates are used.

Lorentz transfomations are also capable of application on an arbitary foriegn 4-vector, as long as that 4-vector has const methods $x(), y(), z()$ and $t()$ to provide its Cartesian coordinates, and a constructor taking four Cartesian coordinates in that order. The result of applying a Lorentz transfomation to a vector is a vector of the same type.

Each Lorentz transfomation class has a `Rectify()` method, to take an instance whose components may have drifted from perfectly satisfying the conditions needed for for a true Lorentz transformation and correct it. This method is trivial for an axial boost.

Each Lorentz transformation class has additional constructors and component access methods particular to that type of transformation. In the list below, arguments are always Scalars passed by value, except in the case of `BetaVector()` for `Boost`, which is passed by const reference, and in the case of all `GetComponents()` methods, which use non-const references.

LorentzRotation Construction, assignment, `SetComponents` and `GetComponents` using an abritrary foreign linear algebra matrix object. The requirements on the matrix object's class is that it have const and non-const `operator()(int i, int j)` where i and j run from 0 to 3. The `operator()(int i, int j)` methods must return a Scalar or const reference to a Scalar, or for `GetComponents`, a non-const reference to a Scalar. For `SetComponents`, the constructor is "raw": The components are accepted as is, without any rectification step.

Construction, assignment, `SetComponents` and `GetComponents` taking four references to 4-vectors. FOR `SetComponents`, THE CLASS USED FOR THE 4-VECTOR OBJECTS MUST HAVE METHODS `x(), y(), z()` AND `t()` RETURNING SCALARS OR CONST REFERENCES TO SCALARS. FOR `GetComponents`, THE CLASS USED FOR THE 4-VECTOR OBJECTS MUST HAVE A CONSTRUCTOR TAKING FOUR SCALARS FOR X, Y, Z, AND T IN THAT ORDER. For constructing or setting a `LorentzRotation`, the vectors should be at least roughly orthosymplectic; the transformation will be rectified to correct for deviations from the orthosymplectic conditions.

Construction, assignment, `SetComponents` and `GetComponents` taking a rotation and a `Boost` in either order. The meaning is that `L(R,B)` is `L(R)*L(B)` while `L(B,R)` is `L(B)*L(R)`. `GetComponents(R,B)` and `GetComponents(B,R)` have to do non-trival work, but the answer is unambiguous.

Boost Construction, assignment, `SetComponents` and `GetComponents` taking three Scalars to use as $\beta_x, \beta_y, \beta_z$.

Explicit construction and assignment taking an object of any `DisplacementVector3D` class, or a foreign vector implementing `x(), y()` and `z()`. This is used as the vector $\vec{\beta}$ characterizing the boost.

A const accessor `BetaVector()` returning `\vec{\beta}` as a `DisplacementVector3D< Cartesian3D< Scalar> >`.

`Gamma()`, giving the time-dilation factor for the boost.

Although, for efficiency in applying the boost to 4-vectors, components of a symmetric 4 by 4 matrix representing the boost are held, these are treated as private implementation details. However (for optimum performance when constructing a `LorentzRotation` from a `Boost`) there is a method `GetLorentzRotation (Scalar & r)` which fills the sixteen elements of a Lorentz ROtation representation.

BoostX Construction, assignment, `SetComponents` and `GetComponents` taking one Scalar to use as β_x.

`Gamma()`, giving the time-dilation factor for the boost.

BoostY Construction, assignment, `SetComponents` and `GetComponents` taking one Scalar to use as β_y.

`Gamma()`, giving the time-dilation factor for the boost. item [BoostZ] Construction, assignment, `SetComponents` and `GetComponents` taking one Scalar to use as β_z.

`Gamma()`, giving the time-dilation factor for the boost.

4.2.2 Manipulation of Lorentz Transformations

Each class representing Lorentz transformations has member functions `Invert()` for in-place inversion and `Inverse()` to return an object of the same kind which is the inverse of the original.

There is a free function `double Distance( L1, L2)` where `L1` and `L1` are instances of (not necessarily the same) Lorentz transformation classes. This forms $L_1 L_2^{-1}$, decomposes that into a rotation and a pure boost, and returns the sum in quadrature of the distance between the distance from unity of the rotation and the $|\beta|$ of the pure boost. (This function has the desired properties in a distance: It is independant of the order [or for that matter the order of decomposition into boost and rotation], is zero for identical transformations, is unaffected by Lorentz coordinate transformationsations, and obeys the triangle inequalities.)

Two Lorentz transformation objects of the identical type can be equality compared. This does a simple test for equal components.

Any two Lorentz transformations R and S can be multiplied. The meaning is that if $\vec{v}$ is a vector, $(R * S)(\vec{v}) = R(S(\vec{v}))$. The product of two axial boosts of identical type is expressed as a third axial boost of that same type; in all other cases the result is expressed as a general `LorentzRotation`. In particular, the product of two pure `Boost`s is a general `LorentzRotation`, not another `Boost`.

Any Lorentz transformation can be multiplied on the right or left by any rotation; the result in all cases will be a `LorentzRotation`.

Any Lorentz transformation may be post-multiplied (`operator *=()`) by another Lorentz transformation or rotation if and only if the resutl of multiplying the two (as detailed above) is the same type as that of the first Lorentz transformation. Axial boosts, for example, may be post-multiplied only by axial boosts of the identical type.

5 Frequently Asked Questions

1. In CLHEP, we could construct a `HepRotation` from an axis and an angle. How is that done here?

   This is done by saying what we mean:

   ```
   XYZVector axis; double angle; // the desired components
   Rotation3D r (AxisAngle(axis,angle));
   ```

   Similarly, we can construct a `Rotation3D` from Euler angles.

2. I'd like to equality-compare two rotations of the same type, but I don't want equivalent representations of the same actual rotation to appear to be unequal. How is that done?

   The best way is to use the Distance method:

   ```
   AxisAngle a1, a2;
   if ( Distance(a1,a2) < 1.0e-15 ) { ... }
   ```

   Note that we suggest you check using some small tolerance rather than zero; as is usual for real arithmetic, roundoff can cuase two numbers calculated in mathematically equivalent ways to compare as unequal using exact equality.