Two-Dimensional Translations, Rotations, and Intersections Using C++

by Robert J. Yager

Approved for public release; distribution is unlimited.
NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
Two-Dimensional Translations, Rotations, and Intersections Using C++

Robert J. Yager
Weapons and Materials Research Directorate, ARL

Approved for public release; distribution is unlimited.
**Two-Dimensional Translations, Rotations, and Intersections Using C++**

**ABSTRACT**

Two-dimensional operations, such as rotations, translations, and intersections, are tools that are essential for many types of scientific modeling. However, the C++ programming language does not natively perform them.
Contents

List of Figures iv
Acknowledgments v

1. Introduction 1

2. Translation of a Point in Space 1
   2.1 Derivation ................................................................. 1
   2.2 C++ Implementation .................................................. 1

3. Rotation of a Point About an Arbitrarily Positioned Axis 3
   3.1 Derivation ...................................................................... 3
   3.2 C++ Implementation .................................................... 4
   Rmatrix2D() Code .............................................................. 4

4. Intersection Between Two Lines 6
   4.1 Derivation ...................................................................... 6
   4.2 C++ Implementation .................................................... 8

5. Summary 10

Distribution List 12
List of Figures

Figure 1. Translate() example. .............................................................................2
Figure 2. Rotate() example. .....................................................................................5
Figure 3. Intersecting lines A and B ..........................................................................7
Figure 4. Intersect2D() example. .............................................................................9
Acknowledgments

The author would like to thank Luke Strohm of the U.S. Army Research Laboratory’s Weapons and Materials Research Directorate. Mr. Strohm provided technical and editorial recommendations that improved the quality of this report.
INTENTIONALLY LEFT BLANK.
1. Introduction

Two-dimensional (2-D) operations, such as rotations, translations, and intersections, are tools that are essential for many types of scientific modeling. However, the C++ programming language does not natively perform them. This report documents a set of functions, written in C++, that can be used to perform 2-D rotations, translations, and intersections. All of the functions have been grouped into the y2DOps namespace, which is summarized at the end of this report.

The functions that are presented in this report are special cases of more general three-dimensional (3-D) functions.\(^1\) Compared to the 3-D functions, the 2-D functions provide simpler interfaces and faster calculations.

2. Translation of a Point in Space

2.1 Derivation

Let the position vector \( \vec{p} \) represent an arbitrary point in a plane, where

\[
\vec{p} = p_x \hat{x} + p_y \hat{y}.
\]  

(1)

Furthermore, let \( \vec{d} \) represent a displacement vector, where

\[
\vec{d} = d_x \hat{x} + d_y \hat{y}.
\]  

(2)

If \( \vec{p}' \) is used to represent \( \vec{p} \) after it has been translated, then

\[
\vec{p}' = (p_x + d_x) \hat{x} + (p_y + d_y) \hat{y}.
\]  

(3)

2.2 C++ Implementation

Translate2D() Code

```cpp
inline void Translate2D( // <== PERFORMS A 2D TRANSLATION double p[2], // <------ COORDINATES TO TRANSLATE (MODIFIED BY THIS FUNCTION) const double d[2] ) // <------- DISPLACEMENT VECTOR p[0]+=d[0], p[1]+=d[1]; } // ~~~~ YAGENAUT@GMAIL.COM ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LAST~UPDATED~02MAY2013~~~
```

Translate2D() Parameters

**p**  
**p** is a two-element array that stores the position vector that is described by equation 1 ($p = \{p_x, p_y\}$). Note that **p** is modified by the Translate() function, as described by equation 3.

**d**  
**d** is a two-element array that stores the displacement vector that is described by equation 2 ($d = \{d_x, d_y\}$). **d** determines the amount and direction by which **p** is translated.

Translate2D() Example

Figure 1 shows point $\vec{p}$ being translated to a new position ($\vec{p}'$) by displacement vector $\vec{d}$.

Figure 1. Translate() example.

Let $\vec{p} = \{3, 1\}$ and $\vec{d} = \{-2, 1\}$. Point $\vec{p}'$ can be found by using the Translate2D() function, as shown in the following sample code.

```c
#include <cstdio>
#include "y_2d_ops.h"

int main()
{
    double p[2] = {3, 1};
    double d[2] = {-2, 1};
    y2DOps::Translate2D(p, d);
    printf("p[0]=%f, p[1]=%f\n", p[0], p[1]);
}
```

OUTPUT:

```
p[0]=1.000000, p[1]=2.000000
```
3. Rotation of a Point About an Arbitrarily Positioned Axis

3.1 Derivation

Suppose that the unit vector \( \mathbf{\hat{v}} \) is used to define an arbitrary axis about which a point in space will be rotated.

\[
\mathbf{\hat{v}} = v_x \mathbf{\hat{x}} + v_y \mathbf{\hat{y}} + v_z \mathbf{\hat{z}}.
\]  

(4)

Rodrigues’s rotation formula can be used to construct a rotation matrix, \( R \), that can be used to perform a rotation about \( \mathbf{\hat{v}} \) by an angle \( \theta \). The direction of the rotation can be determined by using the right-hand-thumb rule (when the right thumb is pointed in the direction of \( \mathbf{\hat{v}} \), the curled fingers of the right hand will point in the direction of the rotation).

\[
R = \begin{bmatrix}
  v_x^2 (1 - c_\theta) + c_\theta & v_x v_y (1 - c_\theta) - v_z s_\theta & v_x v_z (1 - c_\theta) + v_y s_\theta \\
  v_x v_y (1 - c_\theta) + v_z s_\theta & v_y^2 (1 - c_\theta) + c_\theta & v_y v_z (1 - c_\theta) - v_x s_\theta \\
  v_x v_z (1 - c_\theta) - v_y s_\theta & v_y v_z (1 - c_\theta) + v_x s_\theta & v_z^2 (1 - c_\theta) + c_\theta
\end{bmatrix},
\]

(5)

where

\[
c_\theta \equiv \cos(\theta)
\]

(6)

and

\[
s_\theta \equiv \sin(\theta).
\]

(7)

For the 2-D case, assume that \( \mathbf{\hat{v}} \) points in the positive \( \mathbf{\hat{z}} \) direction. Then

\[
v_x = 0, \quad v_y = 0, \quad \text{and} \quad v_z = 1.
\]

(8)

This greatly simplifies equation 5:

\[
R = \begin{bmatrix}
  c_\theta & -s_\theta & 0 \\
  s_\theta & c_\theta & 0 \\
  0 & 0 & 1
\end{bmatrix}.
\]

(9)

Substituting equations 6 and 7 into equation 5, then converting to 2D,

\[
R = \begin{bmatrix}
  \cos \theta & -\sin \theta \\
  \sin \theta & \cos \theta
\end{bmatrix}.
\]

(10)

---

\(^2\text{Mason, M. T. Mechanics of Robotic Manipulation; Massachusetts Institute of Technology Press: Cambridge, MA, 2001, (p 46, equation 3.26).}\)
Let the position vector $\vec{p}$ locate an arbitrary point in a plane, where

$$\vec{p} = p_x \hat{x} + p_y \hat{y}. \quad (11)$$

Let the position vector $\vec{o}$ locate the origin of the rotation axis defined by $\hat{v}$.

$$\vec{o} = o_x \hat{x} + o_y \hat{y}. \quad (12)$$

The translation-rotation-translation sequence described by equation 13 can be used to find $\vec{p}'$, where $\vec{p}'$ is used to represent $\vec{p}$ after it has been rotated about $\hat{v}$.

$$\vec{p}' = R(\vec{p} - \vec{o}) + \vec{o}. \quad (13)$$

### 3.2 C++ Implementation

Two functions are used to perform 2-D rotations. The first function, RMatrix2D(), calculates the rotation matrix that is presented in equation 10. The second function, Rotate2D(), performs the rotation that is presented in equation 13. Breaking the calculation into two functions allows functions that rotate objects containing more than one point to be written in a manner that doesn’t sacrifice performance.

#### RMatrix2D() Code

```cpp
inline void RMatrix2D(double R[4], const double rads){
    R[0]=cos(rads), R[1]=-sin(rads);
}
```

#### RMatrix2D() Parameters

- **R** is a four-element array that stores the rotation matrix that is described by equation 10 ($R = \{R_{0,0}, R_{0,1}, R_{1,0}, R_{1,1}\}$). Note that **R** is modified by the RMatrix2D() function. **R** is intended to be used as the third argument of the Rotate2D() function.

- **rads** is used to represent the angle (in radians) of the rotation. The direction of the rotation is counterclockwise (see figure 2).

#### Rotate2D() Code

```cpp
}
```

#### Rotate2D() Parameters

- **o** is the origin of the rotation axis.
- **R** is the rotation matrix that is calculated by the RMatrix2D() function.
- **p** is the point to be rotated.
**Rotate2D() Parameters**

- **p** is a two-element array that stores the position vector that is described by equation 11 ($p = \{ p_x, p_y \}$). Note that $p$ is modified by the Rotate2D() function, as described by equation 13.

- **o** is a two-element array that stores the position vector that is described by equation 12 ($o = \{ o_x, o_y \}$). $o$ is the point about which $p$ is rotated.

- **R** is a rotation matrix that has been precalculated using the RMatrix2D() function.

**Rotate2D() Example**

Figure 2 shows point $\vec{p}$ being rotated about $\vec{o}$ to a new position ($\vec{p}'$).

Let $\vec{p} = \{ 3,1 \}$ and $\vec{o} = \{ 2,1 \}$. Furthermore, let the angle of rotation be $\pi/2$. Point $\vec{p}'$ can be found by first using the RMatrix2D() function to calculate a rotation matrix, then using the Rotate2D() function to perform the rotation.

```
#include <cstdio>
#include "y_2d_ops.h"
int main(){
    double p[2]={3,1};
    double o[2]={2,1};
    double R[4];/*<-*/y2DOps::RMatrix2D(R,3.14159265358979/2);///<...rotation matrix
    y2DOps::Rotate2D(p,o,R);
    printf("p[0]=%f, p[1]=%f\n",p[0],p[1]);
}~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~LAST~UPDATED~02MAY2013~~~~~~
```

**OUTPUT:**

```
p[0]=2.000000, p[1]=2.000000
```
4. Intersection Between Two Lines

4.1 Derivation

Suppose that line $A$ passes through the points $\tilde{A}_0$ and $\tilde{A}_1$ where

$$\tilde{A}_0 = A_{0,x}\hat{x} + A_{0,y}\hat{y} \quad \text{and} \quad \tilde{A}_1 = A_{1,x}\hat{x} + A_{1,y}\hat{y}. \quad (14)$$

Let $\tilde{p}_A$ represent a point that lies on $A$. $\tilde{A}_0$ and $\tilde{A}_1$ can be used to construct a parametric equation for $\tilde{p}_A$ as a function of the parameter $t_0$:

$$\tilde{p}_A = \tilde{A}_0 + (\tilde{A}_1 - \tilde{A}_0)t_0. \quad (15)$$

The parameter $t_0$ represents the scaled distance from $\tilde{A}_0$ to $\tilde{A}_1$ along $A$. Thus, if $t_0 = 0$, $\tilde{p}_A$ is located at $\tilde{A}_0$. If $t_0 = 1$, $\tilde{p}_A$ is located at $\tilde{A}_1$.

Similarly, suppose that line $B$ passes through the points $\tilde{B}_0$ and $\tilde{B}_1$ where

$$\tilde{B}_0 = B_{0,x}\hat{x} + B_{0,y}\hat{y} \quad \text{and} \quad \tilde{B}_1 = B_{1,x}\hat{x} + B_{1,y}\hat{y}. \quad (16)$$

Let $\tilde{p}_B$ represent a point that lies on $B$. $\tilde{B}_0$ and $\tilde{B}_1$ can be used to construct a parametric equation for $\tilde{p}_B$ as a function of the parameter $t_1$:

$$\tilde{p}_B = \tilde{B}_0 + (\tilde{B}_1 - \tilde{B}_0)t_1. \quad (17)$$

Figure 3 presents an image of lines $A$ and $B$ for the case where they intersect.
The point of intersection between $A$ and $B$ occurs where $\vec{p}_A$ is equal to $\vec{p}_B$. Thus,

$$\vec{A}_0 + (\vec{A}_1 - \vec{A}_0) t_0 = \vec{B}_0 + (\vec{B}_1 - \vec{B}_0) t_1.$$  \hfill (18)

Rearranging terms,

$$\vec{A}_0 - \vec{B}_0 = (\vec{A}_0 - \vec{A}_1) t_0 + (\vec{B}_1 - \vec{B}_0) t_1.$$  \hfill (19)

This can be written in matrix form as

$$\begin{bmatrix} A_{0,x} - B_{0,x} \\ A_{0,y} - B_{0,y} \end{bmatrix} = \begin{bmatrix} A_{0,x} - A_{1,x} & B_{1,x} - B_{0,x} \\ A_{0,y} - A_{1,y} & B_{1,y} - B_{0,y} \end{bmatrix} \begin{bmatrix} t_0 \\ t_1 \end{bmatrix}. $$  \hfill (20)

Solving for $\vec{t}$,

$$\begin{bmatrix} t_0 \\ t_1 \end{bmatrix} = \begin{bmatrix} A_{0,x} - A_{1,x} & B_{1,x} - B_{0,x} \\ A_{0,y} - A_{1,y} & B_{1,y} - B_{0,y} \end{bmatrix}^{-1} \begin{bmatrix} A_{0,x} - B_{0,x} \\ A_{0,y} - B_{0,y} \end{bmatrix}. $$  \hfill (21)

Recall that the vector $\vec{t}$ contains the parameters from equations 15 and 17. Thus, once $\vec{t}$ is known, $t_0$ can be substituted into equation 15 to find the point of intersection between $A$ and $B$. Note that if the two-by-two matrix defined in equation 21 is noninvertible, then $A$ is parallel to $B$. 

Figure 3. Intersecting lines $A$ and $B$. 

Because $t_0$ and $t_1$ are defined to be scaled distances from $\tilde{A}_0$ to $\tilde{A}_1$ and $\tilde{B}_0$ to $\tilde{B}_1$, respectively, if the conditions presented in equation 22 are met, then the point of intersection lies between $\tilde{A}_0$ and $\tilde{A}_1$ and between $\tilde{B}_0$ and $\tilde{B}_1$, as shown in figure 3.

$$0 < t_1 < 1 \text{ and } 0 < t_2 < 1.$$ (22)

### 4.2 C++ Implementation

Two functions are used to find line-line intersections. The first function, `IParameters2D()`, calculates a two-element array that is the solution to equation 21. The second function, `Intersect2D()`, calculates the point of intersection between two lines. Because there is a chance that the two-by-two matrix shown in equation 21 will be singular, a Boolean that indicates whether or not a solution is valid is returned by the `IParameters2D()` function.

#### `IParameters2D()` Code

```cpp
inline bool IParameters2D(void) {
  //<-----------PARAMETERS FOR LINE-LINE INTERSECTION
  double t[2], //<--INTERSECTION PARAMETERS (CALCULATED BY THIS FUNCTION)
  const double A[4], //<-----------------------------LINE A {A0X,A0Y,A1X,A1Y}
  const double B[4], //<-------------------------------LINE B {B0X,B)Y,B1X,B1Y}
  double e=1E-9);//<----CUTOFF VALUE FOR DETERMINING IF A AND B ARE PARALLEL
double D=a*d-b*c; //determinant of 2x2 matrix
  if(fabs(D)<e) return false; //=> A & B are parallel (and t is meaningless)
double f=A[0]-B[0] , g=A[1]-B[1];
t[0]=(d*f-b*g)/D , t[1]=(-c*f+a*g)/D;
  return true; //=> A & B are not parallel
};//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~02MAY2013~~~~~
```

#### `IParameters2D()` Parameters

- **R** is a rotation matrix that has been precalculated using the `RMatrix2D()` function.
- **t** is a two-element array that stores the parameters described in equation 21 ($t=\{t_0,t_1\}$). Note that `t` is modified by the `IParameters2D()` function.
- **A** is a four-element array that stores the line that is defined by equation 14 ($A=\{A_{0x},A_{0y},A_{1x},A_{1y}\}$).
- **B** is a four-element array that stores the line that is defined by equation 16 ($B=\{B_{0x},B_{0y},B_{1x},B_{1y}\}$).
- **e** is the cutoff value for testing whether or not `A` and `B` are parallel. If the determinant of the matrix in equation 21 is less than `e`, then `A` and `B` are considered to be parallel. The default value of `e` is $10^{-9}$. 

8
IParameters2D() Return Value

IParameters2D() returns false if \( A \) is parallel to \( B \). A return value of false indicates that \( t \) has not been calculated and, thus, should not be passed to the Intersect2D() function.

Intersect2D() Code

```cpp
inline bool Intersect2D(QuadLineParams2D& params) {
    //<--------------CALCULATES LINE-LINE INTERSECTION POINT
    double x[2],//<--------POINT OF INTERSECTION (CALCULATED BY THIS FUNCTION)
    const double t[2],//<-------INTERSECTION PARAMETERS (FROM IParameters2D())
    const double A[4]);//<-----------------LINE A {A0X,A0Y,A1X,A1Y}

    return t[0]>0&&t[0]<1&&t[1]>0&&t[1]<1;
}
```

Intersect2D() Parameters

- **x**
  - \( x \) is a two-element array that stores the point of intersection between lines \( A \) and \( B \).
  - Note that \( x \) is modified by the Intersect2D() function.

- **t**
  - \( t \) is a parameter list that has been precalculated using the IParameters2D() function.

- **A**
  - \( A \) is a two-element by two-element array that stores the line that is defined by equation 14 \( (A=\{A_{0,x},A_{0,y},A_{1,x},A_{1,y}\}) \).

Intersect2D() Return Value

Intersect2D() returns true if line segment \( A \) intersects line segment \( B \).

Intersect2D() Example

Figure 4 shows intersecting line-segments \( A \) and \( B \).

![Figure 4. Intersect2D() example.](image-url)
Let $\vec{A}_0 = \{2,0\}$, $\vec{A}_1 = \{2,2\}$, $\vec{B}_0 = \{1,1\}$, and $\vec{B}_1 = \{3,1\}$. The point of intersection can be found by first calling the IParameters2D() function, then using the result in the Intersect2D() function.

```c
#include <cstdio>
#include "y_2d_ops.h"

int main(){
    double A[4]={2,0 , 2,2}; // a line segment
    double B[4]={1,1 , 3,1}; // a line segment
    double t[2];
    double p[2];
    y2DOps::IParameters2D(t,A,B);
    y2DOps::Intersect2D(p,t,A);
    printf("p[0]=%f, p[1]=%f\n",p[0],p[1]);
}
```

OUTPUT:

```
p[0]=2.000000, p[1]=1.000000
```

### 5. Summary

A summary sheet is provided at the end of this report. It presents the y2DOps namespace, which contains the five functions that are described in detail in sections 2, 3, and 4. Also presented are two examples that demonstrate the versatility of the functions described in this report. The first uses the Rotate2D() function to calculate a set of points that defines a simple orbit of a moon around a planet, which in turn is in orbit around a star. The second uses the Intersect2D() function to draw a four-sided spiral. Both functions create text files that contain all of the information needed to create the two images presented in the summary sheet.
EXAMPLE 1

#include <stdio.h> // Redirect output to a file
#include "y_2d_ops.h" // Redirect output to a file

int main(){
    FILE *f=fopen("orbit.txt","w");
    printf("%6.3f,%6.3f,%6.3f,%6.3f\n",x,y,x,y);
    fclose(f);
}

orbit.txt

# A planet, # a moon
# x, y, x, y
1.000, 0.000, 1.000, 0.000
1.000, 0.004, 1.192, 0.060
1.000, 0.009, 1.169, 0.116
...

EXAMPLE 2

#include <stdio.h> // Redirect output to a file
#include "y_2d_ops.h" // Redirect output to a file

int main(){
    FILE *f=fopen("spiral.txt","w");
    printf("%6.3f,%6.3f\n",x,y);
    fclose(f);
}

spiral.txt

# x, y
-1.000, 1.000
-1.000, -1.000
1.000, 1.000
1.000, 0.400
1.000, -0.400
-0.972, -1.000
-1.000, -0.544
0.917, 1.000
...

Y2D_Ops Summary

FIGURE 1
image created from orbit.txt.

FIGURE 2
image created from spiral.txt.