Naval Research Laboratory
Washington, DC 20375-5320

NRL/5320/MR—2022/2

GeoLib, a C++ Library for Phased Array Radar Coordinate Systems

J. B. EVINS

Advanced Radar Systems Branch
Radar Division

June 1, 2022

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
01-06-2022	NRL Memorandum Report	10-01-2015 – 09-30-2021

4. TITLE AND SUBTITLE

GeoLib, a C++ Library for Phased Array Radar Coordinate Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

J. B. Evins

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
1E88

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/5320/MR--2022/2

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research, Code 312
875 N. Randolph Street
Arlington, VA 22217-1995

10. SPONSOR / MONITOR'S ACRONYM(S)

ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

GeoLib is a simple but powerful C++ library for working with geographic and phased array antenna geometries. It includes a Unit-Independent Types subsystem and a Coordinate Types subsystem. This report documents the capabilities of these two subsystems.

15. SUBJECT TERMS

Geodetic coordinates Phased array antenna coordinates
C++ utilities

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			James B. Evins
U	U	U	U	25	**19b. TELEPHONE NUMBER** *(include area code)* (202) 404-1942

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

CONTENTS

This page intentionally left blank.

GeoLib, a C++ Library for Phased Array Radar Coordinate Systems

INTRODUCTION

The Flexible Distributed Array Radar (FlexDAR) is a testbed to demonstrate every-element digital-beamforming (EEDBF) to support distributed radar operations. Two radar nodes for this testbed have been developed. Each node is a complete single-face phased array radar composed of a FlexDAR Front-End (FFE) EEDBF array subsystem developed by Raytheon, and a FlexDAR Back-End (FBE) processing subsystem developed by NRL.

GeoLib is one of several portable libraries developed as part of the FBE software development effort. Figure 1 illustrates the dependency relationships of this library to some of the other FBE software components. GeoLib was designed to be independent of the other components in the FlexDAR software stack, thus easily separable for use in other software projects. It has no external requirements other than a C++11 compiler and standard library [1]. The GeoLib library is designed to be built using the CMake [2] build system (version 3.12 or later).

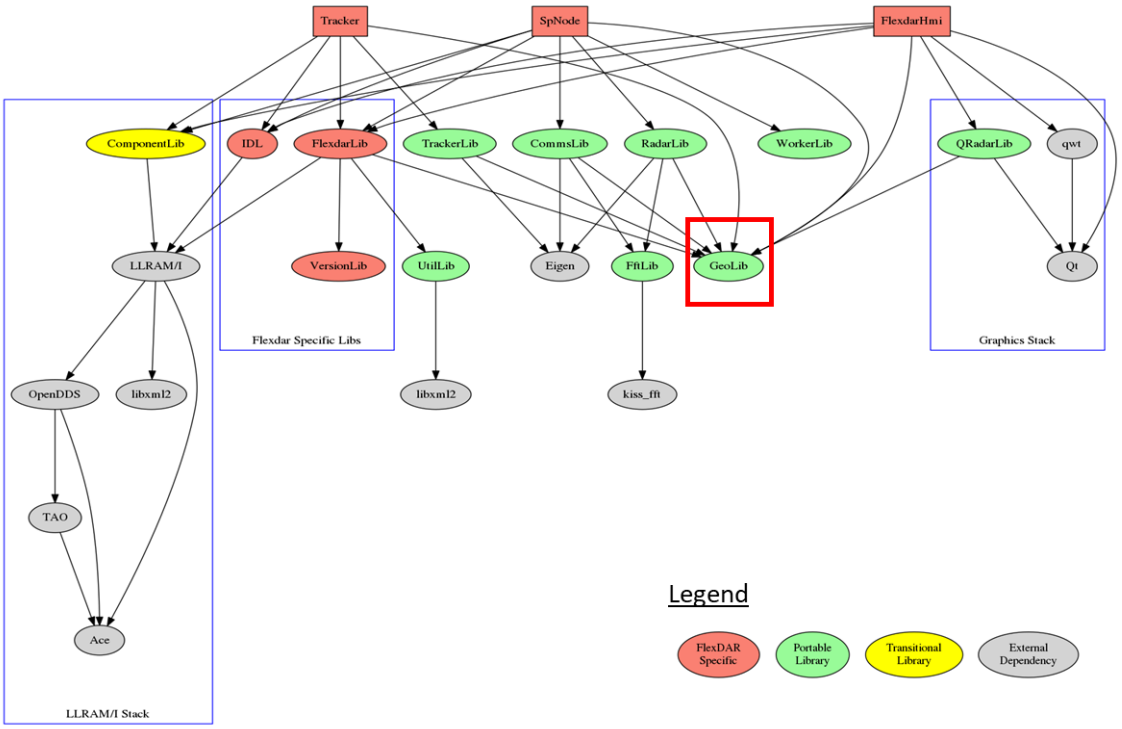

Figure 1 – Dependency relationship of various FlexDAR software modules.

GeoLib is a simple but powerful C++ library for working with geographic and phased array antenna geometries. It includes a Unit-Independent Types subsystem and a Coordinate Types subsystem. This report documents the capabilities of these two subsystems.

Manuscript approved May 31, 2022.

UNIT-INDEPENDENT TYPES SUBSYSTEM

A common programming mistake is to incorrectly assume the particular units of a variable or other value. For example, a variable may contain an angle in degrees, but a programmer may assume its value is in radians. Such mistakes are often difficult to locate and can even cause disastrous results. [3]

One approach to preventing these mistakes is to employ strict naming conventions. For example, the FlexDAR coding standard [4] states

> *If a variable represents time, weight, or some other unit, include the unit in the name so developers can more easily spot problems.*

Such a naming convention is used in all message definitions within the FlexDAR system. However, the FBE software internally uses GeoLib's Unit-Independent Types subsystem.

The Unit-Independent Types subsystem provides a set of classes that represent different physical quantities, such as distance, time or angle. Each class is a simple wrapper around a single member of type `double` which represents a quantity in some native units. Access to the internal value is through unit-specific initializer and accessor methods.

Using the Unit-Independent Types Subsystem

To use the Unit-Independent Types subsystem, include its header file as shown in Listing 1. All symbols provided by the GeoLib library will be in the `geo` namespace.

```
#include "GeoLib/UnitIndependentTypes.h"
```

Listing 1 – Including the Unit-Independent Types subsystem header file.

Initializing Unit-Independent Variables

Each class provides a set of static methods for initializing unit-independent variables from values expressed in a variety of supported units. These methods will convert the supplied values into the type's native units for internal storage. For example, the native unit for the geo::Distance class is meters, so all values are converted to meters for storage as seen in Listing 2.

```
auto d1 = geo::Distance::m( 0.25 );   // Internally, d1 = 0.25
auto d2 = geo::Distance::nmi( 100 );  // Internally, d2 = 185,200
auto d3 = geo::Distance::km( 1.852 ); // Internally, d3 = 1,852
```

Listing 2 – Example initialization of unit-independent variables. Conversion to internal native units is automatic.

Accessing the Values of Unit-Independent Variables

For each static initializer method, there is a corresponding accessor method with the same name. These accessor methods will convert the internal value from the class's native units into the desired units for use in computations or for output as shown in Listing 3.

```
// Print values from previous example in various units

std::cout << d1.ft() << " ft";   // Prints "0.82021 ft"
std::cout << d2.m() << " m";     // Prints "185200 m"
std::cout << d3.nmi() << " nmi"; // Prints "1 nmi"
```

Listing 3 – Examples of accessing unit-independent variables in various units. Conversion to the desired units is automatic.

Supported Unit-Independent Types and Units

Table 1 is a list of the current unit-independent types, their supported units, and their corresponding initializer and accessor methods.

Table 1 – Unit-independent types supported by GeoLib

Type/class	Units Supported	C++ Initializer & Accessor Methods
Distance	meters	m
	feet	ft
	kilometers	km
	nautical miles	nmi
	radar data miles	dm
Area	square meters	m2
	square feet	ft2
Volume	cubic meters	m3
	cubic feet	ft3
Frequency	hertz	hz
	Kilohertz	khz
	Megahertz	mhz
	Gigahertz	ghz
Time	seconds	sec
	milliseconds	mSec
	microseconds	uSec
	nanoseconds	nSec
Angle	radians	rad
	degrees	deg
Velocity	meters per second	mps
	kilometers per hour	kph
	knots	kn
	feet per minute	fpm
Acceleration	meters per second^2	mps2
	feet per second^2	fps2
	standard gravity units	g
AngularVelocity	radians per second	rps
	degrees per second	dps

Choosing Units at Runtime

Each class also provides an enumeration of available units, so that units can be selected at runtime. This is done using the special units() initializer and accessor methods, as illustrated in Listing 4.

```
auto myUnits = geo::Distance::Units::NMI;  // One of M, FT, KM, NMI, or DM
                                           // Note: unit enumerators are all caps,
                                           // while unit initializers and accessors
                                           // are all lower case.

auto d1 = geo::Distance::units( 5.0, myUnits );
std::cout << d1.units( myUnits ) << " " << geo::Distance::unitsToString( myUnits );
```

Listing 4 – Example initialization and access of unit-independent variables in units selected at run-time.

This can be useful in a GUI application where input/output units are user-selectable.

Operators and Dimensional Analysis Semantics

Given just these methods, we can easily avoid using the wrong units simply by being explicit about which units we want to use at each step of our calculations. The methods will perform any unit conversions needed as shown in the example in Listing 5.

```
auto t1 = geo::Time::uSec(1000);
auto t2 = geo::Time::mSec(1.02);
auto deltaT = geo::Time::sec( t2.sec() - t1.sec() );
auto v = geo::Velocity::mps( geo::Distance::m(0.023) / deltaT.sec() );
```

Listing 5 – Explicitly selecting units during calculations.

However, this can be very verbose and cumbersome. It would be much more desirable to not even worry about units except when initializing variables or outputting results. To accomplish this goal, GeoLib uses C++ overloaded operators to implement the semantics of dimensional analysis.

Each type provides a basic set of operators between like values and scalar values. For example, Figure 2 shows the operators available for the Distance unit-independent type.

Distance = Distance + Distance
Distance = Distance - Distance
Distance = -Distance
*Distance = scalar * Distance*
*Distance = Distance * scalar*
Distance = Distance / scalar
scalar = Distance / Distance

Distance += Distance
Distance -= Distance
*Distance *= scalar*
Distance /= scalar

Figure 2 – Supported operators between Distance values and between Distance and scalar values.

GeoLib also provides many operators between different unit-independent types to provide dimensional analysis semantics. For example, Figure 3 shows additional operators involving the Distance type and other unit-independent types.

*Area = Distance * Distance*
*Volume = Distance * Area*
*Volume = Area * Distance*
*Velocity = Frequency * Distance*
*Velocity = Distance * Frequency*
*Distance = Velocity * Time*
*Distance = Time * Velocity*
Distance = Area / Distance
Velocity = Distance / Time
Time = Distance / Velocity
Distance = Velocity / Frequency
Distance = Volume / Area
Area = Volume / Distance

Figure 3 – Operators involving Distance values and other unit-independent types.

For example, if you divide a *Distance* by a *Time* you will get a *Velocity* as in Listing 6.

```
auto t1 = geo::Time::uSec(1000);
auto t2 = geo::Time::sec(1.02);

auto d1 = geo::Distance::m(123.45);
auto d2 = geo::Distance::km(10);

auto v = ( d2 - d1 ) / ( t2 - t1 );  // Perform calculation w/o concern for units

std::cout << "Velocity = " << v.kn() << " kn";
```

Listing 6 – Example of dimensional analysis semantics. In this example, we divide the difference of two distances by the difference of two times to get a velocity. This calculation is independent of the units with which we loaded our variables or the units that we ultimately output.

GeoLib also provides several composite unit-independent types as listed inTable 2. The purpose of these composite types is to support intermediate results in complex expressions without needing to drop out of unit-independent types. Since the primary use case for these composite types is to support intermediate results, they do not provide unit accessors or initializers.

Table 2 – Currently supported GeoLib composite unit-independent types for intermediate results.

Type/class	Description	Internal Units
Acceleration2	Acceleration * Acceleration	$(m/s^2)^2$
Angle2	Angle * Angle	$radians^2$
Distance4	Area * Area	m^4
Time2	Time * Time	$seconds^2$
Velocity2	Velocity * Velocity	$(m/s)^2$

Table 3 and

Table 4 show which multiplication and division operators are currently implemented. These tables are currently very sparse. So far, the missing operators have not been needed in FlexDAR. In order to fill in the missing operators, additional composite types would need to be created for the intermediate results. These additional types would then expand the table, revealing even more missing operators. Thus, these tables would grow without bounds. Even if we automatically generated code for the new operators and composite types, we would need to choose some point to terminate the recursion. The current plan is to add operators and intermediate types as we encounter them in the "real world."

Table 3—Supported unit-independent product operators. Product = Multiplicand × Multiplier. To keep the table compact, the numbers in table cells indicate the Product's type. (E.g. "11. Time" × "2. Acceleration" = "13. Velocity"). An empty cell indicates that the corresponding Product is not currently implemented.

Multiplicand \ Multiplier	*1. scalar*	*2. Acceleration*	*3. Acceleration2*	*4. Angle*	*5. Angle2*	*6. AngularVelocity*	*7. Area*	*8. Distance*	*9. Distance4*	*10. Frequency*	*11. Time*	*12. Time2*	*13. Velocity*	*14. Velocity2*	*15. Volume*
1. Scalar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2. Acceleration	2	3									13				
3. Acceleration2	3														
4. Angle	4			5						6					
5. Angle2	5														
6. AngularVelocity	6										4				
7. Area	7						9	15							
8. Distance	8						15	7		13					9
9. Distance4	9														
10. Frequency	10			6				13			1				
11. Time	11	13				4				1			8		
12. Time2	12														
13. Velocity	13										8		14		
14. Velocity2	14														
15. Volume	15							9							

Table 4—Supported unit-independent quotient operators. Quotient = Dividend / Divisor. To keep the table compact, the numbers in table cells indicate the Quotient's type. (E.g. "8. Distance" / "11. Time" = "13. Velocity"). An empty cell indicates that the corresponding Quotient is not currently implemented.

Dividend \ Divisor	*1. scalar*	*2. Acceleration*	*3. Acceleration2*	*4. Angle*	*5. Angle2*	*6. AngularVelocity*	*7. Area*	*8. Distance*	*9. Distance4*	*10. Frequency*	*11. Time*	*12. Time2*	*13. Velocity*	*14. Velocity2*	*15. Volume*
1. Scalar	1									11	10				
2. Acceleration	2	1													
3. Acceleration2	3	2	1												
4. Angle	4			1							6				
5. Angle2	5				1										
6. AngularVelocity	6					1									
7. Area	7						1	8							
8. Distance	8							1			13	2	11		
9. Distance4	9						7	15	1						
10. Frequency	10									1					
11. Time	11										1				
12. Time2	12											1			
13. Velocity	13							11		8	2		1		
14. Velocity2	14							2					13	1	
15. Volume	15						8	7							1

GeoLib also overloads various math functions as seen Figure 4 so that values do not need to be extracted in specific units to perform these functions. These include trigonometric functions for Angle types and square root for various types such as Area and intermediate and composite types.

scalar = sin(Angle)
scalar = cos(Angle)
scalar = tan(Angle)
Angle = asin(scalar)
Angle = acos(scalar)
Angle = atan(scalar)
Angle = atan2(scalar, scalar)
Angle = atan2(Distance, Distance)
Angle = atan2(Velocity, Velocity)
Angle = atan2(Velocity2, Velocity2)
Angle = atan2(Acceleration, Acceleration)
Angle = atan2(Acceleration2, Acceleration2)

Distance = sqrt(Area)
Velocity = sqrt(Velocity2)
Acceleration = sqrt(Acceleration2)
Angle = sqrt(Angle2)
Area = sqrt(Distance4)

Figure 4 – Overloaded math functions involving unit-independent types.

Utility Methods

Each unit-independent type also includes various utility methods. For example, Listing 7 illustrates various methods to format unit-independent values as strings.

```
auto d = geo::Distance::nmi(1);

std::cout << d.stringNmi() << std::endl;  // Prints "1 nmi"
std::cout << d.stringM() << std::endl;    // Prints "1852 m"
std::cout << d.stringFt() << std::endl;   // Prints "6076 ft"

// Define units at runtime
auto myUnits = geo::Distance::Units::FT;
std::cout << d.string( myUnits ) << std::endl; // Also prints "6076 ft"

// These methods also take an optional width and precision argument
std::cout << d.stringFt( 9, 3 ) << std::endl;  // Prints " 6076.115 ft"
```

Listing 7 – Examples of utility methods to convert unit-independent types into std::string.

Constants

GeoLib also provides several unit-independent constants that can be used in place of unit-independent variables as shown in Listing 8.

```
x = geo::Angle::Pi;    // Pi radians or 180 degrees
x = geo::Angle::TwoPi; // 2Pi radians or 360 degrees
x = geo::Velocity::C;  // The speed of light in a vacuum
```

Listing 8 – Examples of unit-independent constants provided by GeoLib.

Implementation of Unit-Independent Types and Overhead

These unit-independent types have been implemented as simple classes, each wrapping a single `double` value. The core capabilities of these classes are provided by a common template base class.

These classes are simple thin wrappers around the C++ `double` type. Operators and unit-specific initializers and accessors are all implemented as "inlines" so there should be little to no overhead to their use. Being inlined should also provide the compiler opportunities to remove unnecessary function calls, optimize redundant unit conversions and combine unit conversion factors when possible. Inlining is not guaranteed, because the compiler will use its own heuristics to evaluate exactly what optimizations are the most beneficial.

The ultimate goal is to achieve a “zero-cost abstraction.” That is where there is no performance cost to using unit-independent types instead of manually keeping track of physical units. Figure 5 and Figure 6 use Compiler Explorer [5] to demonstrate two cases when there is no performance cost to using unit-independent types versus the manual approach.

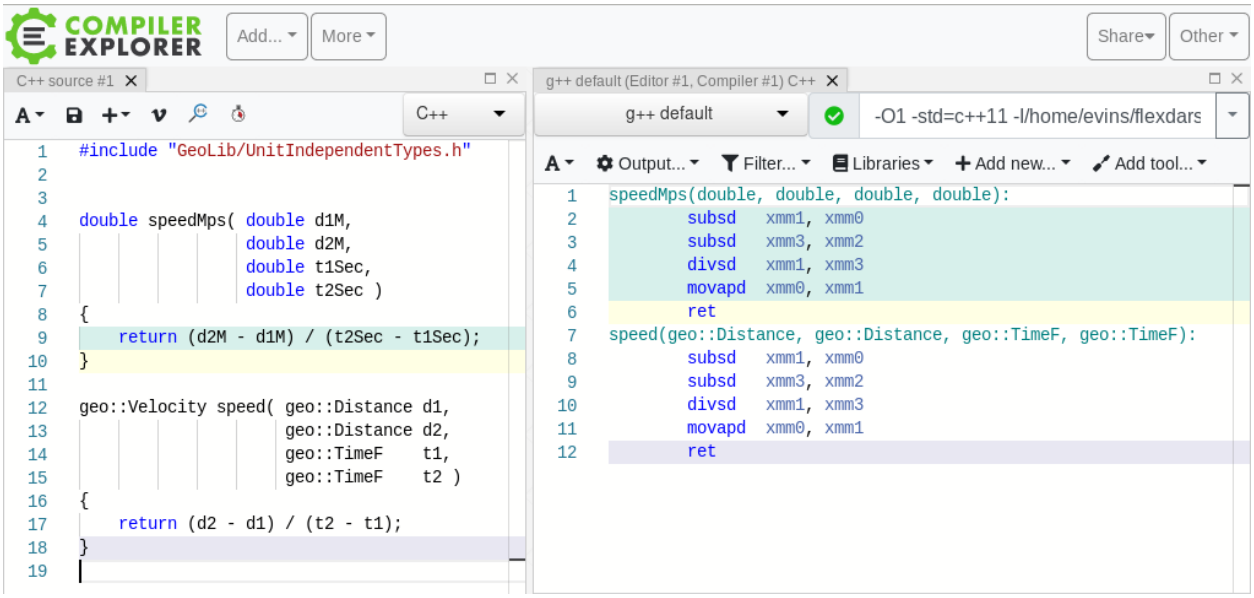

Figure 5—Simple Compiler Explorer demonstration of "zero cost abstraction” using unit-independent types. In this simple case, the resulting assembly language for the two approaches is identical.

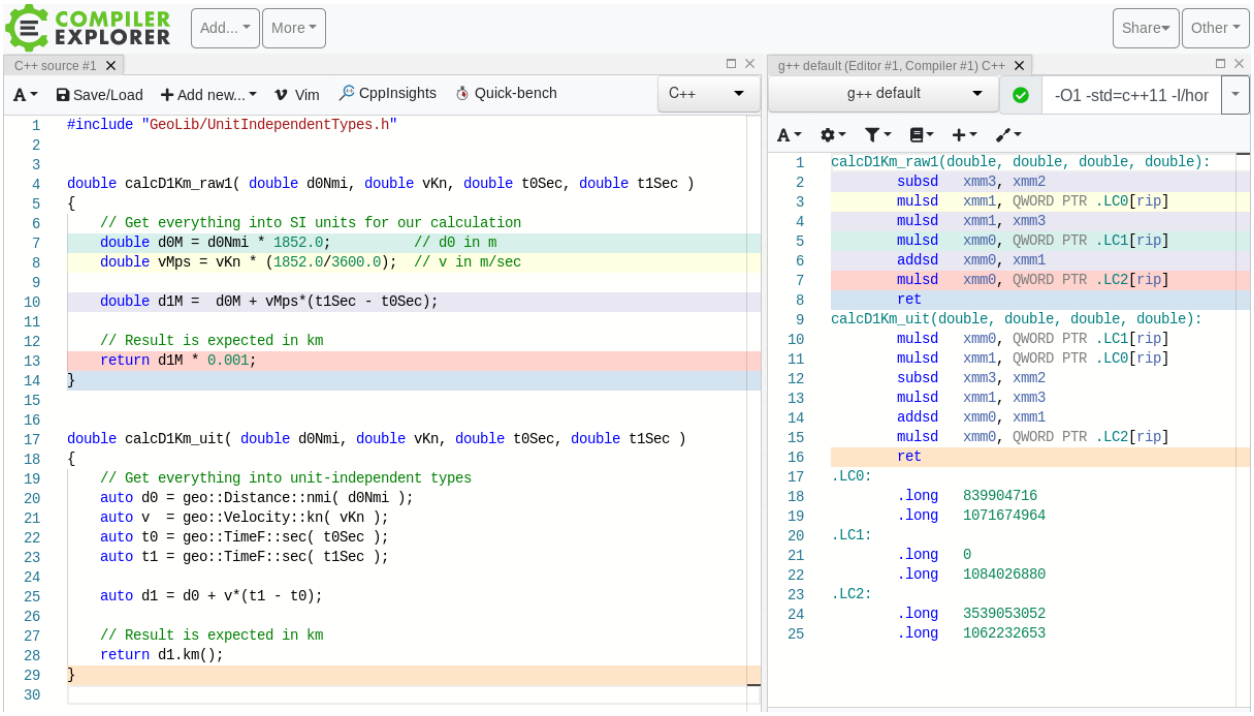

Figure 6 – A second demonstration of "zero cost abstraction." In this more complex case, the resulting assembly language for the two approaches is nearly identical (there is a slight reordering of instructions).

COORDINATE TYPES SUBSYSTEM

GeoLib provides multiple two- and three-dimensional coordinate systems, which are built from multiple unit-independent types.

```
#include "GeoLib/CoordinateTypes.h"
```

Listing 9 – Including Coordinate Types subsystem header file.

Primary Geographic and Antenna Coordinate Systems

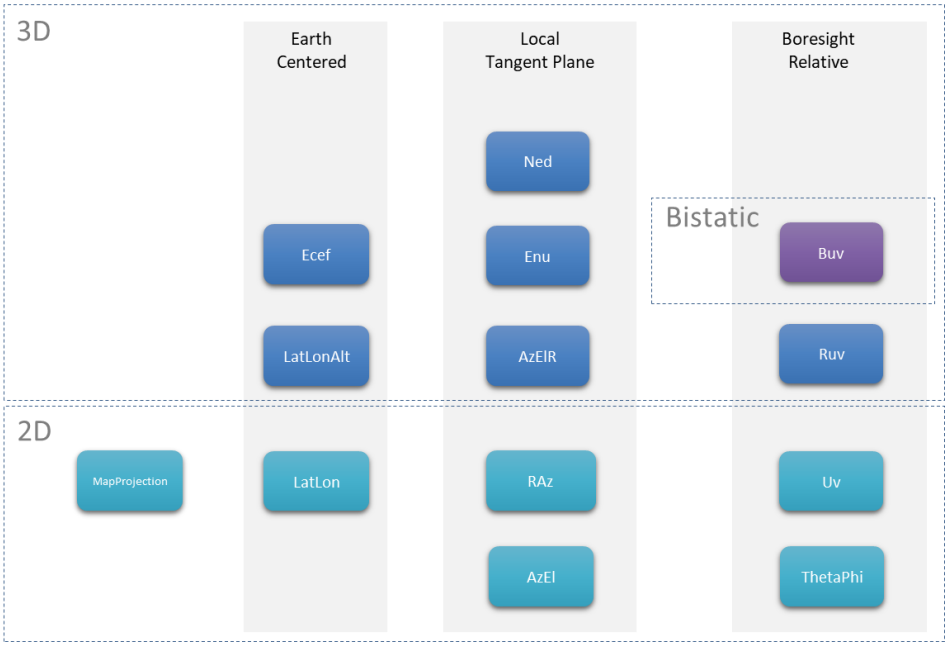

Figure 7 – Venn diagram classifying coordinate systems provided by GeoLib. Coordinate systems can be classified as Earth Centered, Local Tangent Plane, and Boresight Relative. They can also be classified as either 2-dimensional or 3-dimensional. Furthermore they can be classified as Bistatic or not.

Primary Coordinate System Classes

MapProjection

In GeoLib, Map-projection coordinates are another simple two-dimensional coordinate system. It simply represents an X distance (Easting) and a Y distance (Northing) on the plane of a map. The distances will have a non-linear relationship to actual geographic distances, depending on the projection type used. GeoLib currently only supports Gnomonic [6] projections.

Ecef

The Earth-Centered, Earth-Fixed (ECEF) [7] coordinate system is a Cartesian coordinate system centered at the earth's center of mass. Its x-axis extends through the prime meridian at the equator and its z-axis extends through the geographic North Pole.

LatLonAlt

This class is a WGS 84 [7] geodetic coordinate system represented as Latitude, Longitude, and Altitude above or below the WGS 84 reference ellipsoid. This coordinate system is used to represent locations above or below the earth's surface. When constructing a LatLonAlt object from ECEF coordinates, GeoLib uses the algorithm described by D. K. Olson [8] to calculate the geodetic latitude.

LatLon

This class is a WGS 84 geodetic coordinate system represented as Latitude and Longitude. This coordinate system is used to represent locations on the earth's surface. It is similar to LatLonAlt, except it does not include an altitude component.

Enu

This class is a Cartesian coordinate system which is relative to a position on or near the earth's surface. It is represented by three axes: east, north and up. The east-north plane is parallel to a plane which is tangential to the earth's surface. The up axis is normal to WGS 84 reference ellipsoid.

Ned

This class is similar to the Enu class, except it is represented by north, east and down components.

AzElR

This class is a relative coordinate system represented as azimuth, elevation and range. Azimuth is a clockwise angle between the point of interest and true north on a horizontal plane. Elevation is the angle of the point of interest above the horizontal plane. Range is the distance to the point of interest.

RAz

This class is a 2-dimensional relative coordinate system represented as range and azimuth. The component are defined exactly as in the AzElR class.

AzEl

This class is a 2-dimensional relative coordinate system represented by an azimuth and elevation angle. The components are defined exactly as in the AzElR class.

Uv

UV coordinates are the projection of a unit vector onto the plane of an antenna. U is along the horizontal axis and V is along the vertical axis. A third component, W, along the antenna normal, completes the unit vector. W is often ignored, but is sometimes used to determine if a vector points forwards or backwards. UV coordinates are used to represent the direction to a point of interest relative to the plane of an antenna.

In GeoLib, UVW is a left-handed coordinate system, where U, V, and W point right, up, and forward, respectively. Care must be taken if GeoLib is used in a situation where the convention is for UVW to be right-handed, which is quite common. Theta-Phi is also based on the same left-handed coordinate system.

ThetaPhi

Like Uv, this class also represents the direction to a point of interest from the plane of an antenna. Theta is the angle between the antenna normal and the point of interest. Phi is the angle between the

antenna's vertical axis (V) and the plane defined by the antenna normal and the direction to the point of interest.

Ruv

Ruv is a subclass of the Uv class. It adds a range component to fully describe the relative position of a point of interest.

Buv

Buv is similar to the Ruv class, except it adds a bistatic-range (B) component. The bistatic range is equivalent to the monostatic range plus the distance to some other point in space (usually a radar transmitter). In the monostatic case, the bistatic range is twice the monostatic range.

Interchangeability of Coordinate System Types

Using C++'s implicit type conversion feature, many of these coordinate system types can be used interchangeably. This is accomplished either through conversion constructors or through direct inheritance when one type is a subclass of another. The following diagram illustrates which coordinate system types can be used interchangeably.

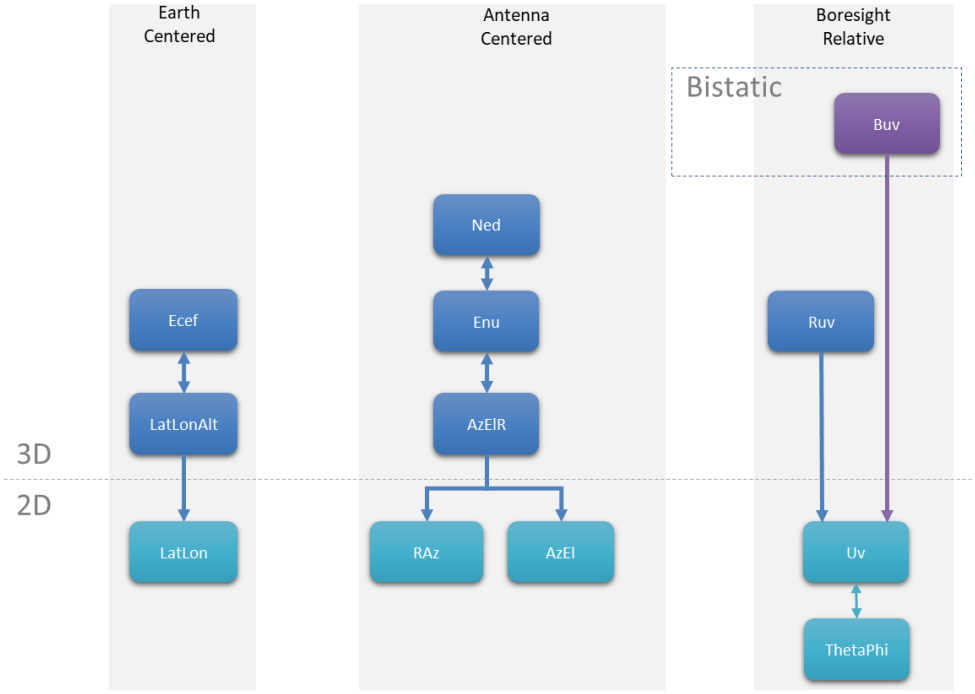

Figure 8 – Interchangeability of different coordinate systems. The arrows indicate the direction of interchangeability. Generally 3D coordinate systems can be collapsed into one or more 2D coordinate systems, but not the other way around.

A double-ended arrow indicates that the relationship is bidirectional. A single-ended arrow indicates a one-directional relationship. For example, you can use a LatLonAlt object for a LatLon object, but not the other way around, because there would be missing information. These relationships are also transitive. For example, you can use a Ned object for an AzElR object, because you can use a Ned object for an Enu object, which can be used for an AzElR object.

Coordinate System Projectors

GeoLib provides a set of "Projector" classes. These are used to project coordinates from one coordinate system to another. These projectors, along with the property of interchangeable coordinate systems allows coordinates to be translated between many of the supported coordinate systems.

In the following diagram, "Projectors" have been added to the previous diagram as large double-ended arrows.

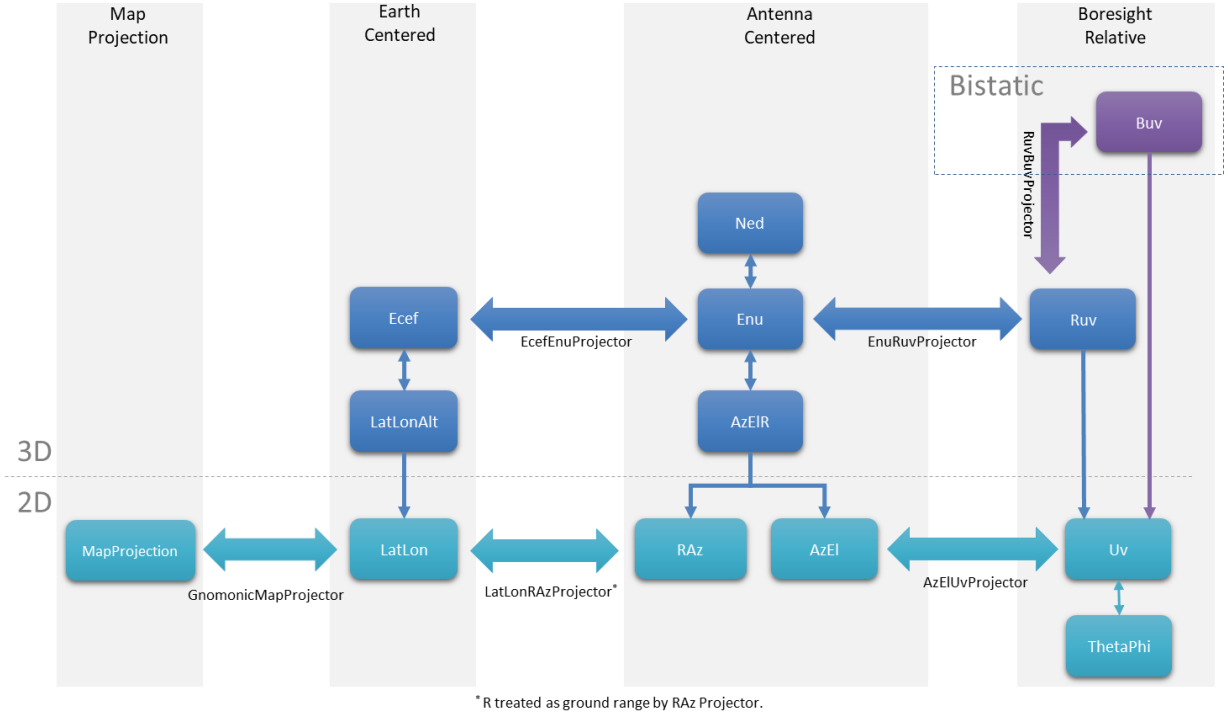

Figure 9 – Coordinate system projectors. Given an appropriate set of anchors (origins and/or orientations), a projector can "project" from one coordinate system into another.

A projector must be initialized with origin information so that it knows how one coordinate system is tied to the other.

```
using namespace geo;

// Initialize an EcefEnuProjector for my current location
LatLonAlt myLocation( Angle::degs(39.4523), Angle::degs(-77.5634), Distance::m(30) );
EcefEnuProjector myProj( myLocation );

// Location of target in LatLonAlt, which can be used in place of Ecef
LatLonAlt t( Angle::degs(39.5628), Angle::degs(-77.5467), Distance::m(9997) );

// Use projector to "project" target into my ENU coordinate system
auto tEnu = myProj.toEnu( t );
```

Listing 10 – Example of using an EcefEnuProjector to "project" a LatLonAlt point (t) into an Enu coordinate system defined by my current location. Although the toEnu() method expects Ecef coordinate, we can see in Figure 8 we can see that LatLonAlt is interchangeable with Ecef.

The implementation of the EcefEnuProjector is described in described in Appendix A – EcefEnuProjector Math. The EnuRuvProjector is described in Appendix B – EnuRuvProjector Math and the RuvBuvProjector is described in Appendix C – RuvBuvProjector Math.

Additional Capabilities

Auxiliary Coordinate System Classes

Xy

XY is a simple two dimensional Cartesian coordinate system. In FlexDAR, this coordinate system is used primarily to locate elements of the phased-array antenna as if looking through the array from its rear.

EcefVelocity, EcefAcceleration, EnuVelocity and EnuAcceleration

In addition to the positional coordinate systems, GeoLib also provides velocity and acceleration classes for both ECEF and ENU coordinate systems. While these coordinate systems represent position-free vectors, they use the same basis vectors as their corresponding positional coordinate system (ECEF or ENU). The EcefEnuProjector class provides methods to convert between velocities and accelerations in these coordinate systems as in the following example:

```
using namespace geo;

// Ecef state vector
class EcefState
{
	Ecef             p;
	EcefVelocity     v;
	EcefAcceleration a;
};

// Enu state vector
class EnuState
{
	Enu              p;
	EnuVelocity      v;
	EnuAcceleration  a;
};

EcefState sEcef = { Ecef( x, y, z ),
                    EcefVelocity( xDot, yDot, zDot ),
                    EcefAcceleration( xDotDot, yDotDot, zDotDot ) };
EnuState sEnu;

sEnu.p = myProj.toEnu( sEcef.p );
sEnu.v = myProj.toEnu( sEcef.v );
sEnu.a = myProj.toEnu( sEcef.a );
```

Listing 11 – Examples of "projecting" a position, velocity and acceleration from ECEF coordinates into ENU coordinates.

Operators on Coordinate System Classes

GeoLib also provides several operators on its Cartesian (ECEF and ENU) coordinate system types to provide dimensional analysis semantics while doing simple vector arithmetic. For example, Figure 10 lists the operators involving the ECEF coordinate system types.

*Ecef = Scalar * Ecef*
*Ecef = Ecef * Scalar*
Ecef = Ecef / Scalar
Ecef = Ecef + Ecef
Ecef = Ecef – Ecef

*EcefVelocity = Scalar * EcefVelocity*
*EcefVelocity = EcefVelocity * Scalar*
EcefVelocity = EcefVelocity / Scalar
EcefVelocity = EcefVelocity + EcefVelocity
EcefVelocity = EcefVelocity – EcefVelocity

*EcefAcceleration = Scalar * EcefAcceleration*
*EcefAcceleration = EcefAcceleration * Scalar*
EcefAcceleration = EcefAcceleration / Scalar
EcefAcceleration = EcefAcceleration + EcefAcceleration
EcefAcceleration = EcefAcceleration - EcefAcceleration

EcefVelocity = Ecef / Time
*Ecef = EcefVelocity * Time*
*Ecef = Time * EcefVelocity*

EcefAcceleration = EcefVelocity / Time
*EcefVelocity = EcefAcceleration * Time*
*EcefVelocity = Time * EcefAcceleration*

Figure 10 – Operators on ECEF coordinate system types.

FUTURE WORK

The current Unit-Independent Types subsystem meets the current FlexDAR needs. However, it can easily be extended if needed:

- Add additional units support to existing classes. E.g. we currently don't support statute miles;
- Add additional unit-independent types. E.g. Temperature, Mass, etc.;
- Add missing operators and intermediate types to extend dimensional analysis semantics to support more complex formulas. Ideally, one could perform something as complex as the radar range equation using entirely unit-independent types;
- Add additional coordinate systems, such as earth-centered-inertial (ECI) [9] to aid in tracking satellites.

In FlexDAR, we are currently only considering land-based fixed sites. To support moving sea-based platforms, an additional set of coordinate systems would be needed. These coordinate systems would

locate and orient one or more antennas relative to the platform. There would need to be an intermediate projection step between Enu and Ruv coordinate systems.

The UV coordinate system in GeoLib is left-handed (the positive direction of U is to the right) when treated as UVW, which is a local preference. This convention is far from universal, and may actually be the minority convention. A possible enhancement would be to support both left-handed and right-handed conventions. Potentially multiple conventions in various parts of the library could also be supported, such as the roll-pitch-yaw order of rotation in the EnuRuvProjector class.

The FlexDAR backend software is currently standardized on the ISO C++11 standard. If in the future, we updated our toolchain we could choose to update this requirement to C++14, C++17, or even the forthcoming C++20 standard. The C++ standard has been undergoing a major transformation over the past decade. One capability that these newer standards would enable is fewer restrictions on the `constexpr` attribute. Use of this attribute in our constructors, methods and operators could allow for additional optimization of code by allowing the compiler to pre-evaluate any code that depends solely on constants at build time, thus coming closer to a "zero-cost abstraction."

The author has recently become aware of a proposal to add Physical Units support to the C++ Standard Library [10]. If this capability becomes part of the C++ standard, the Unit-Independent Types subsystem could become obsolete and potentially be replaced by the standard implementation.

CONCLUSION

NRL developed the GeoLib library as part of the NRL FlexDAR back-end software. It is a simple but powerful C++ library for working with geographic and phased array antenna geometries. This library includes a Unit-Independent Types subsystem to simplify the task of working with various physical units. It also includes a Coordinate Types subsystem which provides various coordinate system types in support of all aspects of radar processing and projector classes for converting between them.

GeoLib was designed to be independent of the other components in the FlexDAR software stack, thus it is easily separable for use in other software projects. It has no external requirements other than a C++11 compiler and standard library.

REFERENCES

[1] cppreference.com, "C++ Compiler Support," [Online]. Available: https://en.cppreference.com/w/cpp/compiler_support. [Accessed 9 August 2021].

[2] Kitware, "CMake," [Online]. Available: https://cmake.org. [Accessed 14 July 2021].

[3] Mars Climate Orbiter Mishap Investigation Board, "Phase I Report," NASA, November 10, 1999.

[4] J. B. Evins, "NRL Radar Division C++ Coding Standard," NRL Memorandum Report, NRL/MR/5329--16-9656, Naval Research Laboratory, Washington, DC, December 6, 2016.

[5] M. Godbolt, "Compiler Explorer," [Online]. Available: https://github.com/compiler-explorer/compiler-explorer. [Accessed 22 06 2021].

[6] J. P. Snyder, "Gnomonic Projection," in *Map Projections - A Working Manual*, Washington, DC, U.S. Government Printing Office, 1987, pp. 164-168.

[7] NGA Standardization Document NGA.STND.0036_1.0.0_WGS84, *Department of Defense World Geodetic System 1984, Its Definition and Relationships with Local Geodetic Systems,* 2014.

[8] D. K. Olson, "Converting Earth-Centered, Earth-Fixed Coordinates to Geodetic Coordinates," *IEEE Transactions on Aerospace and Electronic Systems,* vol. 32, no. 1, pp. 473-476, 1996.

[9] N. Ashby, "The Sagnac Effect in the Global Positioning System," in *Relativity in Rotating Frames*, Norwell, MA: Kluwer Academic Publishers, 2004, p. 11.

[10] M. Pusz, "P1935R2 A C++ Approach to Physical Units," 13 January 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1935r2.html.

APPENDIX A – EcefEnuProjector Math

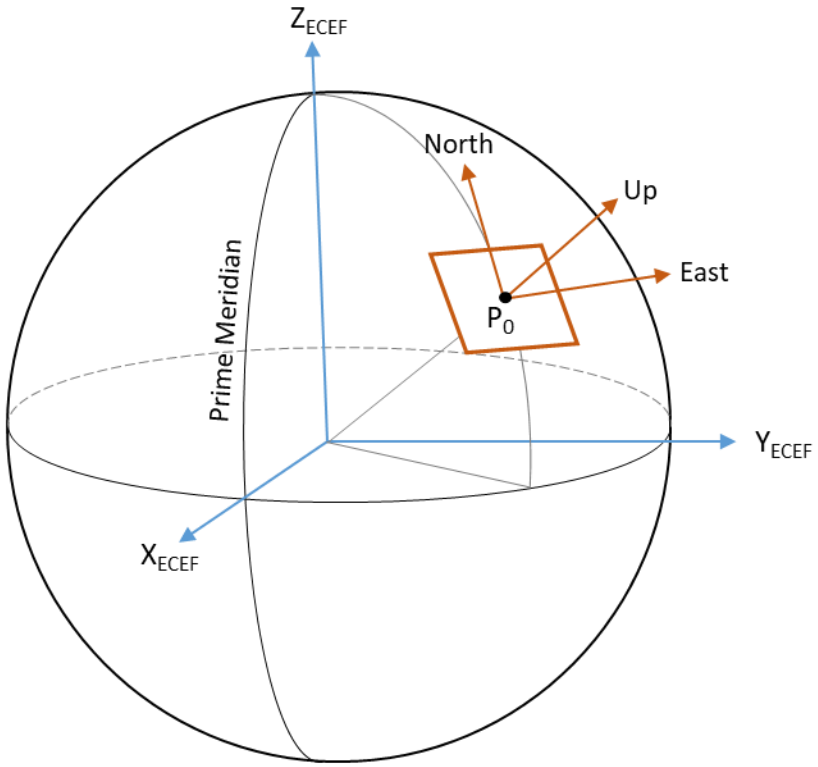

Figure 11 – Relationship between Earth-Centered Earth-Fixed (ECEF) coordinate system and a local tangential plane, East North Up (ENU) coordinate system.

Prior to conversion, we must know both the ECEF coordinates and the latitude (Lat_0) and longitude (Lon_0) of the reference point (P_0). In order for the Up vector to be normal to WGS84 ellipsoid, latitude must be geodetic rather than geocentric. To obtain the geodetic latitude, GeoLib currently uses the algorithm described by D. K. Olson [8]. This calculation is done in the `LatLonAlt( const Ecef& )` constructor which is not described here.

ECEF to ENU Coordinates

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{bmatrix} -\sin(Lon_0) & \cos(Lon_0) & 0 \\ -\sin(Lat_0)\cos(Lon_0) & -\sin(Lat_0)\sin(Lon_0) & \cos(Lat_0) \\ \cos(Lat_0)\cos(Lon_0) & \cos(Lat_0)\sin(Lon_0) & \sin(Lat_0) \end{bmatrix} \begin{bmatrix} X - X_0 \\ Y - Y_0 \\ Z - Z_0 \end{bmatrix}$$

ENU to ECEF Coordinates

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} -\sin(Lon_0) & -\sin(Lat_0)\cos(Lon_0) & \cos(Lat_0)\cos(Lon_0) \\ \cos(Lon_0) & -\sin(Lat_0)\sin(Lon_0) & \cos(Lat_0)\sin(Lon_0) \\ 0 & \cos(Lat_0) & \sin(Lat_0) \end{bmatrix} \begin{bmatrix} E \\ N \\ U \end{bmatrix} + \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$$

APPENDIX B – EnuRuvProjector Math

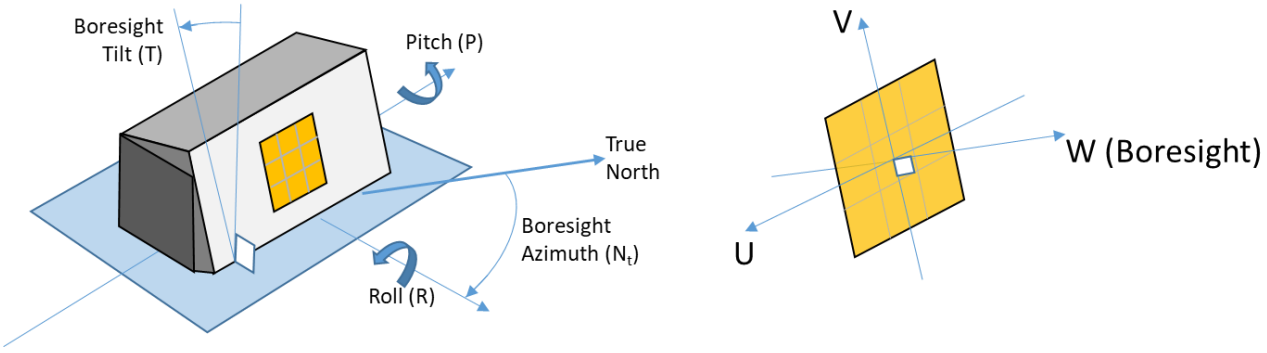

Figure 12 – Relationship between local East North Up (ENU) and sine space (UV) coordinate systems.

To simplify the conversion between RUV and ENU, the RUV coordinates are expressed as a set of right-handed Cartesian coordinates:

$$\begin{bmatrix} rU \\ rW \\ rV \end{bmatrix}$$

These coordinates would coincide with ENU coordinates if the arrays boresight were pointing true north with no other rotations. These coordinates can then be rotated according to the array's actual boresight orientation.

ENU to RUV Coordinates

$$\begin{bmatrix} rU \\ rW \\ rV \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(P+T) & \sin(P+T) \\ 0 & -\sin(P+T) & \cos(P+T) \end{bmatrix} \begin{bmatrix} \cos(R) & 0 & -\sin(R) \\ 0 & 1 & 0 \\ \sin(R) & 0 & \cos(R) \end{bmatrix} \begin{bmatrix} \cos(N_t) & -\sin(N_t) & 0 \\ \sin(N_t) & \cos(N_t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} E \\ N \\ U \end{bmatrix}$$

RUV to ENU Coordinates

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{bmatrix} \cos(N_t) & \sin(N_t) & 0 \\ -\sin(N_t) & \cos(N_t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(R) & 0 & \sin(R) \\ 0 & 1 & 0 \\ -\sin(R) & 0 & \cos(R) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(P+T) & -\sin(P+T) \\ 0 & \sin(P+T) & \cos(P+T) \end{bmatrix} \begin{bmatrix} rU \\ rW \\ rV \end{bmatrix}$$

APPENDIX C – RuvBuvProjector Math

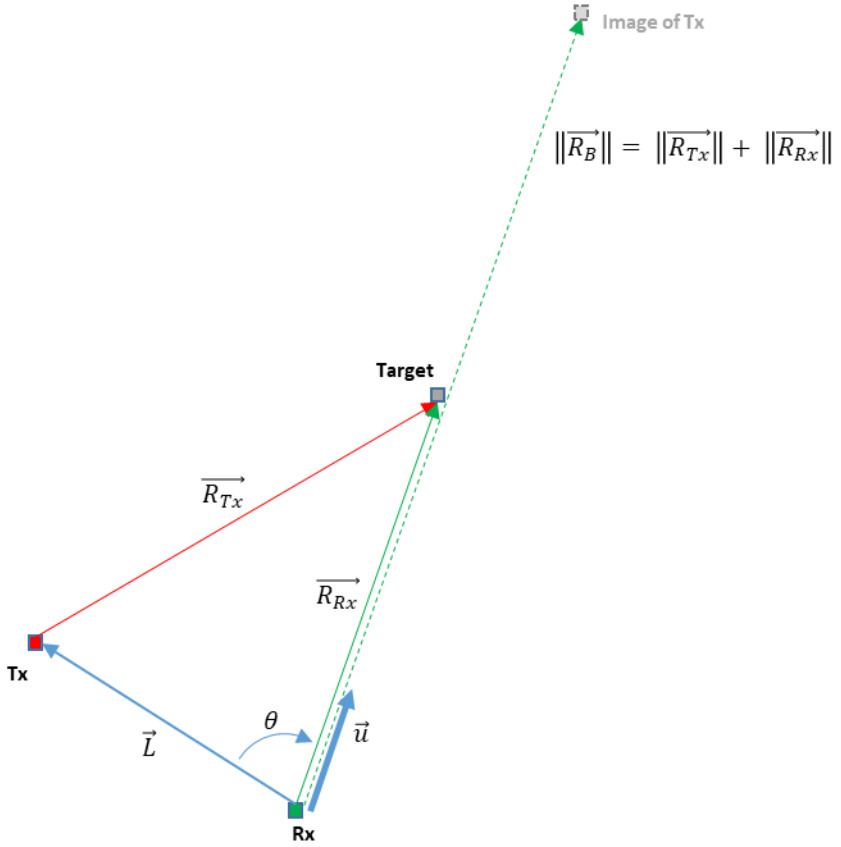

Figure 13 – Bistatic measurements.

Prior to conversion, the location of the Transmit and Receive locations must be known. The baseline vector ($\vec{L}$) is simply the location of the transmit site in the receive site's ENU reference frame. This is obtained using the EcefEnuProjector as described in Appendix A – EcefEnuProjector Math. The unit vector ($\vec{u}$) is obtained by rotating the original UV vector using the rotations described in Appendix B – EnuRuvProjector Math. The actual conversion between bistatic range ($\|\overrightarrow{R_B}\|$) and monostatic range ($\|\overrightarrow{R_{Rx}}\|$) is through the application of the *law of cosines*. The original UV vector carries through unchanged in either conversion.

BUV to RUV Coordinates

From the *law of cosines*:

$$\left(\|\overrightarrow{R_B}\| - \|\overrightarrow{R_{Rx}}\|\right)^2 = \|\vec{L}\|^2 + \|\overrightarrow{R_{Rx}}\|^2 - 2\,\|\vec{L}\|\,\|\overrightarrow{R_{Rx}}\|\cos\theta$$

or

$$\|\overrightarrow{R_{Rx}}\| = \frac{\|\overrightarrow{R_B}\|^2 - \|\vec{L}\|^2}{2\left(\|\overrightarrow{R_B}\| - \|\vec{L}\|\cos\theta\right)}$$

$$\cos\theta = \frac{\vec{u}\cdot\vec{L}}{\|u\|\,\|L\|}$$

$$\boxed{\left\|\overrightarrow{R_{Rx}}\right\| = \frac{\left\|\overrightarrow{R_B}\right\|^2 - \left\|\vec{L}\right\|^2}{2\left(\left\|\overrightarrow{R_B}\right\| - (\vec{u}\cdot\vec{L})\right)}}$$

RUV to BUV Coordinates

$$\left\|\overrightarrow{R_{Tx}}\right\|^2 = \left\|\vec{L}\right\|^2 + \left\|\overrightarrow{R_{Rx}}\right\|^2 - 2\left\|\vec{L}\right\|\left\|\overrightarrow{R_{Rx}}\right\|\cos\theta$$

$$\cos\theta = \frac{\vec{u}\cdot\vec{L}}{\|u\|\,\|L\|}$$

$$\boxed{\left\|\overrightarrow{R_B}\right\| = \left\|\overrightarrow{R_{Rx}}\right\| + \sqrt{\left\|\vec{L}\right\|^2 + \left\|\overrightarrow{R_{Rx}}\right\|^2 - 2\left\|\overrightarrow{R_{Rx}}\right\|(\vec{u}\cdot\vec{L})}}$$