# Using Diamond Coordinates to Power a Differential Drive

By Douglas Taylor
rChordata, LLC

# Summary

This document examines the problem of taking output from a device such as a joystick and applying it to a differential drive system. If the output data is in the form of an X axis value and a Y axis value of a Cartesian coordinate system, there are some hurdles to using that data for a differential drive. This document explores converting the Cartesian values to a different coordinate system which can in turn be used directly on the left and right sides of the drive.

We use the name "Diamond coordinate system" for the intermediate coordinate system that will map to the differential drive. This document describes the algorithm to do the conversion between the two systems. It also outlines an associated software package that encapsulates this algorithm into an easy-to-use Dynamic Linked Library for use in Dot Net applications.

# The Problem

In robotics there are several different drive mechanisms. Several of these mechanisms can be logically grouped as differential drives. These might include two wheeled drives, tank tread drives, and skid steer drives. If these systems are used with telepresence input systems, they are typically being controlled by a joystick or something equivalent.

The problem is that the joystick operates on a Cartesian coordinate system. It outputs values on X and Y perpendicular axes. There must be some mechanism to convert the Y axis and X axis values to speed and direction for the two logical wheel motors. At first, it seems that you would take the Y axis value and simply add or subtract the X axis values to get the left and right motor speeds. Unfortunately, this causes problems with maximum Y values as you cannot add any additional value from the X axis when turning.

Different solutions could include limiting the maximum Y values so that turning would still be possible, but this would reduce the maximum speed in a straight line. Other mathematical solutions would be possible, but the logic would be tedious. The solution provided here would be "off the shelf" and simple to use.

# Diamond Coordinate System

The Cartesian coordinate system uses two axes that are set perpendicular to each other. For the purposes of this discussion, we will consider the Cartesian system to have the 0,0 point in the center with positive X to the right, negative X to the left, positive Y going up and negative Y going down.

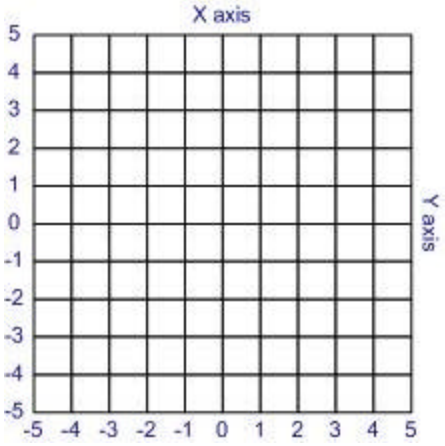


**Figure 1**

The Diamond coordinate system uses diagonals instead of up and down and left and right. Try to imagine a Cartesian coordinate system turned on its side 45°. One of the axes, let's call it Left, will run from the upper right to the lower left. The other axis is perpendicular to it and we'll call it Right. This axis will run from the upper left to the lower right. The center of this system will be 0,0. This means the 0 position of the Left axis and the 0 position of the Right axis will intersect in the middle of the system.

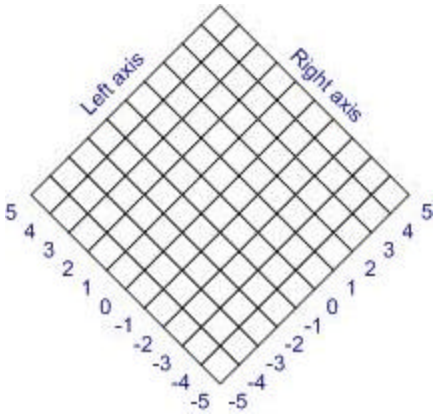


**Figure 2**

The advantage of this system is that the power or speed of the left drive motor can be directly pulled from the value of the Left axis and likewise for the right drive motor.

# Differential Drive

Differential Drive is a term used to describe a drive system where turning is achieved by applying different levels of power to the left and right sides of the drive. Power is converted to speed. If you applied more power or speed to the left side of this type of system it would turn to the right and vice versa. There are several types of drive systems that can be classified as differential drive. Two wheeled drives typically have a rear or sometimes a front and rear castor that supports weight, but does not participate in locomotion. Tank treads have two belts, one on each side of the device. Skid steer is similar to Tank treads but without the belt.

# Alternative to Diamond Coordinate Translation

Frequently, when faced with this problem, programmers have used Polar coordinate translations to solve the problem. In a Polar coordinate system a point is referenced by an angle and a radius. The advantage in using this system is that rotation about the center point is a trivial task. The user would first translate the point in Cartesian coordinates to a Polar coordinate system using Trigonometric translation functions, rotate the point by 45°, then translate the point back into Cartesian coordinates. This allows the user to use the X and Y axes as Left and Right axes. Care must be given to ensure that outlying points do not exceed maximum values after the translation.

This methodology is described in depth in the field of mathematics and will not be covered in this document. An example using this approach can be found in the Simple Dashboard service that Ben Axelrod wrote:

http://www.benaxelrod.com/MSRS/index.html

# Translation from a Cartesian Coordinate System

To use the Diamond coordinate system with a Differential Drive, you would typically have to convert values from a Cartesian coordinate system. An example would be to convert the values from a joystick controller to a vehicle with tank treads. The following would be steps in the algorithm.

## *Determine the maximum size*

First you must determine the maximum values of the Cartesian coordinate system that you will be using. In the case of the previous example, it would be the maximum values that the joystick controller will be using. We are expecting the 0,0 point to be in the center of the system and the vertical and horizontal scale to be the same. This means that the maximum positive X, the maximum negative X, the maximum positive Y and the maximum negative Y should all be the same number. For example, it might be system that goes from -1000 to +1000 on each axis, or a fractional number from -1 to +1 on each axis.

## Overlay the Cartesian system with the Diamond system

Once the scale is determined, you would then overlay the Cartesian coordinate system with the Diamond coordinate system. The Left Max, Right Max point should line up with the X 0, Y Max point. The Left Max, Right Min point should line up with the X Max, Y 0 point. The Left Min, Right Min point should line up with the X 0, Y Min point. The Left Min, Right Max point should line up with the X Min, Y 0 point. Finally the 0,0 points should line up on both systems.

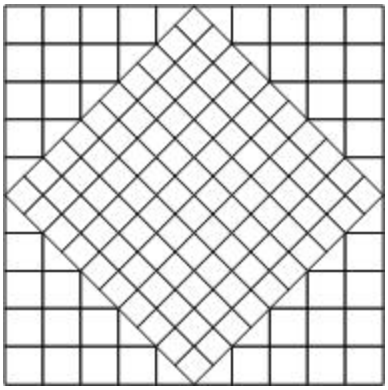

**Figure 3**

Note that this is a mental exercise and no actual code is required here.

## Extrapolate the point to within the Diamond system

You will notice that half of the Cartesian points lie outside for the Diamond system. We must account for this by extrapolating outlying points onto the Diamond system.

To determine if the subject point is outside of the Diamond system we will use an algorithm that tests whether a point is outside of a polygon.

The polygon is referenced as an array of points. First compare the subject point to all the points in the array. If it equals any of them, consider the point to be outside the polygon. Next, create a ray from the subject point horizontally to the right. Look at each line segment in the polygon and test to see if they intersect the ray. Count the number of intersections. An even number means the subject point is outside of the polygon, an odd number means the subject point is inside the polygon. For example:

```
// start with last point hence the closing line segment
p1 = polygon[polygon.GetLength(0) - 1];
for (indx = 0; indx < polygon.GetLength(0); indx++)
{

    p2 = polygon[indx];
    if (testPoint.Y > System.Math.Min(p1.Y, p2.Y))
    {
        if (testPoint.Y <= System.Math.Max(p1.Y, p2.Y))
        {
            if (testPoint.X <= System.Math.Max(p1.X, p2.X))
            {
                // horizontal segments are invalid
                if (p1.Y != p2.Y)
                {
                    xcross = (testPoint.Y - p1.Y) * (p2.X - p1.X)
                        / (p2.Y - p1.Y) + p1.X;
                    if (p1.X == p2.X || testPoint.X <= xcross)
                    {
                        counter++;
                    }
                }
            }
        }
    }
    p1 = p2;
}
ret = (counter % 2 != 0);
```

**Equation 1**

If the Cartesian point is already within the bounds of the Diamond system, you do not need to do any extrapolation. If the point does lie outside of the Diamond system, first you must construct a line from the original point to the point of origin (0,0).

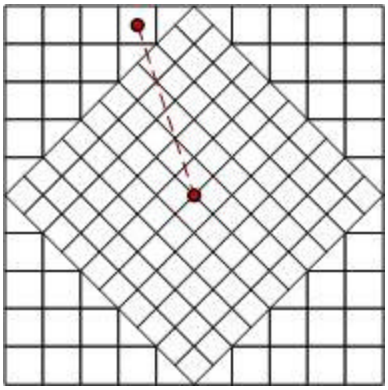

**Figure 4**

Find the intersection of this line and the outside edge of the Diamond coordinate system. This intersection is the extrapolated point.

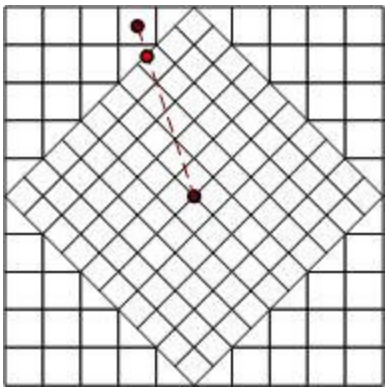

**Figure 5**

The algorithm to find the intersection requires two lines defined by two points each. For the purposes of this discussion we will define the lines by the points: L1P1, L1P2, L2P1, and L2P2. This algorithm does not work if either line is vertical or if the lines are parallel. Those conditions must be handled outside of this algorithm. First ensure the P1 points are to the left of the P2 points on the X axis.

```
if (l1p2.X < l1p1.X)
{
    tempPoint = l1p1;
    l1p1 = l1p2;
    l1p2 = tempPoint;
}
if (l2p2.X < l2p1.X)
{
    tempPoint = l2p1;
    l2p1 = l2p2;
    l2p2 = tempPoint;
}
```

**Equation 2**

Now calculate the slope of each line:

```
slope1 = ((double)(l1p2.Y - l1p1.Y)) / ((double)(l1p2.X - l1p1.X));
slope2 = ((double)(l2p2.Y - l2p1.Y)) / ((double)(l2p2.X - l2p1.X));
```

**Equation 3**

Calculate the intersepts of the slopes through the P1 points of each line:

```
intercept1 = -((slope1 * l1p1.X) - l1p1.Y);
intercept2 = -((slope2 * l2p1.X) - l2p1.Y);
```

**Equation 4**

Finally calculate the X and Y values of the intersection point:

```
intersection.X = (double)((intercept2 - intercept1)
    / (slope1 - slope2));
intersection.Y = (double)((slope1 * intersection.X) + intercept1);
```

**Equation 5**

## *Convert the point to Diamond coordinates*

Now you must convert the point from the Cartesian system to the Diamond system. Start by constructing a ray from the point to the Northwest exactly 45° from the vertical or horizontal. In other words, construct a ray that is perpendicular to the Left axis.

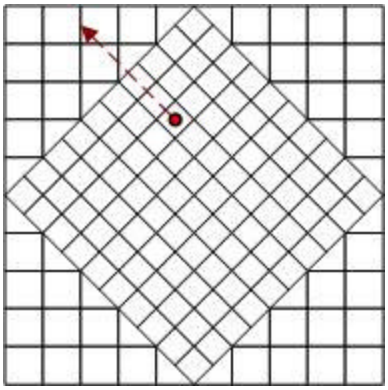

**Figure 6**

To do this we construct a line with an endpoint far outside of the Cartesian coordinate's maximum values. Simply subtract the same value from the X value as you add to the Y value. In the case of this algorithm we used a value that is twice as large as the maximum Cartesian value.

```
DoublePoint leftOuterEnd =
    new DoublePoint(
        cartPoint.X - (2 * radius), cartPoint.Y + (2 * radius));
```

**Equation 6**

Next you need to find the Cartesian point that intersects that ray with the Diamond's Left axis. Note here that if you had to extrapolate the original point from the upper left part of the Cartesian system, then it is already on the Diamond's Left axis.

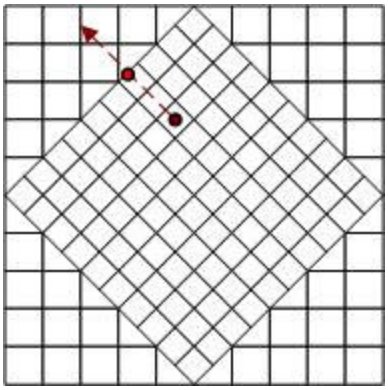

**Figure 7**

You can do this with the same intersection algorithm discussed earlier using the edge of the Diamond coordinate system as the second line: (0, maximum) (-maximum, 0)

```
// Get the point on the ray that intesects the upper left diamond edge
DoublePoint leftIntersect = GetIntersection(
    cartPoint,      // Ray from Cartesian point
    leftOuterEnd,   //   beyond the diamond at 45 degrees Northwest
    new DoublePoint(0, radius),     // Line from North point
    new DoublePoint(-radius, 0));   //   to West point
```

**Equation 7**

At this point you are still working with Cartesian coordinates. Now notice that entire Left axis of the Diamond system can be mapped onto the positive values of the Y axis of the Cartesian system.

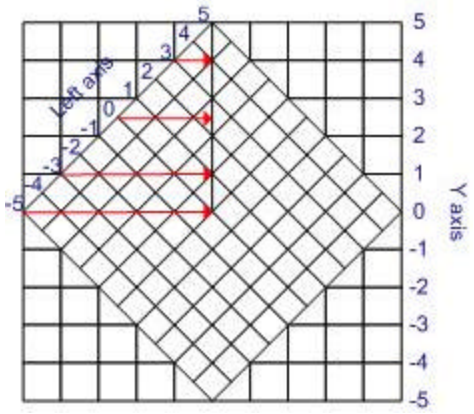


**Figure 8**

To use this relationship, you need to find the Y value of the new intersection point and scale it to find the Left value. Notice that we are still talking about points in the Cartesian system. First we take the Y value of the intersection point. We subtract half of the maximum value and then scale it by doubling the result.

```
double leftScale = (leftIntersect.Y - (radius / 2)) * 2;
```

**Equation 8**

Now simply repeat this process on the other side to find the right axis.

```
// Create a ray from the Cartesian point at a 45 degree angle through
//   the upper right side of the diamond
DoublePoint rightOuterEnd =
    new DoublePoint(
        cartPoint.X + (2 * radius), cartPoint.Y + (2 * radius));
// Get the point on the ray that intesects the upper right diamond edge
DoublePoint rightIntersect = GetIntersection(
    cartPoint,      // Ray from Cartesian point
    rightOuterEnd,  //   beyond the diamond at 45 degrees Northeast
    new DoublePoint(0, radius),     // Line from North point
    new DoublePoint(radius, 0));    //   to East point
// Use the Cartesian Y values to find the diamond scale values
double rightScale = (rightIntersect.Y - (radius / 2)) * 2;
```

**Equation 9**

# GeometryToolbox Software

You now have enough information to build your own software to convert Cartesian coordinates into Diamond coordinates for use in a differential drive; however rChordata, LLC has produced a simple to use DLL containing all of the algorithms described in this document.

## *Disclaimer*

The software referenced by this document is contained within the file DiamondToolbox.DLL and shall be referred to as “the software”. The software is copyright protected © 2009-2010 by rChordata, LLC.

Unless stated otherwise, the software is provided free of charge.

As well, the software is provided on an "as is" basis without warranty of any kind, express or implied.

Under no circumstances and under no legal theory, whether in tort, contract, or otherwise, shall rChordata, LLC, or any representative of rChordata, LLC be liable to you or to any other person for any indirect, special, incidental, or consequential damages of any character including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or for any and all other damages or losses.

If you do not agree with these terms, then you may not use the software.

## *What is in the software*

The software is contained in a file called DiamondToolbox.DLL and is written in C# (a Dot Net language). Inside the DLL are three entities. These are public classes named DiamondToolbox, DoublePoint, and DiamondPoint. All three of these classes fall under the namespace “rChordata”.

## *DoublePoint*

The class rChordata.DoublePoint is similar to the structure System.Drawing.Point except that the type used for the X and Y coordinates are of the System.Double type. The primary methods of the DiamondToolbox class use this instead of System.Drawing.Point for fractional precision.

```
public class DoublePoint
{
    public DoublePoint(double x, double y);
    public double X { get; set; }
    public double Y { get; set; }
}
```

## DiamondPoint

The class rChordata.DiamonPoint is similar to the class rChordata.DoublePoint except that the two axes are named Left and Right. The primary method of the DiamondToolbox class returns a value of this type and references values of a Diamond coordinate system.

```
public class DiamondPoint
{
    public DiamondPoint(double left, double right);
    public double Left { get; set; }
    public double Right { get; set; }
}
```

## DiamondToolbox

The class rChordata.DiamondToolbox is the main class in the DLL and contains all of the functionality. This class contains three public static methods, two of which are overloaded. These are “CartesianToDiamond”, “IsInsidePolygon”, and “GetIntersection”.

```
public class DiamondToolbox
{
    public static DiamondPoint CartesianToDiamond(
        DoublePoint cartPoint,
        double radius);

    public static bool IsInsidePolygon(
        Point[] polygon,
        Point testPoint);

    public static bool IsInsidePolygon(
        DoublePoint[] polygon,
        DoublePoint testPoint);

    public static Point GetIntersection(
        Point l1p1,
        Point l1p2,
        Point l2p1,
        Point l2p2);

    public static DoublePoint GetIntersection(
        DoublePoint l1p1,
        DoublePoint l1p2,
        DoublePoint l2p1,
        DoublePoint l2p2);
}
```

### *CartesianToDiamond*

This is the primary method in the DiamondToolbox class. This method converts a Cartesian coordinate into a Diamond coordinate. The parameters are a Cartesian point of

type rChordata.DoublePoint and a value representing the maximum Cartesian value positive or negative on either axis of type System.Double. The method returns a value of type rChordata.DiamondPoint representing the Cartesian point converted to the Diamond system.

### *IsInsidePolygon*

This method is used internally by CartesianToDiamond(), but the author thought it might have value and made it publically available. This method is overloaded with a System.Drawing.Point version and an rChordata.DoublePoint version.

The method takes a polygon represented by an array of points and a test point. It returns True if the test point is within the polygon and False if it is not.

### *GetIntersection*

This method is used internally by CartesianToDiamond(), but the author thought it might have value and made it publically available. This method is overloaded with a System.Drawing.Point version and an rChordata.DoublePoint version.

The method takes two lines each represented by a pair of points on the line. It returns the point of intersection of the two lines. It throws an exception if either line is vertical or if the lines are parallel.

## *Using the software*

This software requires ".NET Framework 3.5" and as such it requires an operating system compatible with that framework.

Typically you would create a project in some Dot Net language in Microsoft Visual Studio®. The file DiamondToolbox.DLL should be located in a folder that is easily accessible from the project.

In the Project Explorer, right click on the References folder under the project folder. When the "Add Reference" form opens, select the Browse tab. Find the DiamondToolbox.DLL file, select it and click the OK button.

In the class file in which you wish to use the DiamondToolbox functionality you should add a "using" statement at the top. For example:

```
using rChordata;
```

Call the CartesianToDiamond method to get the Diamond coordinates. For example:

```
DoublePoint cartPoint = new DoublePoint(0,0.5); // half speed ahead
DiamondPoint diaPoint = null;
diaPoint = DiamondToolbox.CartesianToDiamond(cartPoint, 1);
motor.left = diaPoint.Left;
motor.right = diaPoint.Right;
```

## Download

You can download this document and a full kit at:
http://www.rchordata.com/DownloadDiamondCoordinatePg.aspx

## Feedback

Please send feedback via http://www.rchordata.com/ContactPg.aspx