Contents

List of Figures

List of Tables

Introduction to Geography and Photogrammetry

Coordinate Systems

The single most important component of GIS and Geography at large is being able to answer *where we are*. Being able to describe your position in a manner appropriate to your situation is essential to accurate navigation.

Spherical Coordinates

Most users are familiar with this first class of coordinates, defined as a *spherical* coordinate system. This is commonly referred to the *latitude* and *longitude*. Spherical coordinates in mathematics typically take the form of r, θ, and ϕ where r is the radius from the center to the coordinate, θ is the angle on the x,y plan, and ϕ is the angle from the z axis to xy plane. Figure 1 shows the relationship between Cartesian and Spherical coordinates.

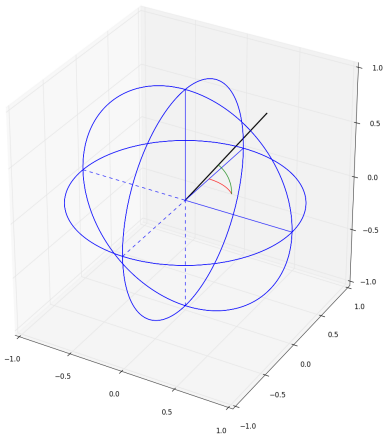

Figure 1: Relationship between spherical and cartesian coordinates.

Now to describe spherical coordinates in a Geographic setting, θ becomes *longitude*, ϕ becomes *latitude*, and r is modified from the distance from the radius, to the distance from the *datum*. Datums will be discussed later. To continue, consider a datum as an ellipsoid which models the Earth for which you can describe the

"sea-level" or zero elevation.

Geographic coordinates are commonly described in degrees. Figure 2 shows the degrees of latitude and longitude for the Lake Tahoe region of the United States.

Figure 2: Shaded relief map of Lake Tahoe showing lines of latitude and longitude.

Geographic Coordinate Representation

Geographic coordinates in a spherical format can take multiple representations. The primary focus here is on the horizontal components, namely the *latitude* and *longitude*.

In this section, we have the coordinate given for Mount Tallac, a mountain near Lake Tahoe in the Desolation Wilderness.

Decimal Degrees

Decimal Degrees formatting takes the latitude and longitude and represents them as floating-point numbers, represented in degrees, with a negative number representing the Western and Southern Hemispheres.

Latitude 38.905984

Projected Coordinates

Expressing coordinates in latitude and longitude is great for navigation as the coordinates are easily relatable to the world-wide coordinates. The issue however comes when we try to create maps and other products using spherical coordinates. Maps require a 2d *Projection* or *mapping* from the globe to a flat surface.

Universal Transverse Mercator (UTM)

GIS C++ Libraries

GDAL

For many applications, this may be the only tool you will ever need. stands for the *Geospatial Data Abstraction Library*. GDAL is largely a file input/output tool which can manage well over 100 different raster and vector files. GDAL also contains *OGR* which can do coordinate conversion using the *Proj4* library.

Project Website	`http://gdal.org`

Installing GDAL

Window

Mac OSX

The recommended method of installing GDAL is through the MacPorts package manager. There are other packages available, however MacPorts is unique in that it provides both a customizable installation through the `variant` flag and it can be updated through the package manager.

Building from Source

Before installing from source, first read the GDAL documentation at for your respective system. GDAL has a huge number of dependencies, most of which are optional. As a consequence, it may be difficult for new users to know what is needed for their respective applications. In general, here are a good set of packages to consider installing prior to building GDAL.

Package	Description	More Info
proj4	Coordinate conversions	`http://trac.osgeo.org/proj/`
openjpeg	JPEG2000 reader	`http://www.openjpeg.org/`

GeographicLib

GeographicLib is a C++ library similar to GeoTrans. It performs primarily coordinate conversions, similar to Proj. For many functions, GDAL/OGR may be enough. Where GeographicLib really shines is its support for gravity models, geoid models, and magnetic models. It also contains the utilities to find and import the required data.

As code samples in this book will illustrate, Geoid models are very useful for

Coordinate Conversions

Overview

Converting coordinates is an essential component in GIS. Whether you are converting one single coordinate, or reprojecting an entire dataset, having a useful toolset can be extremely helpful.

Geographic to UTM using GDAL

In this example, we will use GDAL to project Geographic coordinates to Universal Transverse Mercator (UTM).

Referring back to the UTM overview, in order to convert a Geographic coordinate to UTM, we need to first compute the desired grid zone. To accomplish this, we use the following utility function.

```
int Compute_UTM_Grid_Zone( const double& latitude_degrees,
                           const double& longitude_degrees )
{
    // Create zone using standard algorithm
    int grid_zone = std::floor((longitude_degrees + 180)/6) + 1;

    // Check if we are around an exception near Norway
    if( latitude_degrees  >= 56.0 && latitude_degrees  < 64.0 &&
            longitude_degrees >= 3.0  && longitude_degrees < 12.0 )
    {
        return 32;
    }

    // Check for Svalbard
    if( latitude_degrees >= 72.0 && latitude_degrees < 84.0 )
    {
        if( longitude_degrees >= 0.0  && longitude_degrees <  9.0 ){
            return 32;
        }
        else if( longitude_degrees >= 9.0  && longitude_degrees < 21.0 ){
            return 33;
        }
        else if( longitude_degrees >= 21.0 && longitude_degrees < 33.0 ){
            return 35;
        }
        else if( longitude_degrees >= 33.0 && longitude_degrees < 42.0 ){
            return 37;
        }
    }
    return grid_zone;
}
```

This is needed because GDAL/OGR requires you set the UTM Grid zone before converting to UTM. The reason for this is that you may need to cover a region which needs to span more than 1 UTM grid zone. Each UTM zone has a different origin.

Once the grid zone is determined given the latitude and longitude, the rest is straightforward. This example assumes a WGS84 datum.

```
1
2  // GDAL Libraries
3  #include <ogr_spatialref.h>
4
5
6  // C++ Libraries
7  #include <iomanip>
8  #include <iostream>
9
10
11 // Common Libraries
12 #include "../common/Coordinate_Utilities.hpp"
13
14
15 /**
16  * @brief Main Function
17  */
18 int main( int argc, char* argv[] )
19 {
20
21     // Geographic Coordinates
22     const double latitude_degrees  = 39.5;
23     const double longitude_degrees = -119.5;
24     const double elevation_meters  = 1234;
25     const std::string input_datum  = "WGS84";
26     const std::string output_datum = "WGS84";
27     const bool is_northern = (latitude_degrees >= 0);
28
29     // Compute the Recommended UTM Grid Zone
30     int grid_zone = Compute_UTM_Grid_Zone( latitude_degrees,
31                                            longitude_degrees);
32
33
34     // Latitude Band
35     char lat_band = Compute_UTM_Latitude_Band( latitude_degrees );
36
37
38     // Create the Spatial Reference Objects
39     OGRSpatialReference sourceSRS, targetSRS;
40
41     sourceSRS.SetWellKnownGeogCS(  input_datum.c_str() );
42     targetSRS.SetWellKnownGeogCS( output_datum.c_str() );
43
44     // Configure the Projected Coordinate Components
45     targetSRS.SetUTM( grid_zone,
46                       is_northern );
47
48     // Build the Transform Engine
49     OGRCoordinateTransformation* transform;
50     transform = OGRCreateCoordinateTransformation( &sourceSRS,
51                                                    &targetSRS );
52
53     double easting_meters          = longitude_degrees;
54     double northing_meters         = latitude_degrees;
55     double output_elevation_meters = elevation_meters;
56
57     if( !transform->Transform( 1, &easting_meters,
58                                &northing_meters,
59                                &output_elevation_meters ) )
60     {
61         throw std::runtime_error("Transformation Failed.");
62     }
63
```

```
64
65      // Destroy the Transform
66      OCTDestroyCoordinateTransformation( transform );
67
68      // Print Results
69      std::cout << std::fixed;
70      std::cout << "UTM␣Grid␣Zone␣␣␣:␣" << grid_zone << std::endl;
71      std::cout << "Latitude␣Band␣␣␣:␣" << lat_band << std::endl;
72      std::cout << "Easting␣Meters␣␣:␣" << easting_meters << std::endl;
73      std::cout << "Northing␣Meters␣:␣" << northing_meters << std::endl;
74      std::cout << "Elevation␣Meters:␣" << output_elevation_meters << std::endl;
75
76      return 0;
77  }
```

Querying Geo-Magnetic Information with GeographicLib

Prior to GPS, heading information was attained by using a magnetic compass. Compasses still have a useful purpose and as such, knowing about the Earth's magnetic field is equally useful. GeographicLib has an impressive amount of built-in, yet extensible support for magnetic information. The `MagneticModel` class in the library provides this functionality.

In the demo below, we will be using GeographicLib to take a Geographic Coordinate in decimal degrees and compute the magnetic information for that position. Of particular importance to many users will be the ability to extract the *declination* or angle from North to the Earth's magnetic North Pole.

```
1   /**
2    * @file    geographiclib-magnetic-query.cpp
3    * @author  Marvin Smith
4    * @date    11/12/2015
5   */
6
7   // C++ Libraries
8   #include <exception>
9   #include <iostream>
10
11  #include <GeographicLib/MagneticModel.hpp>
12
13
14  /**
15   * @brief Main Function
16  */
17  int main( int argc, char* argv[] )
18  {
19      // Fetch the components
20      const double lat  = std::stod(argv[1]);
21      const double lon  = std::stod(argv[2]);
22      const double elev = std::stod(argv[3]);
23      const double year = std::stoi(argv[4]);
24
25
26      try
27      {
28
29          // Define the GeographicLib Magnetic Model Object
30          GeographicLib::MagneticModel mag("wmm2010");
31
32          double Bx, By, Bz;
33
34          mag( year, lat, lon, elev,
35               Bx, By, Bz);
36
37          double H, F, D, I;
38
```

```
39
40          GeographicLib::MagneticModel::FieldComponents(Bx, By, Bz, H, F, D, I);
41          std::cout << H << "␣" << F << "␣" << D << "␣" << I << "\n";
42
43      }
44
45      catch (const std::exception& e)
46      {
47          std::cerr << "Caught␣exception:␣" << e.what() << "\n";
48          return 1;
49      }
50
51      return 0;
52  }
```

Appendix A : Common Lookup Information

This appendix seeks to provide the reader with common geographic and programming information.

Coordinate Lookup Table

All information for this lookup table was attained using the Wikipedia and their GeoHack utility [1].

Location Name	Geographic-DD	UTM
Mount Tallac	38.905984 -120.098975	10S 751560 4310345

Bibliography

[1] Wikimedia Foundation Labs. Geohack. `https://tools.wmflabs.org/geohack/`, November 2015.

Index