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.

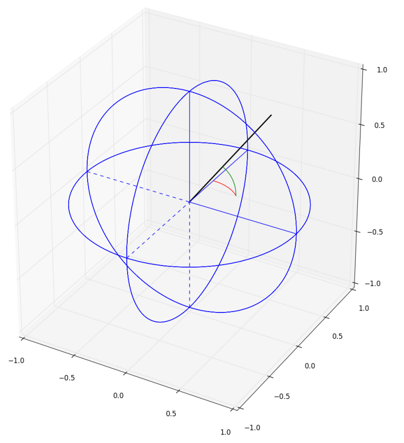

Figure 1: Relationship between spherical and cartesian coordinates.

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.

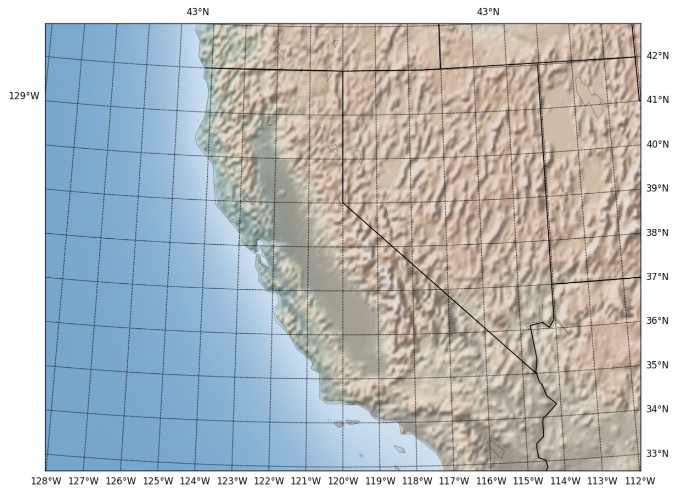
43°N
43°N
129°W
42°N
41°N
40°N
39°N
38°N
37°N
36°N
35°N
34°N
33°N
128°W 127°W 126°W 125°W 124°W 123°W 122°W 121°W 120°W 119°W 118°W 117°W 116°W 115°W 114°W 113°W 112°W

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)

The Universal Transverse Mercator (UTM) is a projected coordinate system based on the standard *Transverse Mercator* projection. There are 120 grid zones defined: 60 for Northern and 60 for Southern Hemispheres.

Universal Polar Stereographic (UPS)

Military Grid Reference System (MGRS)

United States National Grid (USNG)

Whereas MGRS is a grid-based projection coordinate system for the entire planet, the United States National Grid (USNG) covers only the United States. USNG uses MGRS as the fundamental projection. In fact, according to [1, p. 3.1], MGRS and USNG coordinates are identical.

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

Windows

Window bindings can be downloaded for various MSVC versions here at `http://www.gisinternals.com/`. This is a great resource if you need development libraries for a recent or non-standard VC++ version.

Another resource for Windows versions of GDAL is the OSGeo4W distribution. This is a tool similar to Cygwin which provides the latest versions of GDAL packaged with other useful tools such as QGis, GRASS, MapServer and OpenEV. Information can be found on their site at `https://trac.osgeo.org/osgeo4w/`.

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.

```
sudo port install gdal
```

Current MacPorts version is 2.0.1 and contains many different port variants. Recommend using at least the *openjpeg, geos, lzma, netcdf, and poppler* variants.

Linux

GDAL can easily be installed through multiple distributions.
Ubuntu

```
 apt-get install gdal-bin libgdal-dev
```

Fedora/RHEL/Centos

```
yum install gdal-devel gdal-python
```

Red-Hat Linux 6
Due to the stable nature of RHEL distributions, it is recommended to use the ELGIS repo provided by OSGeo. Their Yum repositories can be configured via the following commands.

```
sudo rpm -Uvh http://elgis.argeo.org/repos/6/elgis-release-6-6_0.noarch.rpm
```

Caveats

In order to use ELGIS, you must enable the EPEL repository. Their repos can be configured at `http://fedoraproject.org/wiki/EPEL`.

Do not enable both ELGIS and PGRPMS. They are not compatible and if both enabled, will cause conflicts.

More information can be found on their site at `http://wiki.osgeo.org/wiki/Enterprise_Linux_GIS`.

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.

Another strong feature of GeographicLib is support for MGRS. While MGRS is part of the GDAL NITF driver, it is not natively supported. GeographicLib makes this much easier.

Managing Coordinates and Projections

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. In addition, you may need additional information such as magnetic north, or even the relative position of the tides to sea-level in a particular location. This chapter will provide you with the essential information to be able to more accurately describe the world around you.

Coordinate Conversions

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.

```
 * @return Grid zone value.
*/
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;
        }
    }
```

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.

```
// GDAL Libraries
#include <ogr_spatialref.h>


// C++ Libraries
#include <iomanip>
#include <iostream>


// Common Libraries
#include "../common/Coordinate_Utilities.hpp"


/**
 * @brief Main Function
 */
int main( int argc, char* argv[] )
{

    // Geographic Coordinates
    const double latitude_degrees  = 39.5;
    const double longitude_degrees = -119.5;
    const double elevation_meters  = 1234;
    const std::string input_datum  = "WGS84";
    const std::string output_datum = "WGS84";
    const bool is_northern = (latitude_degrees >= 0);

    // Compute the Recommended UTM Grid Zone
    int grid_zone = Compute_UTM_Grid_Zone( latitude_degrees,
                                           longitude_degrees);


    // Latitude Band
    char lat_band = Compute_UTM_Latitude_Band( latitude_degrees );
    // Create the Spatial Reference Objects
    OGRSpatialReference sourceSRS, targetSRS;

    sourceSRS.SetWellKnownGeogCS(  input_datum.c_str() );
    targetSRS.SetWellKnownGeogCS( output_datum.c_str() );

    // Configure the Projected Coordinate Components
    targetSRS.SetUTM( grid_zone,
                      is_northern );

    // Build the Transform Engine
    OGRCoordinateTransformation* transform;
    transform = OGRCreateCoordinateTransformation( &sourceSRS,
                                                   &targetSRS );

    double easting_meters          = longitude_degrees;
    double northing_meters         = latitude_degrees;
    double output_elevation_meters = elevation_meters;

    if( !transform->Transform( 1, &easting_meters,
                                  &northing_meters,
                                  &output_elevation_meters ) )
    {
        throw std::runtime_error("Transformation Failed.");
    }


    // Destroy the Transform
    OCTDestroyCoordinateTransformation( transform );

    // Print Results
    std::cout << std::fixed;
    std::cout << "UTM Grid Zone   : " << grid_zone << std::endl;
    std::cout << "Latitude Band   : " << lat_band << std::endl;
    std::cout << "Easting Meters  : " << easting_meters << std::endl;
    std::cout << "Northing Meters : " << northing_meters << std::endl;
    std::cout << "Elevation Meters: " << output_elevation_meters << std::endl;

    return 0;
}
```

Converting between UTM and MGRS using GeographicLib

In this example, we will convert between Universal Transverse Mercator (UTM) and the Military Grid Reference System (MGRS) using GeographicLib.

The key item in the GeographicLib package is the ‘MGRS‘ class. This class has a Forward and Reverse transformation method, plus some getters in order to query ellpsoid parameters.

```
int main( int argc, char* argv[] )
{
    // Grab stdin
    const std::string mgrs_string = argv[1];
```

```
std::string resulting_mgrs_string;

// Print the input
std::cout << "MGRS Coordinate: " << mgrs_string << std::endl;

// UTM Elements
int grid_zone;
bool is_northern;
double easting_meters, northing_meters;
int precision_100km;
bool from_center = true;


// Convert to UTM
MGRS::Reverse( mgrs_string,
               grid_zone,
               is_northern,
               easting_meters,
               northing_meters,
               precision_100km,
               from_center );

// Print the results
std::cout << "UTM Coordinate" << std::endl;
std::cout << " - grid zone  : " << grid_zone << std::endl;
std::cout << " - is northern: " << std::boolalpha << is_northern << std::endl;
std::cout << " - easting    : " << easting_meters << std::endl;
std::cout << " - northing   : " << northing_meters << std::endl;
std::cout << " - precision  : " << precision_100km << std::endl;


// Forward back to MGRS
MGRS::Forward( grid_zone,
               is_northern,
               easting_meters,
               northing_meters,
               precision_100km,
               resulting_mgrs_string );

// Print the result
std::cout << std::endl;
std::cout << "Result MGRS String: " << resulting_mgrs_string << std::endl;
```

Distance Measurements

Computing the distance between two points is yet another unique challenge in Geography. To many, computing distance on a map seems like a simple problem. In Cartesian coordinates, the most often used metric is the *Distance Formula.*

Given $P_1(X, Y)$ and $P_2(X, Y)$, the Cartesian distance is simply,

$$d = \sqrt{\left(P_{2_x} - P_{1_x}\right)^2 + \left(P_{2_y} - P_{1_y}\right)^2} \tag{1}$$

or more formally, given N-dimensional coordinates P_1 and P_2,

$$d = \sqrt{\sum_{i=1}^{N} (P_{2_i} - P_{1_i})^2} \tag{2}$$

The issue with Cartesian coordinates is that they are only appropriate for certain conditions. Geographic (lat/lon) uses a spherical coordinate system which represents positions in angles from the origin, not distance from the origin. In figure 3, the differences in routes between the two same points can be observed between an orthographic and mercator projection.

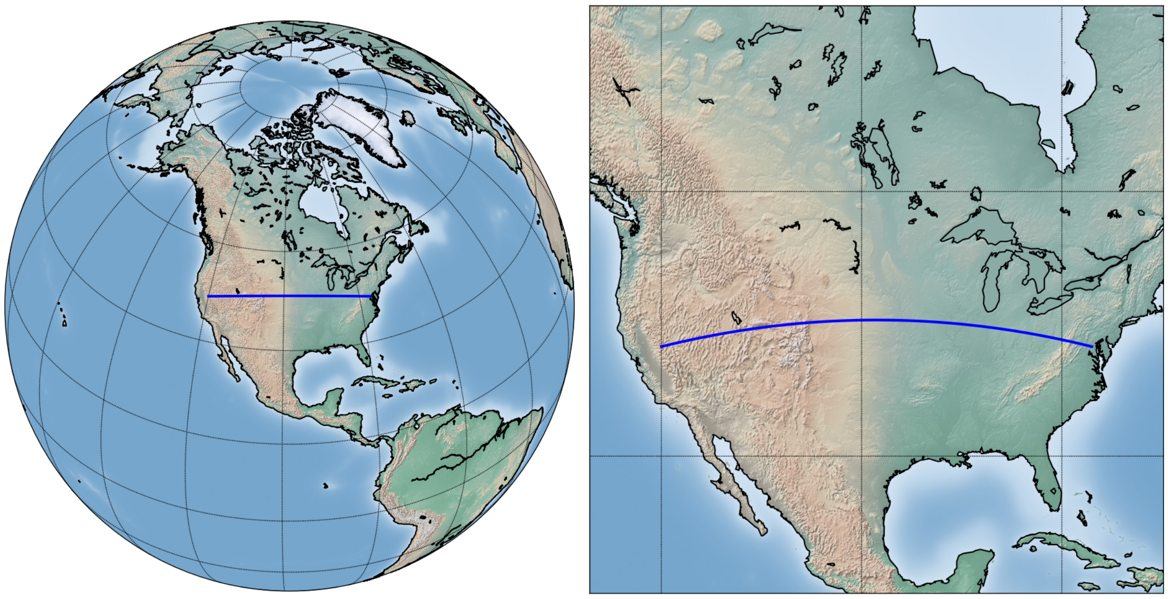

Figure 3: Orthographic vs. Mercator Projection for line between (39,-120) to (39,-77)

Coordinate Distances for Geographic Coordinates

Great Circle Distances

The *Great Circle* is the intersection of a plane with a sphere. This intersection results in a circle which represents the shortest distance between any two points on a sphere[3, p. 108]. This section is not meant to provide a useful navigational aid, but rather to provide the user with a simple computational example. Great Circle distances are no longer popular in GIS as Datums are now in Ellipsoids and computational performance now renders the usefulness less relevant. A popular equation for this is the *Haversine* formula. Avoid the technical great circle equation as it does not perform well under small distances.

The equation is given where Latitude is *phi*, Longitude is λ, and the Earth's radius is r.

$$d_{gc} = 2r \arcsin\left(\sqrt{\sin^2 \frac{\phi_2 - \phi_1}{2} + \cos\phi_1 + \cos\phi_2 \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{3}$$

Here is a C++ example of the Haversine equation. This example was derived partially from [3, p. 109].

```
/**
 * @file   great-circle-distance.cpp
 * @author Marvin Smith
 * @date   11/21/2015
*/

// C++ Libraries
#include <cmath>
#include <iostream>

using namespace std;

/**
 * @brief Compute Great Circle Distance between two coordinates.
 *
 * @param[in] lat_1_degrees
 * @param[in] lon_1_degrees
 * @param[in] lat_1_degrees
 * @param[in] lon_1_degrees
*/
double Haversine_Distance( const double& lat_1_degrees,
                           const double& lon_1_degrees,
                           const double& lat_2_degrees,
                           const double& lon_2_degrees,
                           const double& radius_meters )
{
    // Convert Degrees to Radians
    const double lat_1_rad = lat_1_degrees * M_PI / 180.0;
    const double lon_1_rad = lon_1_degrees * M_PI / 180.0;
    const double lat_2_rad = lat_2_degrees * M_PI / 180.0;
    const double lon_2_rad = lon_2_degrees * M_PI / 180.0;

    // Compute Differences
    const double h_lat = std::pow( std::sin((lat_2_rad - lat_1_rad)/2), 2);
    const double h_lon = std::pow( std::sin((lon_2_rad - lon_1_rad)/2), 2);

    // Compute result
    const double asin_val = std::sqrt(h_lat + std::cos(lat_1_rad) * std::cos(lat_2_rad) * h_lon);

    return (2 * radius_meters * std::asin( asin_val ));
}

int main()
{
    // Coordinate Values
    const double lat_1_deg = 39.5;
    const double lon_1_deg = -120.5;
    const double lat_2_deg = 39;
    const double lon_2_deg = -120;
    const double earth_radius_meters = 6378137.0;

    // Compute Distance in Meters
    const double dist_meters = Haversine_Distance( lat_1_deg,
                                                   lon_1_deg,
                                                   lat_2_deg,
                                                   lon_2_deg,
                                                   earth_radius_meters );

    // Print Distance
    std::cout << std::fixed << "Distance (m): " << dist_meters << std::endl;

    return 0;
}
```

Geodesic Ellipsoid Distances

GeographicLib once again is a useful utility for Geodesic distances.

```
/**
 * @file    geographiclib-coordinate-distance.cpp
 * @author  Marvin Smith
 * @date    11/21/2015
*/

// C++ Libraries
#include <exception>
#include <iostream>


// GeographicLib Libraries
#include <GeographicLib/Constants.hpp>
#include <GeographicLib/Geodesic.hpp>


namespace GeoLib=GeographicLib;


int main( int argc, char* argv[] )
{

    try
    {
        // Define Coordinates: Mt. Tallac
        const double lat_1_deg =   38.905984;
        const double lon_1_deg = -120.098975;

        // Define Coordinates: Mt. Rose
        const double lat_2_deg =   39.343778;
        const double lon_2_deg = -119.917889;

        // GRS80 Compontents
        const double a = GeoLib::Constants::GRS80_a();
        const double f = 298.257222101;
        // Define our Datum
        GeoLib::Geodesic geodesic( a, f);


        // Compute the Distance between Coordinates
        double distance_meters;
        double azimuth1, azimuth2;
        geodesic.Inverse( lat_1_deg,
                          lon_1_deg,
                          lat_2_deg,
                          lon_2_deg,
                          distance_meters,
                          azimuth1,
                          azimuth2 );


        // Compute the Direct and Compute the Endpoint
        double result_lat, result_lon;
        geodesic.Direct( lat_1_deg,
                         lon_1_deg,
                         azimuth1,
                         distance_meters,
                         result_lat,
                         result_lon );


        // Print info
        std::cout << "Coordinate 1 (deg): " << lat_1_deg << ", " << lon_1_deg << std::endl;
        std::cout << "Coordinate 2 (deg): " << lat_2_deg << ", " << lon_2_deg << std::endl;
        std::cout << "Distance      (m) : " << distance_meters << std::endl;
        std::cout << "Azimuth 1    (deg): " << azimuth1 << std::endl;
        std::cout << "Azimuth 2    (deg): " << azimuth2 << std::endl;
        std::cout << "Direct Coord (deg): " << result_lat << ", " << result_lon << std::endl;


    }
    catch ( const std::exception& e )
    {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }

    return 0;
}
```

Projected Coordinate Distances

Digital Elevation Models (DEMs)

Magnetism, Gravitation, and Tidal Forces

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.

```
/**
 * @file    geographiclib-magnetic-query.cpp
 * @author  Marvin Smith
 * @date    11/12/2015
*/

// C++ Libraries
#include <exception>
#include <iostream>

#include <GeographicLib/MagneticModel.hpp>


/**
 * @brief Main Function
*/
int main( int argc, char* argv[] )
{
    // Fetch the components
    const double lat  = std::stod(argv[1]);
    const double lon  = std::stod(argv[2]);
    const double elev = std::stod(argv[3]);
    const double year = std::stoi(argv[4]);


    try
    {

        // Define the GeographicLib Magnetic Model Object
        GeographicLib::MagneticModel mag("wmm2010");

        double Bx, By, Bz;

        mag( year, lat, lon, elev,
             Bx, By, Bz);

        double H, F, D, I;


        GeographicLib::MagneticModel::FieldComponents(Bx, By, Bz, H, F, D, I);
        std::cout << H << " " << F << " " << D << " " << I << "\n";

    }

    catch (const std::exception& e)
    {
        std::cerr << "Caught exception: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

Appendix A : Common Lookup Information

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

Coordinate Lookup Table

All location positions in Geographic (lat/lon) were attained using the Wikipedia and their GeoHack utility [2]. All conversions to other coordinate systems was performed by GEOTRANS[4].

Location Name	**Geographic-DD**	**Geographic-DMS**	**UTM**	**USNG**	**MGRS**
Mount Tallac	38.905984 -120.098975	38d 54' 21.54"N 120d 5' 56.31"W	10S 751560 4310345	10SGJ5155910345	10SGJ5155910345

Bibliography

[1] Federal Geographic Data Committee. United states national grid. http://www.fgdc.gov/standards/projects/FGDC-standards-projects/usng/fgdc_std_011_2001_usng.pdf, December 2001. FGDC-STD-011-2001.

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

[3] Thomas Meyer. *Introduction to Geometrical and Physical Geodesy : Foundations of Geomatics.* ESRI Press, Redlands, Calif, 2010.

[4] Office of Geomatics. Msp geotrans 3.5 (geographic translator). http://earth-info.nga.mil/GandG/geotrans, May 2015. GEOTRANS Software Application.

Index