# RPCMapper Documentation

This documentation file explains the members and methods of the RPCMapper class developed for the NGA Pathfinder program. The code itself also has significant inline documentation.

## What This Code Does:

This class mathematically transforms a geodetic point expressed in Latitude, Longitude, and Altitude into a 2 dimensional point expressed in X and Y coordinates according to the Rapid Positioning Coefficient (also known as Rational Polynomial Coefficient) model. The X and Y coordinates correspond to a location on an imaging sensor where the geodetic point in question is focused. Thus this class maps the geodetic location in object space to its semantic representation on the image plane.

### The RPC Algorithm Walkthrough:

The RPC algorithm accepts as input a geodetic point expressed in Latitude, Longitude, and Altitude (LLA).

The algorithm then applies a set of biases and scale factors (provided by the Geotiff or other meta-data file) to the LLA so that the scaled LLA.

All valid “Scaled” entries exist between +/- 1 so if any of the LLAScaled values are outside that range then the point is deemed invalid for this set of RPC coefficients and processing should be abandoned.

The Scaled values are then aggregated into a 20 dimensional coefficient vector by concatenating the permutations of Latitudei \* Longitudej \*Altitiduek for i, j, and k in the set [0,1,2,3]

An important note on this vector is that the order matters. The interested reader is referred to

<http://www.gwg.nga.mil/ntb/baseline/docs/RPC/RPC00A%20vs%20RPC00B%20White%20Paper.docx>

Once the LLACoeff vector has been generated we use the RPC coefficients for the X Numerator, X Denominator, Y Numerator, and Y Denominator (again provided in the meta data) to calculate scaled (between -/+ 1) X and Y sensor locations.

The final step in the process is to turn the scaled values into sensor pixel locations. We do that by multiplying the scaled values by the scale factor and then adding a bias (both again provided by the meta-data).

And that completes the algorithm.

A more detailed explanation of the algorithm used in this class for RPC Mapping can be found in:

“Mathematical Model for 3D Feature Extraction from Multiple Satellite Images Described by RPCs”

J. Grodecki, G. Dial, J. Lutes : ASPRS Annual Conference Proceedings , May 2004

## Class Style Guide:

1. Readability was chosen over efficiency so that debuggers and porters could follow the algorithm easily
2. the class members and methods are grouped by overall function
3. All member variables are private.
4. All member variables have public get/set accessor functions.
5. When appropriate member variables have update accessor functions.
6. Static variables start with a lowercase "s".
7. Member variables start with a lowercase "m".
8. Local variables start with a lowercase "l".
9. Loop variables are a single lowercase letter. In our case we used "i"
10. Method return variables are all labeled "rvalue". They are used to insure that a method only has 1 exit point.
11. Return values (rvalue) are defaulted to indicate failure either through assignment to "false" or "NaN"
12. Accessor methods start with a lowercase "a".
13. Calculation methods start with a lowercase "c".
14. Void returns are discouraged.

## Class Structures:

public:

struct ImageCoordinates{

int XValue;

int YValue;

};

The ImageCoordinates structure exist to represent a sensor location.

## Class Static Members:

public:

const static int sLLASize = 3;

const static int sCoefficientSize = 20;

const static int sPixelSize = 2;

sLLASize - used to define arrays holding geodetic information (Lat, Lon, and Alt)

sCoefficientSize - the number of entries each Coefficient array should expect

sPixelSize – used to define arrays holding sensor data (X,Y)

## Class Instance Members:

double mLLAOffsets[sLLASize];

double mLLAScaleFactors[sLLASize];

double mLLAMultiples[sCoefficientSize];

double mPixelOffsets[sPixelSize];

double mPixelScaleFactors[sPixelSize];

double mRPCBoundingBox[sLLASize \* 2];

double mXDenCoefficients[sCoefficientSize];

double mXNumCoefficients[sCoefficientSize];

double mYDenCoefficients[sCoefficientSize];

double mYNumCoefficients[sCoefficientSize];

mLLAOffsets:

holds the 3 geodetic biases used by the RPC algorithm

mLLAScaleFactors:

holds the 3 geodetic scale factors used by the RPC algorithm

mLLAMultiples:

holds the 20 permutations of (Lat)i\*(Lon)j\*(Alt)k for i,j,k in [0,1,2,3] for a the geodetic coordinate used as input to the RPC algorithm

mPixelOffsets:

holds the 2 sensor array biases used by the RPC algorithm

mPixelScaleFactors:

holds the 2 sensor array scale factors used by the RPC algorithm

mRPCBoundingBox:

holds the 6 values (LLA min and max values) that define the 3D volume of geodetic coordinates for which the current RPC coefficients are valid

mXDenCoefficients:

holds the 20 coefficients that are projected onto the mLLAMultiples array to determine the denominator of the fraction that defines the normalized X position on the sensor array.

mXNumCoefficients:

holds the 20 coefficients that are projected onto the mLLAMultiples array to determine the numerator of the fraction that defines the normalized X position on the sensor array.

mYDenCoefficients:

holds the 20 coefficients that are projected onto the mLLAMultiples array to determine the denominator of the fraction that defines the normalized Y position on the sensor array.

mYNumCoefficients:

holds the 20 coefficients that are projected onto the mLLAMultiples array to determine the numerator of the fraction that defines the normalized Y position on the sensor array.

## Class Accessor Methods:

### Get Accessors

const double \* aGetLLAOffsets();

double aGetLLAOffsetsElement(int i);

const double \* aGetLLAScaleFactors();

double aGetLLAScaleFactorsElement(int i);

const double \* aGetLLAMultiples();

double aGetLLAMultiplesElement(int i);

const double \* aGetPixelOffsets();

double aGetPixelOffsetsElement(int i);

const double \* aGetPixelScaleFactors();

double aGetPixelScaleFactorsElement(int i);

const double \* aGetRPCBoundingBox();

double aGetRPCBoundingBoxElement(int i);

const double \* aGetXDenCoefficients();

double aGetXDenCoefficientsElement(int i);

const double \* aGetXNumCoefficients();

double aGetXNumCoefficientsElement(int i);

const double \* aGetYDenCoefficients();

double aGetYDenCoefficientsElement(int i);

const double \* aGetYNumCoefficients();

double aGetYNumCoefficientsElement(int i);

The Get Accessor functions all work in one of two ways that are self-evident from their naming convention.

aGet<Variable>:

Returns a const pointer to the array in question. Note that the pointer is a constant so the pointer is useful for reading the arrays, but you cannot override array elements from these pointers.

aGet<Variable>Element:

Returns a copy of the element in the ith index of the variable array.

### Set Accessors:

bool aSetLLAOffsetsElement(int index, double value);

bool aSetLLAScaleFactorsElement(int index, double value);

bool aSetLLAMultiplesElement(int index, double value);

bool aSetPixelOffsetsElement(int index, double value);

bool aSetPixelScaleFactorsElement(int index, double value);

bool aSetRPCBoundingBoxElement(int index, double value);

bool aSetXDenCoefficientsElement(int index, double value);

bool aSetXNumCoefficientsElement(int index, double value);

bool aSetYDenCoefficientsElement(int index, double value);

bool aSetYNumCoefficientsElement(int index, double value);

The Set Accessor functions all work the same way.

aSet<Variable>Element:

Sets the ith index element to the value passed to the method.

### Update Accessors:

bool aUpdateLLAMultiples(double scaledLat, double scaledLon, double scaledAlt);

This is the only Update Accessor in the class.

aUpdateLLAMultiples:

Sets the mLLAMultiples for the scaled LLA values being passed to it. The LLA variables being passed in are all scaled between +/- 1 according to the RPC algorithm model.

## Class Calculation Methods:

bool cGenerateBoundingBox();

bool cIsMappable(double lat, double lon, double alt);

double cScaleAltitude(double alt);

double cScaleLatitude(double lat);

double cScaleLongitude(double lon);

double cScaledX();

double cScaledY();

double cSValue(double scaledX);

double cLValue(double scaledY);

cGenerateBoundingBox:

Calculates the volume in which the RPCs are valid by calculating the Latitude, Longitude, and Altitude coordinates that correspond to +/-1 scaled Latitude, Longitude, and Altitude values.

cIsMappable:

Checks to make sure that the Latitude, Longitude, and Altitude that are passed into the function are within the RPC Bounding Box.

cScaleAltitude:

Scales the Altitude that it is given as input to the range [-1,1].

cScaleLatitude:

Scales the Latitude that it is given as input to the range [-1,1].

cScaleLongitude:

Scales the Longitude that it is given as input to the range [-1,1].

cScaledX:

Maps the current LLA (as specificed by the values of mLLAMultiples) to an value in the range [-1,1]. This value is in the Sample direction.

cScaledY:

Maps the current LLA (as specificed by the values of mLLAMultiples) to an value in the range [-1,1]. This value is in the Line direction.

cSValue:

Maps the ScaledX value input to an actual column on the sensor image.

cLValue:

Maps the ScaledY value input to an actual row on the sensor image.

## How To Use This Class:

### Prerequisites:

1. The RPCMapper.h file should be included via #include <RPCMapper.h> in your code and the directory holding the include file should be included during your compilation.
2. Using the class requires you to have access to the RPC coefficient values. Typically these are read from a file. Here we assume that you have access to them by some a priori means.

### Code Execution Example:

RPCMapper lMapper = RPCMapper::RPCMapper();

//set LLA offsets and scale factors

lMapper.aSetLLAOffsetsElement(0, latitude\_offset);

lMapper.aSetLLAOffsetsElement(1, longitude\_offset);

lMapper.aSetLLAOffsetsElement(2, altitude\_offset);

lMapper.aSetLLAScaleFactorsElement(0, latitude\_scale);

lMapper.aSetLLAScaleFactorsElement(1, longitude\_scale);

lMapper.aSetLLAScaleFactorsElement(2, altitude\_scale);

//set pixel offsets and scale factors

lMapper.aSetPixelOffsetsElement(0, sample\_offset);

lMapper.aSetPixelOffsetsElement(1, line\_offset);

lMapper.aSetPixelScaleFactorsElement(0, sample\_scale);

lMapper.aSetPixelScaleFactorsElement(1, line\_scale);

//set the mapping coefficients

for (int i = 0; i<20; i++){

lmMapper.aSetYNumCoefficientsElement(i, YNumCoefficient[i]);

lmMapper.aSetYDenCoefficientsElement(i, YDenCoefficient[i]);

lmMapper.aSetXNumCoefficientsElement(i, XNumCoefficient[i]);

lmMapper.aSetXDenCoefficientsElement(i, XDenCoefficient[i]);

}

lMapper.cGenerateBoundingBox()

//everything is now setup and you can call the mapping function as much as you want

ImageCoordinates lSensorLocation = lMapper.MapLLA(latitude, longitude, altitude);