Converting Between Coordinate Systems

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: April 23, 2014
Last Modified: March 14, 2016

Contents

The original version of this document was entitled *Conversion of Left-Handed Coordinates to Right-Handed Coordinates* and was written to handle the conversion of LightWave coordinate systems (left-handed) to Wild Magic coordinate systems (right-handed). The process was specific to LightWave's choice of representing rotations using Euler angles, and the discussion included how to deal with cameras, lights, and transformation hierarchies. The original document was first written in December 2003 and was last modified in March 2008.

When someone wants to convert between coordinate systems, the question is generally: "I have my coordinate system and I want to convert to someone else's coordinate system." Moreover, given an affine transformation (usually rotation) in the first coordinate system, one wants the equivalent transformation in the second coordinate system that performs the same geometric operation in the common world to which the coordinate systems are attached. This topic is generally referred to as *affine change of basis*. I have rewritten the document to show how to do this, and pseudocode is provided so that you can automate this process in a program.

1 Coordinates for a Common World

When working with multiple coordinate systems, the presumption is that these are for a common world. We need a simple affine algebra framework for this world. Let $\mathbb{R}^n$ denote the set of n-tuples whose components are real numbers, where the question of coordinate system conversion in computer graphics tends to be for $n = 2$ or $n = 3$. The discussion here applies generally, though, for any $n > 1$. To locate points in the world, we need a reference point, called the *origin*, and a set of *n linearly independent directions* in which to measure distances from the origin. There are many choices, which really is the heart of converting between coordinate systems. The *standard origin* of the world is denoted $\mathcal{O}$ and, by default, the n-tuple of measurements to locate the origin are all zero; that is, the standard origin is represented by the n-tuple $(0, \ldots, 0)$, whose components are all zero. The standard directions are chosen to be vectors $\mathbf{E}_i$ for $0 \leq i \leq n-1$, where the components of $\mathbf{E}_i$ are zero except for component i which is 1. In 2D, the origin is $(0, 0)$ and the directions are $(1, 0)$ and $(0, 1)$. In 3D, the origin is $(0, 0, 0)$ and the directions are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. Any point $\mathcal{P}$ in the world is identified by an n-tuple of signed distances from the origin, say, $\mathbf{P} = (p_0, \ldots, p_{n-1})$, where p_i is the signed distance measured from $\mathcal{P}$ in the direction $\mathbf{E}_i$.

The order of the standard directions is implied in the n-tuple; that is, the i-th component of the n-tuple is a measurement in the direction of $\mathbf{E}_i$. It is possible to rearrange the standard directions and, simultaneously, rearrange the components. The resulting n-tuple, viewed as a collection of measurements of signed distances, represents the same geometric point in the world. All we have done is change the bookkeeping, so to speak. The geometry is the same but the algebra has changed. For example, if (p_0, p_1) is the 2-tuple for the ordered direction set $\{(1, 0), (0, 1)\}$, the 2-tuple (p_1, p_0) for the ordered direction set $\{(0, 1), (1, 0)\}$ represents the same point in the 2D world. However, if you do not tell others about your bookkeeping conventions, it will be difficult to communicate information to them properly about your view of the world.

Let's consider the standard directions to be $n \times 1$ column vectors. The ordered set $\{\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\}$ is referred to as the *standard Euclidean basis* for $\mathbb{R}^n$. The ordering is said to form a *right-handed coordinate system*. If you swap the order of two of the directions, the ordering is said to form a *left-handed coordinate system*. The motivation for using the term *handedness* comes from 3D, where you discuss the right-hand or left-hand rule for cross products. This is a natural way for tagging the ordering of a set of linearly independent directions, but the ancient ones could have just as easily used a 2-tag system, say, *standard ordering* and *not-standard ordering* or even *type 1* and *type 2*. What's in a name anyway? If you create an $n \times n$ matrix $[\mathbf{E}_0 \cdots \mathbf{E}_n]$ whose columns are the specified vectors, the matrix is actually the identity matrix. The determinant of this matrix is positive. If you were to swap the order of two columns, the determinant of the resulting matrix is negative. The determinant of a matrix whose columns are from a basis is either positive or negative. If positive, we call the basis right-handed. If negative, we call the basis left-handed. The process of converting between two coordinate systems may be coded in a manner that does not require the handedness of the inputs to be specified explicitly.

2 Linear Change of Basis

Consider two observers, each using the standard origin for their coordinate systems. However, one observer decides to use n linearly independent directions that are different from the standard ones; call these $\mathbf{U}_i$ for $0 \le i \le n-1$. The set of these directions is a basis for $\mathbb{R}^n$. Another observer decides to use n linearly independent directions that are different from the standard ones and different from those of the other observer; call these $\mathbf{V}_i$ for $0 \le i \le n-1$. The set of these directions is another basis for $\mathbb{R}^n$. Both bases provide bookkeeping information for representing points in the common world. To convert from the first coordinate system to the second is a matter of converting the first to the common world's coordinate system and then converting the common world's coordinate system to the second coordinate system.

Let $\mathbf{P} = (p_0, \ldots, p_{n-1})$ be the n-tuple of signed distance measurements that locate the point in the common world. The n-tuple is referred to as the *coordinates of the point relative to the standard Euclidean basis.* The observers store the location of the point in their own coordinate systems. The first observer stores

$$\mathbf{P} = \sum_{i=0}^{n-1} x_i \mathbf{U}_i \tag{1}$$

where $(x_0, \ldots, x_{n-1})$ are referred to as the *coordinates of the point relative to the* $\mathbf{U}$*-basis.* The second observer stores

$$\mathbf{P} = \sum_{i=0}^{n-1} y_i \mathbf{V}_i \tag{2}$$

where $(y_0, \ldots, y_{n-1})$ are referred to as the *coordinates of the point relative to the* $\mathbf{V}$*-basis.* Define two $n \times n$ matrices, $U = [\mathbf{U}_0 \ \cdots \ \mathbf{U}_{n-1}]$ and $V = [\mathbf{V}_0 \ \cdots \ \mathbf{V}_{n-1}]$, each having columns that are the specified vectors. Define two $n \times 1$ vectors $\mathbf{X}$ and $\mathbf{Y}$, where the rows of $\mathbf{x}$ are the x_i and the rows of $\mathbf{Y}$ are the y_i. Equation (1) becomes $\mathbf{P} = U\mathbf{X}$ and equation (2) becomes $\mathbf{P} = V\mathbf{Y}$. Equating these, we have

$$\mathbf{Y} = V^{-1}U\mathbf{X}, \ \ \mathbf{X} = U^{-1}V\mathbf{Y} \tag{3}$$

which specifies how to convert the coordinates of one system to those of the other. Each basis vector $\mathbf{V}_i$ may be written as a linear combination of the $\mathbf{U}$-basis, namely, $\mathbf{V}_i = \sum_{j=0}^{n-1} \mathbf{U}_i c_{ij}$ for some scalars c_{ij}. The matrix $C = [c_{ij}]$ is $n \times n$ and the basis relationship in matrix form is $V = UC$, which implies

$$C = U^{-1}V \tag{4}$$

Equation (3) becomes

$$\mathbf{Y} = C^{-1}\mathbf{X}, \ \ \mathbf{X} = C\mathbf{Y} \tag{5}$$

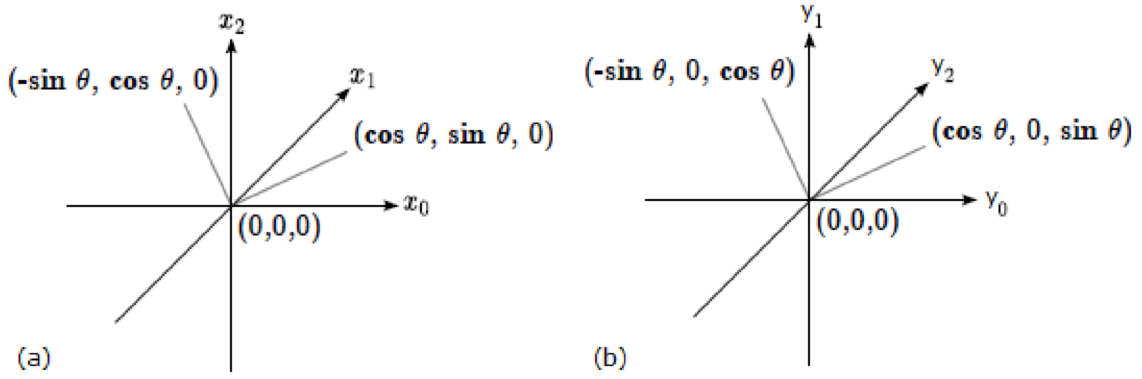
The process is referred to as a *linear change of basis.* The matrix C represents that change. Although linear algebra books refer to this solely as *change of basis*, I added the modifier *linear* to distinguish this from coordinate system conversions when the two systems have origins not equal to the standard origin of the common world. In the latter case, I call the related process an *affine change of basis.*

For example, the first observer uses the standard Euclidean basis for locating points. The basis is $\{\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2\} = \{(1,0,0),(0,1,0),(0,0,1)\}$. The second observer uses the basis $\{\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2\} = \{(1,0,0),(0,0,1),(0,1,0)\}$. The matrices of basis vectors and the change-of-basis matrix and its inverse are

$$U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = U^{-1}V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad C^{-1} = V^{-1}U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{6}$$

Note that $\det(U) > 0$, so the **U**-basis is right-handed, and $\det(V) < 0$, so the **V**-basis is left-handed. Figure 1 shows the coordinate systems. The points labeled with trigonometric components are used in an example in the next section.

Figure 1. Two coordinate systems with common origin. The system of (a) is right-handed and the system of (b) is left-handed. The x_1-axis and the y_2-axis both point into the plane of the page.



The conversions $\mathbf{X} = C\mathbf{Y}$ and $\mathbf{Y} = C^{-1}\mathbf{X}$ are $(x_0, x_1, x_2) = (y_0, y_2, y_1)$ and $(y_0, y_1, y_2) = (x_0, x_2, x_1)$, which are just swaps in the last two components. This agrees with the illustrations in Figure 1.

3 Conversion of Linear Transformations

Once we have established the change of basis, the next natural question is how to represent a geometric action in the common world as matrix transformations relative to the observers' coordinate systems. For example, consider a rotation in the common world by a positive angle θ around the up-axis. Figure 1 shows this geometric action applied to a point using coordinates relative to each coordinate system.

Define $c = \cos\theta$ and $s = \sin\theta$. In Figure 1(a), the point $(x_0, x_1, x_2) = (1,0,0)$ is rotated to $(x'_0, x'_1, x'_2) = (c, s, 0)$, the point $(x_0, x_1, x_2) = (0,1,0)$ is rotated to $(x'_0, x'_1, x'_2) = (-s, c, 0)$, and the point $(x_0, x_1, x_2) = (0,0,1)$ is unchanged by the rotation. The matrix R that represents the rotation with respect to the **U**-basis is

$$\mathbf{X}' = \begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = R\mathbf{X} \tag{7}$$

In Figure 1(b), the point $(y_0, y_1, y_2) = (1, 0, 0)$ is rotated to $(y'_0, y'_1, y'_2) = (c, 0, s)$, the point $(y_0, y_1, y_2) = (0, 0, 1)$ is rotated to $(y'_0, y'_1, y'_2) = (-s, 0, c)$, and the point $(y_0, y_1, y_2) = (0, 1, 0)$ is unchanged by the rotation. The matrix $\bar{R}$ that represents the rotation with respect to the $\mathbf{V}$-basis is

$$\mathbf{Y}' = \begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \bar{R}\mathbf{Y} \tag{8}$$

We can obtain the relationship between R and $\bar{R}$ by applying equation (3) to equations (7) and (8). Specifically, the change of basis leads to $\mathbf{X} = C\mathbf{Y}$, $\mathbf{X}' = C\mathbf{Y}'$, $\mathbf{Y} = C^{-1}\mathbf{X}$, and $\mathbf{Y}' = C^{-1}\mathbf{X}'$. This leads to $C\mathbf{Y}' = RC\mathbf{Y}$ and $C^{-1}\mathbf{X}' = \bar{R}C^{-1}\mathbf{X}$, $\mathbf{Y}' = C^{-1}RC\mathbf{Y}$ and $\mathbf{X}' = C\bar{R}C^{-1}\mathbf{X}$, and subsequently

$$\bar{R} = C^{-1}RC, \quad R = C\bar{R}C^{-1} \tag{9}$$

These are referred to as *similarity transformations*. Generally, given a matrix transformation M relative to the $\mathbf{U}$-basis and the change of basis matrix C to convert to the $\mathbf{V}$-basis, the matrix transformation in the latter basis is $\bar{M} = C^{-1}MC$.

4 Affine Change of Basis

The process for linear change of basis applies when the two observers use the same origin for their coordinate systems. However, in some applications the origins might be different. Let the origin for the first coordinate system be located at $\mathbf{E}$ and the origin for the second coordinate system be located at $\mathbf{F}$, neither origin necessarily the standard one. We will use the same basis vectors as before. A point $\mathcal{P}$ in the world now has representations that are translations of those in equations (1) and (2), namely,

$$\mathbf{P} = \mathbf{E} + \sum_{i=0}^{n-1} x_i \mathbf{U}_i \tag{10}$$

for the first observer and

$$\mathbf{P} = \mathbf{F} + \sum_{i=0}^{n-1} y_i \mathbf{V}_i \tag{11}$$

for the second observer.

In matrix-vector form, the representations are $\mathbf{P} = \mathbf{E} + U\mathbf{X}$ and $\mathbf{P} = \mathbf{F} + V\mathbf{Y}$. For matrix-vector notation, we may use 4×4 matrices written in block form and store the point representations in homogenous form,

$$\begin{bmatrix} U & \mathbf{E} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{P} \\ 1 \end{bmatrix} = \begin{bmatrix} V & \mathbf{F} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \tag{12}$$

We may solve for either coordinate vector by inversion,

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} = A^{-1} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = A \begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \tag{13}$$

where

$$A = \begin{bmatrix} U & \mathbf{E} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}^{-1} \begin{bmatrix} V & \mathbf{F} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} = \begin{bmatrix} U^{-1}V & U^{-1}(\mathbf{F}-\mathbf{E}) \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}, \tag{14}$$

and

$$A^{-1} = \begin{bmatrix} V & \mathbf{F} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}^{-1} \begin{bmatrix} U & \mathbf{E} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} = \begin{bmatrix} V^{-1}U & V^{-1}(\mathbf{E}-\mathbf{F}) \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \tag{15}$$

5 Conversion of Affine Transformations

The conversion between matrix representations of affine transformations is similar to that of linear transformations. An affine transformation for the first coordinate system is $\mathbf{X}' = M\mathbf{X} + \mathbf{T}$. In homogeneous form,

$$\begin{bmatrix} \mathbf{X}' \\ 1 \end{bmatrix} = \begin{bmatrix} M & \mathbf{T} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = H \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \tag{16}$$

where the last equality defines the 4×4 matrix H. Let the transformation for the second coordinate system representing the same action in the common world be

$$\begin{bmatrix} \mathbf{Y}' \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{M} & \bar{\mathbf{T}} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} = \bar{H} \begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \tag{17}$$

where the last equality defines the 4×4 matrix $\bar{H}$. Using the affine change of basis from equation (13), we obtain the relationships

$$\bar{H} = A^{-1}HA, \quad H = A\bar{H}A^{-1} \tag{18}$$

The computation of $\bar{H}$ from H depends on the assumption that we are using the *vector-on-the-right* convention for matrix-vector multiplication; that is, to transform an $n \times 1$ vector $\mathbf{W}$ by an $n \times n$ matrix H, we consider $\mathbf{W}$ a $n \times 1$ vector and multiply using $H\mathbf{W}$. The *vector-on-the-left* convention performs the multiplication by $\mathbf{W}^\mathsf{T}H$. When H is not a symmetric matrix, these products are generally different: written as $n \times 1$ outputs, the two conventions produce $H\mathbf{W}$ and $H^\mathsf{T}\mathbf{W}$. The multiplication convention must be taken into account when converting transformations between coordinate systems.

6 An Implementation

The source code shown next uses the GTEngine, but it should be straightforward to write your own with whatever mathematics library you use. Some samples are mentioned in the comments at the top of the file GteConvertCoordinates.h.

```
// David Eberly, Geometric Tools, Redmond WA 98052
// Copyright (c) 1998-2016
// Distributed under the Boost Software License, Version 1.0.
// http://www.boost.org/LICENSE_1_0.txt
// http://www.geometrictools.com/License/Boost/LICENSE_1_0.txt
// File Version: 2.1.1 (2016/03/14)

#pragma once

#include "GteMatrix.h"
#include "GteGaussianElimination.h"

// Convert points and transformations between two coordinate systems.
// The mathematics involves a change of basis.  See the document
//   http://www.geometrictools.com/Documentation/ConvertingBetweenCoordinateSystems.pdf
// for the details.  Typical usage for 3D conversion is shown next.
//
// // Linear change of basis.
// ConvertCoordinates<3, double> convert;
// Vector<3, double> X, Y, P0, P1, diff;
// Matrix<3, 3, double> U, V, A, B;
// bool isRHU, isRHV;
// U.SetCol(0, Vector3<double>{1.0, 0.0, 0.0});
// U.SetCol(1, Vector3<double>{0.0, 1.0, 0.0});
// U.SetCol(2, Vector3<double>{0.0, 0.0, 1.0});
// V.SetCol(0, Vector3<double>{1.0, 0.0, 0.0});
// V.SetCol(1, Vector3<double>{0.0, 0.0, 1.0});
// V.SetCol(2, Vector3<double>{0.0, 1.0, 0.0});
// convert(U, true, V, true);
// isRHU = convert.IsRightHandedU();  // true
// isRHV = convert.IsRightHandedV();  // false
// X = { 1.0, 2.0, 3.0 };
// Y = convert.UToV(X);  // { 1.0, 3.0, 2.0 }
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0 }
// Y = { 0.0, 1.0, 2.0 };
// X = convert.VToU(Y);  // { 0.0, 2.0, 1.0 }
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0 }
// double cs = 0.6, sn = 0.8;  // cs*cs + sn*sn = 1
// A.SetCol(0, Vector3<double>{  c,   s, 0.0});
// A.SetCol(1, Vector3<double>{ -s,   c, 0.0});
// A.SetCol(2, Vector3<double>{0.0, 0.0, 1.0});
// B = convert.UToV(A);
//   // B.GetCol(0) = { c, 0, s}
//   // B.GetCol(1) = { 0, 1, 0}
//   // B.GetCol(2) = {-s, 0, c}
// X = A*X;  // U is VOR
// Y = B*Y;  // V is VOR
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0 }
//
// // Affine change of basis.
// ConvertCoordinates<4, double> convert;
// Vector<4, double> X, Y, P0, P1, diff;
// Matrix<4, 4, double> U, V, A, B;
// bool isRHU, isRHV;
// U.SetCol(0, Vector4<double>{-1.0, 0.0, 0.0, 0.0});
// U.SetCol(1, Vector4<double>{0.0, 0.0, 1.0, 0.0});
// U.SetCol(2, Vector4<double>{0.0, -1.0, 0.0, 0.0});
// U.SetCol(3, Vector4<double>{1.0, 2.0, 3.0, 1.0});
// V.SetCol(0, Vector4<double>{0.0, 1.0, 0.0, 0.0});
// V.SetCol(1, Vector4<double>{-1.0, 0.0, 0.0, 0.0});
// V.SetCol(2, Vector4<double>{0.0, 0.0, 1.0, 0.0});
// V.SetCol(3, Vector4<double>{4.0, 5.0, 6.0, 1.0});
// convert(U, true, V, false);
// isRHU = convert.IsRightHandedU();  // false
// isRHV = convert.IsRightHandedV();  // true
// X = { -1.0, 4.0, -3.0, 1.0 };
// Y = convert.UToV(X);  // { 0.0, 2.0, 1.0, 1.0 }
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0, 0 }
// Y = { 1.0, 2.0, 3.0, 1.0 };
// X = convert.VToU(Y);  // { -1.0, 6.0, -4.0, 1.0 }
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0, 0 }
// double c = 0.6, s = 0.8;  // c*c + s*s = 1
// A.SetCol(0, Vector4<double>{  c,   s,    0.0, 0.0});
// A.SetCol(1, Vector4<double>{ -s,   c,    0.0, 0.0});
// A.SetCol(2, Vector4<double>{0.0, 0.0,   1.0, 0.0});
// A.SetCol(3, Vector4<double>{0.3, 1.0,  -2.0, 1.0});
// B = convert.UToV(A);
// // B.GetCol(0) = {   1,     0,     0, 0 }
// // B.GetCol(1) = {   0,     c,     s, 0 }
// // B.GetCol(2) = {   0,    -s,     c, 0 }
// // B.GetCol(3) = { 2.0,  -0.9,  -2.6, 1 }
// X = A*X;  // U is VOR
// Y = Y*B;  // V is VOL (not VOR)
// P0 = U*X;
// P1 = V*Y;
// diff = P0 - P1;  // { 0, 0, 0, 0 }

namespace gte
{

template <int N, typename Real>
class ConvertCoordinates
{
public:
    // Construction of the change of basis matrix.  The implementation
    // supports both linear change of basis and affine change of basis.
    ConvertCoordinates();

    // Compute a change of basis between two coordinate systems.  The return
    // value is 'true' iff U and V are invertible.  The matrix-vector
    // multiplication conventions affect the conversion of matrix
    // transformations.  The Boolean inputs indicate how you want the matrices
    // to be interpreted when applied as transformations of a vector.
    bool operator()(
        Matrix<N, N, Real> const& U, bool vectorOnRightU,
        Matrix<N, N, Real> const& V, bool vectorOnRightV);

    // Member access.
    inline Matrix<N, N, Real> const& GetC() const;
    inline Matrix<N, N, Real> const& GetInverseC() const;
    inline bool IsVectorOnRightU() const;
    inline bool IsVectorOnRightV() const;
    inline bool IsRightHandedU() const;
    inline bool IsRightHandedV() const;

    // Convert points between coordinate systems.  The names of the systems
    // are U and V to make it clear which inputs of operator() they are
    // associated with.  The X vector stores coordinates for the U-system and
    // the Y vector stores coordinates for the V-system.

    // Y = C^{-1}*X
    inline Vector<N, Real> UToV(Vector<N, Real> const& X) const;

    // X = C*Y
    inline Vector<N, Real> VToU(Vector<N, Real> const& Y) const;

    // Convert transformations between coordinate systems.  The outputs are
    // computed according to the tables shown before the function
    // declarations. The superscript T denotes the transpose operator.
    // vectorOnRightU = true:  transformation is X' = A*X
    // vectorOnRightU = false: transformation is (X')^T = X^T*A
    // vectorOnRightV = true:  transformation is Y' = B*Y
    // vectorOnRightV = false: transformation is (Y')^T = Y^T*B
    // vectorOnRightU   | vectorOnRightV   | output
    // -----------------+------------------+-------------------
    // true             | true             | C^{-1} * A * C
    // true             | false            | (C^{-1} * A * C)^T
    // false            | true             | C^{-1} * A^T * C
    // false            | false            | (C^{-1} * A^T * C)^T
    Matrix<N, N, Real> UToV(Matrix<N, N, Real> const& A) const;

    // vectorOnRightU   | vectorOnRightV   | output
    // -----------------+------------------+-------------------
    // true             | true             | C * B * C^{-1}
    // true             | false            | C * B^T * C^{-1}
    // false            | true             | (C * B * C^{-1})^T
    // false            | false            | (C * B^T * C^{-1})^T
    Matrix<N, N, Real> VToU(Matrix<N, N, Real> const& B) const;

private:
    // C = U^{-1}*V, C^{-1} = V^{-1}*U
    Matrix<N, N, Real> mC, mInverseC;
    bool mIsVectorOnRightU, mIsVectorOnRightV;
    bool mIsRightHandedU, mIsRightHandedV;
};

//----------------------------------------------------------------------------
template <int N, typename Real>
ConvertCoordinates<N, Real>::ConvertCoordinates()
    :
    mIsVectorOnRightU(true),
    mIsVectorOnRightV(true),
    mIsRightHandedU(true),
    mIsRightHandedV(true)
{
    mC.MakeIdentity();
    mInverseC.MakeIdentity();
}
//----------------------------------------------------------------------------
template <int N, typename Real>
bool ConvertCoordinates<N, Real>::operator()(
    Matrix<N, N, Real> const& U, bool vectorOnRightU,
    Matrix<N, N, Real> const& V, bool vectorOnRightV)
{
    // Initialize in case of early exit.
    mC.MakeIdentity();
    mInverseC.MakeIdentity();
    mIsVectorOnRightU = true;
    mIsVectorOnRightV = true;
    mIsRightHandedU = true;
    mIsRightHandedV = true;

    Matrix<N, N, Real> inverseU;
    Real determinantU;
    bool invertibleU = GaussianElimination<Real>()(N, &U[0], &inverseU[0],
        determinantU, nullptr, nullptr, nullptr, 0, nullptr);
    if (!invertibleU)
    {
        return false;
    }

    Matrix<N, N, Real> inverseV;
    Real determinantV;
    bool invertibleV = GaussianElimination<Real>()(N, &V[0], &inverseV[0],
        determinantV, nullptr, nullptr, nullptr, 0, nullptr);
    if (!invertibleV)
    {
        return false;
    }

    mC = inverseU * V;
    mInverseC = inverseV * U;
    mIsVectorOnRightU = vectorOnRightU;
    mIsVectorOnRightV = vectorOnRightV;
    mIsRightHandedU = (determinantU > (Real)0);
    mIsRightHandedV = (determinantV > (Real)0);
    return true;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
Matrix<N, N, Real> const& ConvertCoordinates<N, Real>::GetC() const
{
    return mC;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
Matrix<N, N, Real> const& ConvertCoordinates<N, Real>::GetInverseC() const
{
    return mInverseC;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
bool ConvertCoordinates<N, Real>::IsVectorOnRightU() const
{
    return mIsVectorOnRightU;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
bool ConvertCoordinates<N, Real>::IsVectorOnRightV() const
{
    return mIsVectorOnRightV;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
bool ConvertCoordinates<N, Real>::IsRightHandedU() const
{
    return mIsRightHandedU;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
bool ConvertCoordinates<N, Real>::IsRightHandedV() const
{
    return mIsRightHandedV;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
Vector<N, Real> ConvertCoordinates<N, Real>::UToV(Vector<N, Real> const& X)
    const
{
    return mInverseC * X;
}
//----------------------------------------------------------------------------
template <int N, typename Real> inline
Vector<N, Real> ConvertCoordinates<N, Real>::VToU(Vector<N, Real> const& Y)
    const
{
    return mC * Y;
}
//----------------------------------------------------------------------------
template <int N, typename Real>
Matrix<N, N, Real> ConvertCoordinates<N, Real>::UToV(
    Matrix<N, N, Real> const& A) const
{
    Matrix<N, N, Real> product;

    if (mIsVectorOnRightU)
    {
        product = mInverseC * A * mC;
        if (mIsVectorOnRightV)
        {
            return product;
        }
        else
        {
            return Transpose(product);
        }
    }
    else
    {
        product = mInverseC * MultiplyATB(A, mC);
        if (mIsVectorOnRightV)
        {
            return product;
        }
        else
        {
            return Transpose(product);
        }
    }
}
//----------------------------------------------------------------------------
template <int N, typename Real>
Matrix<N, N, Real> ConvertCoordinates<N, Real>::VToU(
    Matrix<N, N, Real> const& B) const
{
    // vectorOnRightU  | vectorOnRightV  | output
    // ----------------+-----------------+---------------------
    // true            | true            | C * B * C^{-1}
    // true            | false           | C * B^T * C^{-1}
    // false           | true            | (C * B * C^{-1})^T
    // false           | false           | (C * B^T * C^{-1})^T
    Matrix<N, N, Real> product;

    if (mIsVectorOnRightV)
    {
        product = mC * B * mInverseC;
        if (mIsVectorOnRightU)
        {
            return product;
        }
        else
        {
            return Transpose(product);
        }
    }
    else
    {
        product = mC * MultiplyATB(B, mInverseC);
        if (mIsVectorOnRightU)
        {
            return product;
        }
        else
        {
            return Transpose(product);
        }
    }
}
//----------------------------------------------------------------------------

}
```