# ZOOM **PhysicsVector**
# Space and Lorentz Vector Package
# Summary

Fermilab Physics Class Library Task Force

1.1; March 19, 1998

## Contents

The user documentation for the ZOOM PhysicsVector package consists of:

- This short *Package Summary.*
- A *Formulas and Definitions* document providing precise mathematical definitions of methods.
- Some tutorial examples, bundled in the doc area of the package.
- A *Description and User's Guide.*
- A *Names and Keywords* document listing all the tokens that could affect a user.

There is also web-based documentation of the classes and methods and a list of not-yet-implemented features. These can be found by going to the ZOOM home page and clicking on the PhysicsVectors package.

Users should be able to get along reading just this summary for orientation, and looking at the header files to see the details of methods and signatures, The *Formulas and Definitions* provides exact definitions where needed.

# 1 Purpose, Data Types and General Functionality

This is a set of classes and methods implementing the physicist's familiar 3-vector, 4-vector, rotation and Lorentz transformation concepts.

| | **Classes** | **Concepts** |
|---|---|---|
| `Support` | `EulerAngle`<br>`AxisAngle` | Euler angels, PI, and<br>axis/angle structures |
| `PhysicsVectors` | `SpaceVector`<br>`UnitVector` | 3-D vectors, and<br>vectors of unit length |
| | `LorentzVector` | 4-vectors |
| | `EBvector` | paired electric and magnetic<br>field vectors $(\vec{E}, \vec{B})$ |
| | `Rotation`<br>`RotationX`<br>`RotationY`<br>`RotationZ` | rotations of 3-vectors<br>and simpler rotations<br>about a coordinate axis |
| | `LorentzTransformation`<br>`LorentzBoost`<br>`LorentzBoostX`<br>`LorentzBoostY`<br>`LorentzBoostZ` | Lorentz Transformations,<br>pure Lorentz Boosts,<br>and simpler boosts along<br>the coordinate axes |

*The current version is 0.1.x, and is considered a temporary incomplete version– layers below* `LorentzVectors` *are not included. Without the rotation and Lorentz transformation classes, a user can still rotate and transform 3- and 4-vectors, but cannot treat transformations as independent objects in their own right, to be saved, combined, and decomposed.*

`SpaceVector`s may be expressed, and components obtained or set, in cartesian, polar, or cylindrical coordinates; Methods include dot and cross products, unit vectors and magnitudes, angles between vectors, and rotations and boosts. There are also functions of particular use to HEP, like pseudo-rapidity, projections, and transverse part of a `SpaceVector`, and kinetic methods on 4-vectors such as `invariantMass()` of pairs or containers of particles.

## 1.1 Using the PhysicsVectors Package

The PhysicsVectors package is obtained through the usual ZOOM cvs/SoftRelTools mechanism, and is in CDF and D0 repositories; the library name is `PhysicsVectors`.

To use classes, for example SpaceVector and Rotation, include the header file in the following way:

```
#include "PhysicsVectors/SpaceVector.h"
#include "PhysicsVectors/Rotation.h"
```

For convenience the package provides a single overall header which itself draws in all class definitions:

```
#include "PhysicsVectors/PhysicsVectors.h"
```

## 1.2 Components, Constructors, and Accessors

For each class we define in this package, the methods for construction and for setting or obtaining data obey certain conventions. These use the idea of the "components" of an object.

- The components are the input to the constructor for that object. If there are several forms of components listed here, there is a constructor for each form, taking arguments *in the order listed.*

- For each component listed in the table, there is a method to return the value of that component. That method is given the same name as the component. That same method allows you–where sensible–to change the value of that component *keeping the values of the other components in that form of expressing the object fixed.* For example,

  ```
  Float8 r, theta, phi;
  SpaceVector v ( r, theta, DEGREES, phi, DEGREES );
      //  Components can be expressed as r, theta, phi --
  Float8 alpha = v.theta();
  v.theta() = .5; // Changes phi to .5 radians, keeping r, theta fixed
  ```

- The object may be constructed in one form and accessed in another form. That is, you might specify (x,y,z) and then read off r, theta, or phi.

- Also, there is a method `set()` which sets all components at once. For each form of components there is a signature of `set()` matching the signature of that form of constructor.

## 1.3 The isNear Mechanism

For all objects in these classes, the operators `==` and `!=` have obvious meanings, in terms of exact equality. However, when working with actual vectors, it is useful to have a looser criterion for practical equality. And boolean methods such as `isOrthogonal()` are more certainly useful if they allow for some tolerance. The methods in the PhysicsVectors package support checking for equality *within a given relative tolerance* using these rules:

1. For each object, the exact == comparision is supplemented by an `isNear()` method, which takes an object of the same class and returns a `bool` result (`true` or `false`). For example,

```
SpaceVector v1, v2;
if ( v1.isNear(v2) ) { /* do whatever */ }
```

2. The meaning of isNear() is dependent upon some *relative* tolerance: An optional last argument to all such functions is a (`Scalar`) relative error which we will call `tolerance`. Normally this would be omitted, in which case it defaults to a universal `tolerance` which will be 100 times the machine epsilon.

# 2 SpaceVectors and UnitVectors

Ordinary 3-vectors are supported via the `SpaceVector` class. Directions in 3-space are supported via the `UnitVector` class. Where only a direction is needed, methods are supplied to take advantage of the knowledge that a `UnitVector` has magnitude 1. However, there is a conversion constructor, valid for any non-zero `SpaceVector` to a `UnitVector`, so anywhere a `UnitVector` may be used, a `SpaceVector` may be substitued.

## 2.1 Vector Coordinates

This package recognizes three forms of coordinates: Cartesian (x, y, z), Polar (r, theta, phi), and Cylindrical (rho, phi, z). A `SpaceVector` may be constructed using any of these forms. A given component may be read off in any of these coordinate systems. And a given component may be set while keeping the other two components in that coordinate system fixed.

| | | | |
|---|---|---|---|
| x | y | z | Cartesian coordinates |
| r | theta | phi | (Spherical) polar coordinates |
| rho | phi | z | Cylindrical coordinates |

```
Scalar x, y, z, r, theta, phi, rho, et;
SpaceVector v( x, y, z   );
SpaceVector v1( r, theta, RADIANS, phi, RADIANS );
SpaceVector v2( r, eta, ETA, phi, RADIANS );
SpaceVector v3( rho, phi, RADIANS, z   );
Scalar c1, c2, c3, r, pa, aa;
c1 = v.x();   c2 = v.y();     c3 = v.z();
c1 = v.r();   pa = v.theta(); aa = v.phi(); et = v.eta();
c1 = v.rho(); aa = v.phi();   c3 = v.z();  // only rho() is new
v.x() = c1;   v.y() = c2;     v.z() = c3;
v.r() = r;    v.theta() = pa; v.phi() = aa; v.eta() = et;
v.rho() = rho;
```

Notice the funny-looking keyword `RADIANS` in the example. This is an argument which serves to distinguish one signature of constructor from another. Instead of `RADIANS`, you can substitute `DEGREES` or (when supplying $\theta$) psuedorapidity `ETA`.

`UnitVector` is a class that inherits from `SpaceVector`, but some methods of `SpaceVector` which don't make sense for unit vectors are made private to flag their use as an error at compile time. `UnitVector`s may be constructed from a non-zero `SpaceVector` or by supplying the polar $(\theta, \phi)$.

## 2.2 SpaceVector Operations and Properties

`SpaceVector`s form a vector space: Supported operations on `SpaceVector`s include assignment, unary minus, multiplication (left or right) and division by a Scalar, and addition/subtraction of two `SpaceVector` s.

```
v1 = v2 + c*v3;    v1  = v2 - v3*c;  v1  = v2 - v3/c;
v1 *= c;           v1 += v2;         v1 -= v2;
```

`UnitVectors` do not form a vector space; they may be multiplied by a Scalar or added tegether, but the result is an ordinary `SpaceVector`.

There are methods representing the usual binary 3-vector operations: Dot and cross products. There are also logical tests for whether vectors are parallel or orthogonal.

```
Scalar v1.dot(v2)
SpaceVector v1.cross(v2)
Scalar v1.diff2(v2) // squared difference
bool v1.isNear(v2)
bool v1.parallel(v2)
bool v1.orthogonal(v2)
```

The boolean methods test for near-alignment with a relative tolerance rather than perfect alignment.

The `SpaceVector` class provides a variety of methods returning numerical properties of a single vector:

```
Scalar v.mag(),         Scalar v.mag2()
Scalar v.beta(),        Scalar v.gamma()
```

There are also methods returning properties of a vector, or performing operations on two vectors, with respect to some reference direction. Typically the physicist wants these properties relative to the Z direction. We call these "z-implied" methods: The reference direction is usually omitted, and the Z axis is assumed. This example shows a signature using a reference direction only in the most commonly used case: The angle between a vecotr and a direction.

```
Scalar v.perp()              Scalar v.perp2()
Scalar v.angle()             v.angle(dir)
Scalar v.cosTheta()          Scalar v.cos2Theta()
Scalar v.eta()               Scalar v.rapidity()
SpaceVector v.project()      SpaceVector v.perpPart()
Scalar v.azimAngle(v2)       Scalar v.polarAngle(v2)
```

See the *Formulas and Definitions* document for precise meanings of these methods, particularly rapidity and angular decomposition.

## 2.3 Transformations by Rotating

The `SpaceVector` class, which has no knowledge of the `Rotation` class, contains methods to rotate this vector by some amount specified in the method arguments. The `SpaceVector` and `UnitVector` classes have methods which rotate the vector itself (e.g. `v.rotate(axis, delta)`). They also provide global functions which return a new vector leaving the original unchanged (e.g. `v2 = rotationOf(v1, axis, delta)`).

The rotation may be specified in terms of an axis and angle of rotation, or by specifying a set of Euler Angles. Rotation about one of the coordinate axes is a special case; methods are provided to take advantage of the simpler form of operation.

For convenience in reading off values of `Rotation` objects, the package provides two classes `EulerAngles` and `AxisAngle` which may be used as structs to receive the returned result. For consistency, the rotate methods also accept these structs as input.

```
SpaceVector& v.rotate(Scalar phi, Scalar theta, Scalar psi);
SpaceVector& v.rotate(UnitVector dir, Scalar delta);
SpaceVector& v.rotate(EulerAngles ex);
SpaceVector& v.rotate(AxisAngle ax);
SpaceVector& v.rotate(Scalar phi, Scalar theta, Scalar psi);
SpaceVector& v.rotateX(Scalar delta);
SpaceVector& v.rotateY(Scalar delta);
SpaceVector& v.rotateZ(Scalar delta);

SpaceVector rotationOf (v, Scalar phi, Scalar theta, Scalar psi);
SpaceVector rotationOf (v, UnitVector dir, Scalar delta);
SpaceVector rotationOf (v, EulerAngles ex);
SpaceVector rotationOf (v, AxisAngle ax);
SpaceVector rotationXOf(v, delta);
SpaceVector rotationYOf(v, delta);
SpaceVector rotationZOf(v, delta);
```

# 3 Lorentz Vectors

4-vectors in Minkowski space are supported via the `LorentzVector` class.

When constructing or setting a `LorentzVector` from four cartesian components, one could be confused as to whether to supply (x,y,z,t) or (t,x,y,z). To eliminate possible accidents, the signature for such methods takes three Scalars x, y, and z, and an instance of type `Tcomponent` for t.

```
LorentzVector w ( 1, 3, 5, Tcomponent(40) ); // Preferred
LorentzVector w ( Tcomponent(40), 1, 3, 5 ); // Also accepted
LorentzVector w ( 40, 1, 3, 5 );             // Nope.  No such signature.
```

Constructors are also provided taking a `SpaceVector` and a time component, or just a time component for a particle at rest. Access is supported for these forms of components.

```
LorentzVector w(SpaceVector v, Tcomponent t);
LorentzVector w(SpaceVector v, Scalar t);     // OK
LorentzVector w(Tcomponent t); // (0,0,0,t)
LorentzVector w(Scalar t);     // (0,0,0,t)   // OK
Scalar x = w.x();  Scalar x = w.y();
Scalar z = w.z();  Scalar t = w.t();
SpaceVector v = w.v(); v  = w.getV();
w.set(Tcomponent t, Scalar x, Scalar y, Scalar z);
w.set(Scalar x, Scalar y, Scalar z, Tcomponent t);
w.x() = x; w.y() = y; w.z() = z; w.t() = t;
```

As shown, in cases where no ambiguity would result, a Scalar value of $t$ may be used in place of the Tcomponent.

## 3.1 LorentzVector Operations and Properties

Supported operations on `LorentzVector`s include assignment, unary minus, multiplication (left or right) and division by a Scalar, and addition/subtraction of two `LorentzVector`s.

```
w1 = w2 + c*w3;    w1  = w2 - w3*c;  w1  = w2 - w3/c;
w1 *= c;           w1 += w2;         w1 -= w2;       w2 = -w1;
```

The dot product is supported; by default, a $(- - - +)$ metric applies. That is, if `w` is timelike, w.dot(w) will be positive.

```
LorentzVector w, w1;
Scalar x = w.dot(w1);                /* Metric-dependant */
Scalar x = w.diff2(w1);              /* Metric-dependant */
Scalar x = w.delta2Euclidean(w1);
bool b = w.isNear(w1);             bool b = w.isNearCM(w1);
bool b = w.isParallel(w1);
```

Notice that isNearCM(), a Lorentz-invariant test for near-equality in the center-of-mass frame–applicable only to timelike `LorentzVectors`–is provided along with the Euclidean-norm method isNear(). The mathematical meanings of near-equivalence and isParallel() are listed in the *Formulas and Definitions* document.

There are three special boolean methods `isTimelike()`, `isLightlike()`, and `isSpacelike()`, which test whether the squared norm `w.dot(w)` is positive, zero, or negative. The `isLightlike()` method implies a check for equality, and therefore uses a tolerance as described in §1.3.

Several other simple property methods are provided as well:

```
LorentzVector w;
bool b = w.isSpacelike();  b = w.isLightlike(); b = w.isTimelike();
Scalar x = w.mag2();   /* w.dot(w) -- Metric-dependant */
Scalar  plus();   Scalar  minus();  /* t + z  and t - z */
Scalar  euclideanNorm2();   Scalar  euclideanNorm()
```

## 3.2 4-Vector Kinematics

Methods implementing useful kinematic concepts are provided.

*These methods all return results independent of the choice of metric.*

There are kinematic functions relating to the boost it would take to bring a particle at rest into having some specified `LorentzVector` as its 4-momentum. These return the relativistic beta, gamma, and boostVector, the `SpaceVector` which when used as a boost would make this transformation.

There are also methods to extract the rest mass, and the invariant mass of a pair or a list of `LorentzVectors`.

Pseudorapidity and rapidity are computed with respect to some reference direction, which defaults to the Z direction; see the *Formulas and Definitions* document for precise meanings.

```
Scalar restMass2()                  Scalar restMass()
LorentzVector rest4Vector()  /* ( 0, 0, 0, restMass() ) */
SpaceVector boostVector()
Scalar eta()                        Scalar rapidity()
Scalar w1.invariantMass(w2)         Scalar w1.invariantMass2(w2)
SpaceVector w1.findBoostToCM(w2) // boost to joint CM frame
```

These important concepts are also provided for standard containers of `LorentzVectors`– see the *Description and User's Guide* for how to do that.

## 3.3 Rotations and Boosts of Lorentz Vectors

The `LorentzVector` class contains methods to transform the vector by some amount specified in the method arguments: A `LorentzVector` may be rotated by in the same manner as a SpaceVector, leaving the time component intact. And a LorentzVector may be boosted by some fraction $\beta$ of the speed of light along one of the three space axes or in a given direction, or by some SpaceVector with magnitude less than 1, which is treated as a velocity (in units where $c = 1$).

The class has methods that transform the 4-vector itself (e.g. `w.boost(axis, beta)` and `w.rotate(axis, delta)`). It also provides global functions which return a new LorentzVector leaving the original unchanged (e.g. `w2 = boostOf(w1, axis, beta)` and `w2 = rotationOf(w1, axis, delta)` ).

Rotation about one of the coordinate axes, or boosts along a coordinate axis, are special cases; methods are provided to take advantage of the simpler form of these operations.

```
LorentzVector w, w2;
UnitVector dir;
Scalar beta;
SpaceVector b;
w2 = boostOf(w, dir, beta);
w2 = boostOf(w, b);
w2 = boostXOf(w, beta); w2 = boostYOf(w, beta); w3 = boostZOf(w, beta);
w.boost(dir, beta);     w.boostZ(beta);
Scalar phi;
w2 = rotationOf(w, dir, phi);
w.rotateX(phi);  w2 = rotationYOf(w, phi);  w.rotateZ(phi);
```

# 4 Electromagnetic Fields

The EBvector class (*not yet available*) represents a pair of vectors $(\vec{E}, \vec{B})$ that transform under rotations and Lorentz Transformations as members of an electromgnetic field tensor $F_{\mu\nu}$.

The operations on an EBvector include:

- Algebraic linear operations.
- Intrinsic properties of the vector representing physical properties of the electromagnetic field–for instance, `eb.energyDensity()` and `SpaceVector eb.poynting()`
- Transformations by rotating.
- Transformations by Lorentz boosts.

# 5 Rotations

This package deals with Rotations (and LorentzTransformations) as objects, representing members of a group. Thus you can construct a Rotation with some value, multiply two Rotations to produce a third, and invert a Rotation.

A Rotation is conceptually a re-orientation all of space with respect to the axes of some fixed coordinate system, which when applied to a SpaceVector will modify the coordinates of the vector. That is, we are talking about *active* rotations (and later boosts), rather than transformations of the reference frame.

A Rotation can be represented by a real, orthogonal matrix with determinant +1. Combining two Rotations is represented by multiplying their matrices.

## 5.1 Constructing and Accessing Rotations

A `Rotation` can be constructed as an angle of rotation about an axis direction (as always, a `SpaceVector` may be substituted for the `UnitVector` specifying the direction), or in terms of Euler Angles. Also, you may construct a `Rotation` by supplying three orthogonal `UnitVectors` to use as its rows.

The value of a `Rotation` may be obtained in any of those forms, and also in the form of individual components of the matrix representing the `Rotation`.

| | | | |
|---|---|---|---|
| phi | theta | psi | Individual Euler angles |
| eulerAngles | | | An `EulerAngles` struct |
| axis | delta | | Axis and angle of rotation |
| axisAngle | | | An `AxisAngle` struct |
| rowX | rowY | rowZ | orthogonal unit vectors |

As is the case for vectors, methods are supplied to set one "component" of a `Rotation` keeping the remaing ones fixed. But here by "component" we mean one argument to a possible constructor of `Rotation`, keeping the other arguments in that constructor fixed.

The default constructor sets the `Rotation` to the identity rotation. The package also provides a static constant `Rotation::IDENTITY`.

```
Rotation r0 ();   ASSERT (r0 == Rotation::IDENTITY);
Rotation r1 ( SpaceVector(1,2,3), .02 ); // axis and delta
Rotation r2 ( PI/4, PI/3, PI/6 ); // Euler angles phi, theta, psi

UnitVector v  = r1.axis();  // unit vector
Scalar angle  = r1.delta(); // Angle of rotation about an axis
Scalar psi    = r1.psi();   // Euler psi (==angle psi around axis)
Scalar theta  = r1.theta(); // Will always be <= 90 degrees.
Scalar  phi   = r1.phi();

r.setDelta (Scalar delta);  // change delta, keep axis fixed
r.setPsi (Scalar psi);      // change psi, keep phi, theta fixed
```

In addition to the forms of components, you can obtain, but cannot individually change, the values of a row or a single element of the matrix representation of a `Rotation`.

```
Rotation r1;
Scalar elem = r1.xx();
UnitVector u1 = r1.rowX();
```

## 5.2 Operations in the Rotation Group

The usual assignment operator and comparisons for equality apply to `Rotations`. Moreover, a multiplication operator is defined, and `Rotations` form a group under this operation. Multiplication of `Rotations` is written using the usual `*` (or `*=`) operator.

Inversion of a `Rotation`, represented by inverting the orthogonal matrix representation, is provided by the inverse() method. Note that `r1.inverse()` produces a result of the inverse of r1; it does not change r1 itself. To invert a `Rotation` in place, use `r1.invert()`.

Equality (and dictionary ordering) comparison is supported. The isNear mechanism is also supported among rotations; see the *Formulas and Definitions* document for a precise definition.

```
Rotation r1, r2, r3;
r3 = r2 * r1;
r2 = r1.inverse();
r3 = r2 * r1;
ASSERT ( r3.isNear(Rotation::IDENTITY));
```

### 5.3 Applying Rotations

A `Rotation` may be used as a "functor," applying to a `SpaceVector` and yielding a new `SpaceVector`. The notation reflects this point of view:

```
Rotation r1;
SpaceVector v1, v2;
v2   = r1(v1);
```

The application of a `Rotation` to a `SpaceVector` may be viewed as left multiplying the vector representation by the matrix representing that rotation.

Similarly, a `Rotation` applied to a `UnitVector` produces a `UnitVector`, and a `Rotation` applied to a `LorentzVector` produces a `LorentzVector`. And a `Rotation` may be applied an `EBvector`, rotating each component to yield a new `EBvector`.

The `*` multiplication notation is *not* used. Multiplication of a rotation by by a vector on its left is not permitted.

A `Rotation` may be applied in bulk to each element of a container of `SpaceVector`s, `UnitVector`s, `LorentzVector`s, or `EBvector`s. A new container is returned, each of whose elements is the corresponding vector in the original, rotated by the specified `Rotation`.

The container may be any of the standard container types.

```
Rotation r1;
list<SpaceVector> v_list;
list<SpaceVector> vprime_list = r1(v_list);
```

### 5.4 Special-case Rotations

The package provides three additional classes, representing rotations around the X, Y, and Z axes. `RotationX`, `RotationY`, `RotationZ` have internal representations which are more compact that that for a general `Rotation`, and the forms of their constructors are simpler.

Aside from the simpler constructors (and correspondingly simpler `set()` methods), all methods and signatures available for `Rotation`are available for `RotationX`, `RotationY`, `RotationZ`. Of course, the implementations will take advantage of the simpler forms, to do much less work.

```
RotationX rx1 (.25); // delta is specified in radians, the axis is X
SpaceVector v1, v2;
v2 = rx1(v1);
```

These specialized rotation objects are defined in `Rotation.h`. Multiplying two specialized rotations of different types will give a general `Rotation`.

## 6 LorentzTransformations

Lorentz transformations are general combinations of pure rotations and pure Lorentz Boosts. `LorentzTransformation`s form a group. Generally, these `LorentzTransformation`s are useful because they may be *applied* to a `LorentzVector`.

A `LorentzTransformation` may be represented by a $4 \times 4$ matrix. Combining two `LorentzTransformations` is represented by multiplying their matrices. They form a group under this multiplication operation. The matrix has the property that for any `LorentzVector`, the transformed `LorentzVector` has the same mag2() as the original.

A few words of orientation: We always work with $c = 1$, and never consider "tachyonic" transformations. And in defining Lorentz Transformations we have use the same "active" viewpoint as was used in defining rotations: Space (or in this case, spacetime) is "rotated" by some amount, rather than saying that the new coordinate system is "rotated" by some amount relative to the old. A pure boost by $\beta$ in the X direction can be represented by a matrix

$$\begin{pmatrix} \gamma & 0 & 0 & \beta\gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta\gamma & 0 & 0 & \gamma \end{pmatrix}$$

## 6.1 LorentzBoosts and LorentzTransformations

A general `LorentzTransformation` has six degrees of freedom; any `LorentzTransformation` can be expressed as the product of a rotation and a pure boost (though this decomposition is not trivial to find.)

However, many (probably most) HEP applications use pure boosts, and not mixed combinations of rotations and boosts. Since the operations with a `LorentzTransformation` known to be a pure boost can be accomplished with less work than for the general case, this package supports a class `LorentzBoost` representing a pure boost in some direction.

`LorentzBoost` is on the same footing as `Rotation`: Each represents three of the six degrees of freedom of a general `LorentzTransformation`. Other than constructors and component access, the same set of methods apply to `LorentzBoost` as to the more general `LorentzTransformation`. In particular, `LorentzBoosts`, `Rotations` and `LorentzTransformations` may be multiplied by one another, with the obvious form of results: Boost times boost remains a pure boost, rotation times rotation remains a pure rotation, and all other combinations result in a general `LorentzTransformation`.

The package also provides additional classes, representing pure boosts along the X, Y, and Z axes. `LorentzBoostX`, `LorentzBoostY`, `LorentzBoostZ` have internal representations which are more compact that that for a general `LorentzBoost`, and the forms of their constructors are simpler.

Again, aside from the simpler constructors and access methods, all methods and signatures available for `LorentzBoost` and `LorentzTransformation` are available for these classes, but implementations will take advantage of the simpler forms.

All these boost objects are defined in `LorentzTransformation.h`.

## 6.2 Constructing and Accessing Lorentz Transformations

A `LorentzBoost` can be constructed by supplying a `UnitVector` direction and $\beta$ between -1 and 1, or by supplying a `SpaceVector` of less than unit length to use as $\vec{\beta}$.

The default constructor sets the `LorentzBoost` to the identity transformation, which is equivalent to using a boost vector of zero.

A `LorentzTransformation` can be constructed as a pure rotation or a pure boost, or by combining the two (in either order). It can also be constructed by supplying four `LorentzVectors` obeying appropriate conditions.

```
UnitVector dir; Rotation r1; SpaceVector bv;
LorentzBoost t1 ( dir, -.04);
LorentzBoost t2 ( bv );              // bv.mag2() must be < 1
```

```
LorentzTransformation t3 ( r1 );
LorentzTransformation t4 ( bv );
LorentzTransformation t5 ( bv, r1 );   // t5 = t4*t3
LorentzTransformation t6 ( bv, r1 );   // t6 = t3*t4
LorentzVector lam1, lam2, lam3, lam4;
LorentzTransformation t7 (lam1, lam2, lam3, lam4);
```

There are methods to get and/or modify components of a `LorentzBoost`. In addition, ALL the access methods available for a general `LorentzTransformation` are available for `LorentzBoost` (and for `Rotation`as well), although most of these are trivial in one respect or another.

Every `LorentzTransformation` can be expressed as a pure boost along some direction, followed by a pure rotation (or by a rotation followed by a boost) and methods are provided to decompose the transformation in this way.

```
Rotation r1;
LorentzTransformation t1;
LorentzBoost b1;
t1.decompose ( &r1, &b1);     // T = R*B
t1.decompose ( &b1, &r1);     // T = B*R
```

Finally, one can get (but not change) the rows or individual components of the representation matrix for a `LorentzTransformation`:

```
LorentzTransformation t1;
Scalar u;
LorentzVector v;
v = t1.row1();  v = t1.row2();  v = t1.row3();  v = t1.row4();
u = t1.xx();    u = t1.yz();    u = t1.tz();    u = t1.tt();
```

## 6.3 Combining and Manipulating LorentzTransformations

The usual assignment operator and comparisons for equality apply to `LorentzTransformation`s.

The group operation among `LorentzTransformation`s is written as multiplication, using the usual `*` (or `*=`) operator. This multiplication is associative but not commutative. It may be pictured as matrix multiplication of the matrices representing the two transformations.

Equality (and dictionary ordering) comparison is supported. The isNear() mechanism is also supported among `LorentzTransformations`; see the ***Formulas and Definitions*** document for a precise definition of when two Lorentz transformations are considered to be near one another.

Inversion of a `LorentzTransformation`, represented by inverting the matrix representation, is provided by the `inverse()` method.

## 6.4 Applying LorentzTransformations

A `LorentzTransformation`, `LorentzBoost`, or `Rotation` may be used as a "functor", applying to a `LorentzVector`. and yielding a new object of that type. The notation reflects this point of view:

```
LorentzTransformation t1;
LorentzVector w1, w2;
w2 = t1(w1);
```

The application of a `LorentzTransformation` to a `LorentzVector` may be viewed as left multiplying the vector representation by the matrix representing that transformation.

An `LorentzTransformation` may also be applied to an `EBvector`. This is mathematically defined as applying the transformation to the field tensor $F_{\mu\nu}$ and reading off the new $\vec{E}$ and $\vec{B}$ from that transformed tensor. This of course mixes $\vec{E}$ and $\vec{B}$.

A `LorentzTransformation` may be applied in bulk to each element of a container of `LorentzVector`s or `EBvector`s. A new container is returned, each of whose elements is the corresponding vector in the original, transformed by the specified `LorentzTransformation`.