FPCL **PhysicsVector**
Space and Lorentz Vector and Transformation Package
Rotations and Transformations
Technical Justifications

Fermilab Physics Class Library Task Force

Version 1.1, March 6, 1998

Contents

1 Rotations

We wish to answer two questions for the three classes `Rotation`, `AxisAngle`, and `EulerAngles`:

1. What criterion should be used to evaluate `isNear()` and `howNear()`?
2. What algorithm should be implemented for comparisons?

Each of these issues have a couple of desired behaviors.

1.1 Behavior of isNear()

`isNear()` and `howNear()` are intended to measure closeness in some sense. While we will desire certain invariance properties, we should *not* care about differences between two definitions if these differences are of order ϵ^2 for objects which are close within ϵ. (This statement is a bit circular on the face of it, but obviously could be said in a more rigorous manner.)

The invariance properties we care about in general are:

1. $a \approx b$ is exactly equivalent to $b \approx a$.
2. The definition obeys (at least to order ϵ) invariance under the known symmetries of the objects, e.g., rotational invariance.
3. If various special cases have obvious meanings of distance and near-ness, the definition for the general case reduces to those obvious meanings in the special cases.
4. `isNear()` and `howNear()` are related by $a \approx b$ if and only if `a.howNear(b)` $\leq \epsilon$.

There may be trade-offs between these desirata and computational efficiency.

For rotations, considered as an abstract concept, a physically meaningful definition of being near is that rotations $R_1 \approx R_2$ means that

$$\sup_{\hat{u}} \{|R_1(\hat{u}) - R_2(\hat{u})|\} \leq \epsilon \tag{1}$$

That is, using as a probe the unit vector most sensitive to the difference between the two rotations, the effect on that unit vector differs by at most ϵ.

This has exact rotational invariance. But it is computationally difficult to apply: It is equivalent to saying that the absolute value of the largest eigenvalue of $R_1 - R_2$ is less than ϵ. The task of finding the eigenvalues of $R_1 - R_2$ is non-trivial.

The special-cases, where we should demand that the general algorithm match some obvious definition, are the coordinate axial rotations. There, the obvious statement is that if R_1 and R_2 are both rotations by δ_1 and δ_2 about the same coordinate axis, then $R_1 \approx R_2$ is equivalent to $\delta_1 \approx \delta_2$ modulo 2π.

1.2 Behavior of Comparisons

The rationale for supporting comparisons other that for equality is to support the various standard library methods that may require $> \geq < \leq$ to be present. For vectors, we use a "dictionary ordering", ordering acording to x, then y, then z.

To be useful in this sense, the properties required are:

1. Exclusivity: for any a and b, Exactly one of $a > b$, $a < b$, or $a = b$ holds true.
2. Semantics: $a \leq b$ if and only if NOT$(a > b)$, and similarly for $a \geq b$ and $a \neq b$.
3. Transitivity: If $a < b$ and $b < c$ then $a < c$.

Since a reasonably intuitive definition has some potential benefits, we also care to some extent that:

1. Our equality relation is tight, requiring exact matches in the data representation. (We have separate `isNear()` methods if the user wants to check for near equality.)
2. There is at least some 1-dimensional manifold in which the ordering looks relatively continuous. For example, for vectors, ordering according to $|x|$ and taking $+x$ before $-x$ for matching absolute values, would be worse in this way than ordering according to x itself.

3. If various special cases have an intuitively natural ordering, then the algorithm used for the general case ought to reduce to those intuitive orderings in those special cases.

4. It's not clear whether efficiency is important for ordering, but it is clear that when simplicity of definition leads to efficiency, that simplicity is a plus. But if a computationally simpler definition is less intuitively reasonable, or violates some other of the above benfits, then the clearer definition may be better.

The special-cases, where we should at least desire that the general algorithm matches an obvious definition, are again for coordinate axial rotations. There, the obvious ordering would be that for two rotations around the same axis, $R_1 < R_2$ if $\delta_1 < \delta_2$.

For `AxisAngle`s, an excellent set of ordering criteria is to compare the two axes, using the `UnitVector` comparison methods, and if they are equal, then compare δ.

1.3 Definitions of isNear() and howNear()

Say we know the components of two rotations P and Q. Then to order ϵ, the condition that

$$\sup_{\hat{u}} \{|P(\hat{u}) - Q(\hat{u})|\} \leq \epsilon \tag{2}$$

is equivalent to saying that if you decompose PQ^{-1} as an angle rotation by δ about an axis, then $\delta \leq \epsilon$. In mathematicians terms, equation (2) states that the 2-norm $\left\|PQ^{-1}\right\|_2 \leq \epsilon$. (Notice that if P and Q are grossly different, this definition has room for jitter, since you could as well have used $Q^{-1}P$ and they don't commute. But as long as we are discussing fairly close rotations, the "don't care about differences of $\mathrm{O}(\epsilon^2)$" lets us use this definition.)

Thus in this sense, the definition we will develop is equivalent (up to higher orders in ϵ) to this excellent definition; and it may be much easier to compute.

If we look at the formula for a rotation about an axis,

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

$$\begin{pmatrix} \cos\delta + (1-\cos\delta)u_x^2 & (1-\cos\delta)u_x u_y - \sin\delta u_z & (1-\cos\delta)u_x u_z + \sin\delta u_y \\ (1-\cos\delta)u_y u_x + \sin\delta u_z & \cos\delta + (1-\cos\delta)u_y^2 & (1-\cos\delta)u_y u_z - \sin\delta u_x \\ (1-\cos\delta)u_z u_x - \sin\delta u_y & (1-\cos\delta)u_z u_y + \sin\delta u_x & \cos\delta + (1-\cos\delta)u_z^2 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{3}$$

we see that we could say that $3\text{–}\mathrm{Tr}(PQ^{-1}) = 2 - 2\cos\delta \approx (\delta^2)$ should be compared to ϵ^2. This becomes:

$$P \approx Q \Longleftrightarrow 3 - \sum_{ij} P_{ij} Q_{ij} \leq \epsilon^2 \tag{4}$$

This is fine mathematically, but involves needing $\mathrm{O}(\epsilon^2)$ accuracy in additions of quantities of order 1, so it is flawed computationally.

We end up using that definition, but here is the worry: The trace involves adding the dot product of each row of P with the corresponding row of Q. Now if each element has a relative error due to machine precision of up to ε, then that dot product has an absolute error of up to $\varepsilon(\sup(|p|) + \sup(|q|))$, that is, ε times the largest number in each row. This means the total uncertainty in the trace is up to 6ε. So comparing $3\text{–}\mathrm{Tr}(PQ^{-1})$ to ϵ^2 introduces a computational error of order $\sqrt{6\varepsilon}$, making it meaningless for comparisons to accuracy better than 8 decimal places (using double precision math).

It turns out you can't do any better than this definition. One consequnce is that we select as the default tolerance for comparing rotations one part in a million, and point out that comparisons to accuracy closer than $4 \cdot 10^{-8}$ are computationally meaningless.

1.3.1 A Rejected Alternative

An alternate possiblity is to look at the off-diagonal terms of PQ^{-1}. The xy term, for example, is $P_x \cdot Q_y$ where P_x is the x row of P, and we have used the orthogonality property of Q to write the y column of Q^{-1} as Q_y. Now we see that

$$\begin{aligned} &(P_x \cdot Q_y)^2 + (P_x \cdot Q_z)^2 + (P_y \cdot Q_z)^2 \\ &+(P_x \cdot Q_y)^2 + (P_x \cdot Q_z)^2 + (P_y \cdot Q_z)^2 \\ = \; &2\sin^2\delta + 2(1-\cos\delta)^2\left[(u_x u_y)^2 + (u_x u_z)^2 + (u_y u_z)^2\right] \end{aligned} \tag{5}$$

where we have of course used the fact that $u_x^2 + u_y^2 + u_z^2 = 1$.

So for delta small, we would just compare that sum of squares of dot products to ϵ^2. This would not quite be enough, since by that criterion, a 180 degree rotation about a coordinate axis would be equal to the identity rotation. That is, we have assumed δ to be small, and been stung by the region $\delta \approx \pi$ when the higher order term in δ

$$2(1-\cos\delta)^2\left[(u_x u_y)^2 + (u_x u_z)^2 + (u_y u_z)^2\right]$$

cannot be neglected.

This can be rectified if we notice that if P and Q are close in any way, $P_x \cdot Q_x$ (and similarly for y and z) will be very close to 1. To be more precise, if P_x and Q_x differ by some small $\vec{\rho}$

$$\begin{aligned} |P_x| &= 1 \\ |Q_x| &= 1 \\ P_x &= Q_x + \vec{\rho} \\ P_x \cdot Q_x = P_x \cdot P_x + P_x \cdot \vec{\rho} &= 1 + P_x \cdot \vec{\rho} \end{aligned} \tag{6}$$

and since $\vec{\rho}$ has to be nearly perpendicular to P_x (to keep the length of Q_x unity) $P_x \cdot Q_x$ is 1 - O(ϵ^2).

Then we are free to add the squares of this sort of small terms, without changing the small-epsilon behavior. However, when $\delta \approx \pi$, these terms become close to 2, and adding the squares rectifies the criterion for the case of rotations related by nearly 180 degree differences.

So our criterion would be that the sum of the squares of the off-diagonal row-products, plus some positive constant times the squares of one minus the same-row-products, would be less than ϵ. Choosing that constant to be 1, we get a fairly natural condition: The Frobenius norm (root mean squared of the individual elements) of $I - PQ^{-1}$ is compared to ϵ. This criterion is not precisely rotation-invariant, but it is rotation-invariant to order ϵ. It would be superior to the simpler trace definition if it were computationally unflawed.

Unfortunately, those dot products, each of which are near zero for nearly equal rotations, are calculated as the sum of three products, each of which can be of order 1. That again introduces O(ε) errors into a comparison to ϵ. That is, the computational accuracy of the quantities which determine this criterion is no better than for the simpler one. Therefore, we reject this approach and stick with equation 4.

We separately provide `FrobeniusDiff()` using the Frobenius norm definition.

1.3.2 isNear() for AxisAngle and EulerAngles

Now consider `AxisAngle`. An example of a really poor criterion would be the following, for deciding whether two `AxisAngle`s are near:

$$|\hat{u}_1 - \hat{u}_2| \leq \epsilon \text{ and } |\delta_1 - \delta_2| \leq \epsilon \tag{7}$$

Aside from the obvious "box-like" nature of a 2-criteria, the more serious flaw can be seen by taking two `AxisAngle` structures, with smallish, nearly equal angles and directions which are off by a smallish amount, but by much more than ϵ. For instance, say we take $\epsilon = 10^{-6}$, and $|\hat{u}_1 - \hat{u}_2| \approx \delta_1 \approx \delta_2 \approx 10^{-4}$, and $|\delta_1 - \delta_2| \approx 10{-}7$. Then although the difference in direction blows away the possibility of nearness (by this flawed definition), the rotations represented have the same effect (to order ϵ) on every possible $\hat{u}$.

We will avoid this trap, in fact, by expressing a pair of `AxisAngle`s or `EulerAngles` as rotation matrices and using the same criterion (4) for determining nearness. We will, of course, have constructors of a rotation from these data types, so this is no coding effort. The does not preclude switching to a quicker definition, if we later find one that is equivalent to order ϵ but takes advantage of the knowledge of axis or Euler angle information.

1.4 Definitions of Comparisons

In defining how to compare two `Rotation`s, the main headache is to match on to the natural ordering amongst coordinate-axis rotations. Frankly, I see only two sane approaches:

1. If we will worry about speed, we can just punt the special-case naturalness requirement, and use straight dictionary ordering: Order by xx components, or if they are equal, by xy, or …up to zz.
2.

If we do it the quick way, then we will have situations like the following:

```
Rotation r1;
RotationY ry2;
Rotation r2 = ry2;
if ( (r1 > r2) && (r1 < ry2) ) { /* Yup, this can happen */ }
```

As mentioned earlier, an excellent criterion for ordering `AxisAngle`s is to compare the two axes, using the `UnitVector` comparison methods, and if they are equal, then compare δ. This is easy to understand, and trivially behaves properly for rotations about coordinate axes.

We can use this definition for `Rotation`s and `EulerAngles` as well; the only cost is that extracting the `AxisAngle` representation is not very quick. Since sorting containers of rotations is not likely to be a common operation, we will go with this criterion. We should warn that if sorting (or using other comparison-based algorithms on) large containers of `Rotation`s, it will be best to convert them to `AxisAngle`s at the start.

Thus, for two rotations `r1, r2`, in pseudo-C++

```
operator > (Rotation r1, Rotation r2) {
  AxisAngle a1(r1), a2(r2);
  if ( a1.getAxis() > a2.getAxis() ) {
    return true;
  } else if {
  if ( a1.getAxis() < a2.getAxis() ) {
    return false;
  } else {
    return (a1.delta() > a2.delta() );
  }
}
```

Of course, we probably would write one compare function rather that three operators.

For `EulerAngles`, we again convert to `AxisAngle`, this time going to `Rotation` along the way.

1.5 Extracting AxisAngle from Rotation

Determining the axis and angle of a `Rotation` is on the face of it an eigenvector problem. Fortunately, it is an easy one, because you know that the eigenvalue associated with the axis eigenvector is 1. In principle, you can just solve the equation $(R-I)u=0$. An eigenvector (of non-unit length) can be computed can be read off immediately as the cross product of two non-zero rows of $(R-I)$. This can then be normalized, unless it is zero, in which case a different choice of rows is made. (If all such cross products are zero, then the rotation is the identity.)

An alternative method, which is in the end simpler, more stable, and requires fewer decisions, is to use equation (3) directly. Repeating the matrix for convenience:

$$\begin{pmatrix} \cos\delta+(1-\cos\delta)u_x^2 & (1-\cos\delta)u_xu_y-\sin\delta u_z & (1-\cos\delta)u_xu_z+\sin\delta u_y \\ (1-\cos\delta)u_yu_x+\sin\delta u_z & \cos\delta+(1-\cos\delta)u_y^2 & (1-\cos\delta)u_yu_z-\sin\delta u_x \\ (1-\cos\delta)u_zu_x-\sin\delta u_y & (1-\cos\delta)u_zu_y+\sin\delta u_x & \cos\delta+(1-\cos\delta)u_z^2 \end{pmatrix} \tag{8}$$

We can quickly find

$$\begin{aligned} \hat{u}_z^2 &= \frac{R_{zz}-\cos\delta}{1-\cos\delta} \\ \hat{u}_y^2 &= \frac{R_{yy}-\cos\delta}{1-\cos\delta} \\ \hat{u}_x^2 &= \frac{R_{xx}-\cos\delta}{1-\cos\delta} \end{aligned} \tag{9}$$

Adding these,

$$\cos\delta = \frac{\mathrm{Tr}(R)-1}{2} \tag{10}$$

$$\begin{aligned} \hat{u}_x^2 &= 1-2\frac{1-R_{xx}}{3-\mathrm{Tr}(R)} \\ \hat{u}_y^2 &= 1-2\frac{1-R_{yy}}{3-\mathrm{Tr}(R)} \\ \hat{u}_z^2 &= 1-2\frac{1-R_{zz}}{3-\mathrm{Tr}(R)} \end{aligned} \tag{11}$$

From there, you can find the sign of $\hat{u}_x, \hat{u}_y, \hat{u}_z$ (we already know their squares) by knowing that $\sin\delta \geq 0$ (we define the `AxisAngle`that way) and

$$\begin{aligned} 2\sin\delta\hat{u}_x &= R_{zy}-R_{yz} \\ 2\sin\delta\hat{u}_y &= R_{xz}-R_{zx} \\ 2\sin\delta\hat{u}_z &= R_{yx}-R_{xy} \end{aligned}$$

We use this method, even though it involves three square roots instead of one (to normalize each component of $\hat{u}$ separately). The advantage is substantially simpler code and fewer decision branches.

Still, there are special cases which stem from the fact that a rotation by π about some direction is equivalent to rotating about the opposite direction. That is, when $\sin\delta=0$, simply multiplying the known absolute values of $(\hat{u}_x, \hat{u}_y, \hat{u}_z)$ by the signs of $R_{zy}-R_{yz}$ and so forth doesn't work: Those off-diagonal symmetric differences are zero for the non-zero components of $\hat{u}$.

This is resolved by noting that if $\cos\delta=1$, you simply return the IDENTITY Rotation which by convention is $(\hat{u}=\hat{z},\ \delta=0)$. And when $\cos\delta=-1$, this is a rotation by π, so it remains so when forcing $\hat{u}$ to be greater (in dictionary ordering) than $-\hat{u}$. This can be written in pseudo-C++ as:

```
if ( trace(R) = -1 ) {
  if ( absUz != 0 ) {
    u = UnitVector ( absUx * signum(Rxz), absUy * signum(Ryz), absUz );
  } else if ( absUy != ) {
    u = UnitVector ( absUx * signum(Rxy), absUy, 0 );
  } else {
    u = UnitVector ( 1, 0, 0 );
  }
}
```

1.6 Extracting EulerAngles from Rotation

The Euler angle rotation matrix is

$$\begin{pmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{pmatrix} \tag{12}$$

Looking at the physical definition of an Euler angles transformation, it is obvious that you can always choose $0 \leq \theta \leq \pi$ since the last two steps can trade off 180 extra degrees of ψ rotation for changing the sign of θ. We adapt that convention. Then ignoring the special cases of $\theta = 0$ and $\theta = \pi$ we can compute the Euler angles as follows:

The R_{zz} component is $\cos\theta$; given that we want $\theta \geq 0$, we immediately have

$$\theta = \cos^{-1} R_{zz} \tag{13}$$

Now to compute ϕ:

First, assume $|R_{zz}| < 1$. Then since R_{zy} is $-\sin\theta\cos\phi$, $|\phi|$ is determined by $\cos^{-1}(-R_{zy}/\sin\theta$. Since the `acos()` function always returns an angle with positive $\sin\phi$, you may need to negate this returned value. The sign of ϕ matches that of $R_{zx} = \sin\theta\sin\phi$ because $\sin\theta$ has to be positive.

But what if $|R_{zz}| < 1$ and $R_{zx} = 0$? Then $\sin\phi\sin\theta = 0$ and since $\sin\theta > 0$, ϕ must be 0 or π and $R_{zy} = \pm\sin\theta$ (which makes sense since $R_{zz}^2 + R_{zy}^2 = 1$). When $R_{zy} > 0$, $-\sin\theta\cos\phi$ has to be positive, so $\phi = \pi$. When $R_{zy} < 0$, $phi = 0$; and since $|R_{zz}| < 1$ and R_{zx} is 0, R_{zy} cannot be exactly zero.

Having covered the case of $|R_{zz}| < 1$, we move on to the case of $R_{zz} = \pm 1$. Here we have freedom in the choice of ϕ and ψ because the individual ϕ and ψ do not contribute to the rotation matrix; the only thing that matters is the combination $\phi + \psi$ (or $\phi - \psi$ if $R_{zz} = -1$). We resove the ambiguity by taking $\phi = \psi R_{zz}$. R_{xx}, which is $\cos(\phi \pm \psi)$ becomes $\cos 2\phi$; R_{xy}, which is $\sin(\phi \pm \psi)$ becomes $\sin 2\phi$. Since 2ϕ is an arc cosine, $-\pi/2 < \phi \leq \pi/2$ so $\sin\phi$ and hence ϕ has the same sign as $\sin 2\phi$ which is R_{xy}. What if R_{xy} is zero? Then, since $R_{xz} = 0$ as well, $\cos 2\phi = R_{xx} = \pm 1$. When $R_{xx} = 1$, $\phi = 0$ and when $R_{xx} = -1$, $\phi = \pi/2$.

So ϕ is given by:

$$\phi = \begin{cases} \cos^{-1}\frac{-R_{zy}}{\sin\theta}\mathrm{signum}(R_{zx}) & |R_{zz}| < 1,\ R_{zx} \neq 0 \\ 0 & |R_{zz}| < 1,\ R_{zx} = 0,\ R_{zy} < 0 \\ \pi & |R_{zz}| < 1,\ R_{zx} = 0,\ R_{zy} > 0 \\ \frac{1}{2}\cos^{-1} R_{xx}\mathrm{signum}(R_{xy}) & |R_{zz}| = 1,\ R_{xy} \neq 0 \\ 0 & |R_{zz}| = 1,\ R_{xy} = 0,\ R_{xx} = +1 \\ \pi/2 & |R_{zz}| = 1,\ R_{xy} = 0,\ R_{xx} = -1 \end{cases} \tag{14}$$

Similarly, ψ is derived from the last column in normal cases and the first column in special cases. When $|R_{zz}| < 1$, R_{yz} is $+\sin\theta\cos\psi$, $|\phi|$ is determined by $\cos^{-1}(R_{yz}/\sin\theta$. The sign of ψ matches that of $R_{xz} = \sin\theta\sin\psi$ because $\sin\theta$ has to be positive.

If $|R_{zz}| < 1$ and $R_{xz} = 0$, then ψ must be 0 or π and $R_{yz} = \pm \sin\theta$. When $R_{yz} > 0$, $\sin\theta\cos\psi$ has to be positive, so $\psi = 0$. When $R_{yz} < 0$, $\psi = \pi$.

When $R_{zz} = \pm 1$, we take $\psi = \pm\phi$. R_{xx}, which is $\cos(\phi \pm \psi)$ becomes $\cos 2\psi$; R_{yx}, which is $-\sin(\psi \pm \phi)$ becomes $-\sin 2\psi$. Since 2ψ is an arc cosine, $-\pi/2 < \psi \leq \pi/2$ so $\sin\psi$ and hence ψ has the same sign as $\sin 2\psi$ which is $-R_{yx}$. If R_{yx} is zero, then since $R_{zx} = 0$ as well, $\cos 2\psi = R_{xx} = \pm 1$. When $R_{xx} = 1$, $\psi = 0$ and when $R_{xx} = -1$, $\psi = -\pi/2$ (the negative vlaue is chosen so that we may consistently have $\psi = -\phi$ whenever $\theta = \pi$).

$$\psi = \begin{cases} \cos^{-1}\frac{+R_{yz}}{\sin\theta}\mathrm{signum}(R_{xz}) & |R_{zz}| < 1,\ R_{xz} \neq 0 \\ 0 & |R_{zz}| < 1,\ R_{xz} = 0,\ R_{yz} > 0 \\ \pi & |R_{zz}| < 1,\ R_{xz} = 0,\ R_{yz} < 0 \\ -\frac{1}{2}\cos^{-1} R_{xx}\mathrm{signum}(R_{yx}) & |R_{zz}| = 1,\ R_{yx} \neq 0 \\ 0 & |R_{zz}| = 1,\ R_{yx} = 0,\ R_{xx} = +1 \\ \pi/2 & |R_{zz}| = 1,\ R_{yx} = 0,\ R_{xx} = -1 \end{cases} \tag{15}$$

2 LorentzTransformations

2.1 Decomposition into Boost and Rotation

The decomposition of `LorentzTransformation` T into a `LorentzBoost` times a `Rotation` $T = BR$ is surprisingly easy: Consider the unit timelike 4-vector $\hat{u}_t$. Then

$$T(\hat{u}_t) = B\left(R(\hat{u}_t)\right) = B\hat{u}_t \tag{16}$$

So you can read off γ times the boost vector of boost B, in 3-vector form, from the 3-space part of the last column of T. (γ, pf course, is the time elemnet of that column.) Then

$$R = B^{-1}T \tag{17}$$

and since we alreay know how to form a general `LorentzTransformation` matrix from a pure boost, the problem is solved.

Similarly, to decompose into $T = RB$, consider the adjoint of the unit timelike 4-vector–the row $(1,\ 0,\ 0,\ 0)$–$u_t^\dagger$. Then

$$u_t^\dagger T = \left(u_t^\dagger R\right) B = u_t^\dagger B \tag{18}$$

In this case we read B off the bottom row of T, and then apply

$$R = TB^{-1} \tag{19}$$

2.2 Inversion

Of course, inverting a pure boost is much easier than inverting a general 4x4 matrix—we simply construct a boost out of $-\vec{\beta}$. This merely negates the space-time components of the boost matrix.

At first blush it looks as if the orthosymplectic property will allow trivial inversion of a general `LorentzTransformation` as well. This is not the case: About the best you could do is to decompose into a rotation and a pure boost, and use the inversion and transpose properties of each.

$$\begin{aligned} T &= \mathrm{Boost}(\beta)\left(\mathrm{Boost}(-\beta)T\right) = \mathrm{Boost}(\beta)R \\ T^{-1} &= R^{-1}\mathrm{Boost}(-\beta) = \left(\mathrm{Boost}(-\beta)T\right)^T \mathrm{Boost}(-\beta) \\ &= T^T\,\mathrm{Boost}(-\beta)^T\,\mathrm{Boost}(-\beta) = T^T\,\mathrm{Boost}(-\frac{2\beta}{1+\beta^2}) \end{aligned} \tag{20}$$

Having used the decomposition property, the inversion property of a pure boost, the fact that a rotation may be inverted by transposing, and the formula for combing two boosts along the same direction, we are still left with the need to construct a pure boost and then multiply two 4x4 matrices. Since matrix multiplication is to highest-order in N just as expensive as partial-pivot inversion (Crout or Gauss-Jordan), straightforward matrix inversion would appear to be the best you can do to invert a general `LorentzTransformation`.

This is an illusion, however. Because of the need for partial pivoting, determination of implicit scaling factors, and the increased algorithmic bookkeeping, direct inversion for 4x4 is much longer than multiplication. therefore, we use the above formula instead of matrix inversion.

2.3 Comparisons of LorentzTransformations

For comparisons other than `==` and `!=`, we will opt for quality of definition over speed of application.

The special-case "naturalness" conditions we would like to obey are

- For pure boosts along a coordinate axis, the transfromations are ordered by their β values.
- For pure boosts along the same direction, the transformations are again ordered by β.
- For pure rotations, the same ordering is obtained as for the `Rotation` class.

The obvious first step is to decompose $T_i = B_i R_i$. B_i is trivial to get, and only if $B_1 = B_2$ does one need to compute (and compare) R_1 and R_2. The primary comparison involve two pure boosts.

One possible mode of comparing pure boosts would be to compare their boost `SpaceVector`s (using dictionary ordering). This is quick but violates the second condition above whenever the two boosts are in the same "negative" direction (that is, whenever $\hat{b_1} = \hat{b_2} < (0,0,0)$ using `SpaceVector` comparison).

Instead, we define the comparison $>$ of two pure boosts with `SpaceVector`s b1 and b2 as follows:

```
  if ( (b1 >= 0) || (b1 >= 0) ) {
return b1 > b2;
  } else {
return b1 < b2;
  }
```

This is fairly fast, and obeys all three desired limits.

2.4 isNear() for LorentzTransformations

Given the angst that went into defining tolerant comparisons for `Rotation`s, and the difficulties arising from the Minkowski metric when defining isNear() for `LorentzVector`s, it would come as no shock to notice that defining isNear() for general `LorentzTransformation`s is a tough job.

A first possibility is to use the analogue of equation 4:

$$4 - \mathrm{Tr}(PQ^{-1}) \leq \epsilon^2 \qquad (21)$$

The problem with this definition is that a rotation component in PQ^{-1} decreases the trace, while a Lorentz boost component increases the trace. So there can be Lorentz transformation results $PQ^{-1} \neq I$ with trace exactly 4. This would be a horrible definition of nearness!

A second possibility, inspired by the rejected approach for Rotations, would be to look at the Frobenius norm of $I - PQ^{-1}$ and compare it to 2ϵ (the multiplier is chosen so as to match the definition for rotations when the differnce is small). This has fine properties, but for Lorentz transformations which happen to be pure rotations, one would hope that the nearness measure would agree with that used for ordinary `Rotations`. This measure instead agrees with the Frobenius norm measure for that case.

We separately provide `FrobeniusDiff()` using this Frobenius norm definition, but that should not be our primary isNear() measure.

Looking back, we selected the trace measure for the rotation case because it is slightly easier to understand, and substantially simpler to compute. Similarly, we shall see below that there is a comfortable definition of distance between two pure boosts, and it is desirable that the general case match this definition.

The strategy will be to decompose the transformation into a boost and a rotation and add the nearness measures in quadrature.

2.4.1 isNear() for LorentzBoosts

Since we have a natural scale for a boost ($c = 1$) it makes sense to talk about absolute nearness of two boosts. The first temptation is to compare the SpaceVectors of magnitude $\beta < 1$ defining the two boosts. This is not a good definition, since a boost with $\beta = 1 - 10^{-12}$ is very different from one with $\beta = 1 - 10^{-11}$.

However, there is a natural, quick and good definition, which jumps out at you if you write out the transformation in matrix form. There, the boost vector can be read off the last column, which is $(\gamma\beta_x, \gamma\beta_y, \gamma\beta_z, \gamma)$. If you take the magnitude of the difference between the spacelike parts of those columns, you get a good definition, which is small only for two boosts which have nearly identical effects on all 4-vectors.

2.4.2 isNear() for General LorentzTransformations

The strategy is to decompose the transformation into a boost and a rotation $T_i = B_i R_i$ and add the nearness measures for those pure boosts and pure rotations in quadrature.

So the first step is to take the last columns of the two rotations, and find mag2() of the three space elements of that column difference. Then apply pure boosts based on that column to each of the transformations, and find the norm of the resulting rotations.

Some speed considerations: One could use the Frobenius measure, and retain the desired matching between the special and the general case by switching to the Frobenius measure in the special cases. The special cases would become harder to compute; the general case a bit easier.

Although I would place speed of the two special cases (`Rotation` and `LorentzBoost`) as more important than speed for the fully general case, in fact the suggested definition is quite efficient some of the time when used for `isNear()`. The point is that when the transformations are different by more tha ϵ in the boost sector, the cost of determining that is trivial. Only when that sector meets the conditions do you have to decompose the transformation. The Frobenius measure, by contrast, always requires a matrix multiplication (though the sum of the squares of the last row can still be used to filter out cases where the boosts are grossly different, that still represents a fair amount of work).