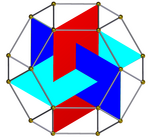
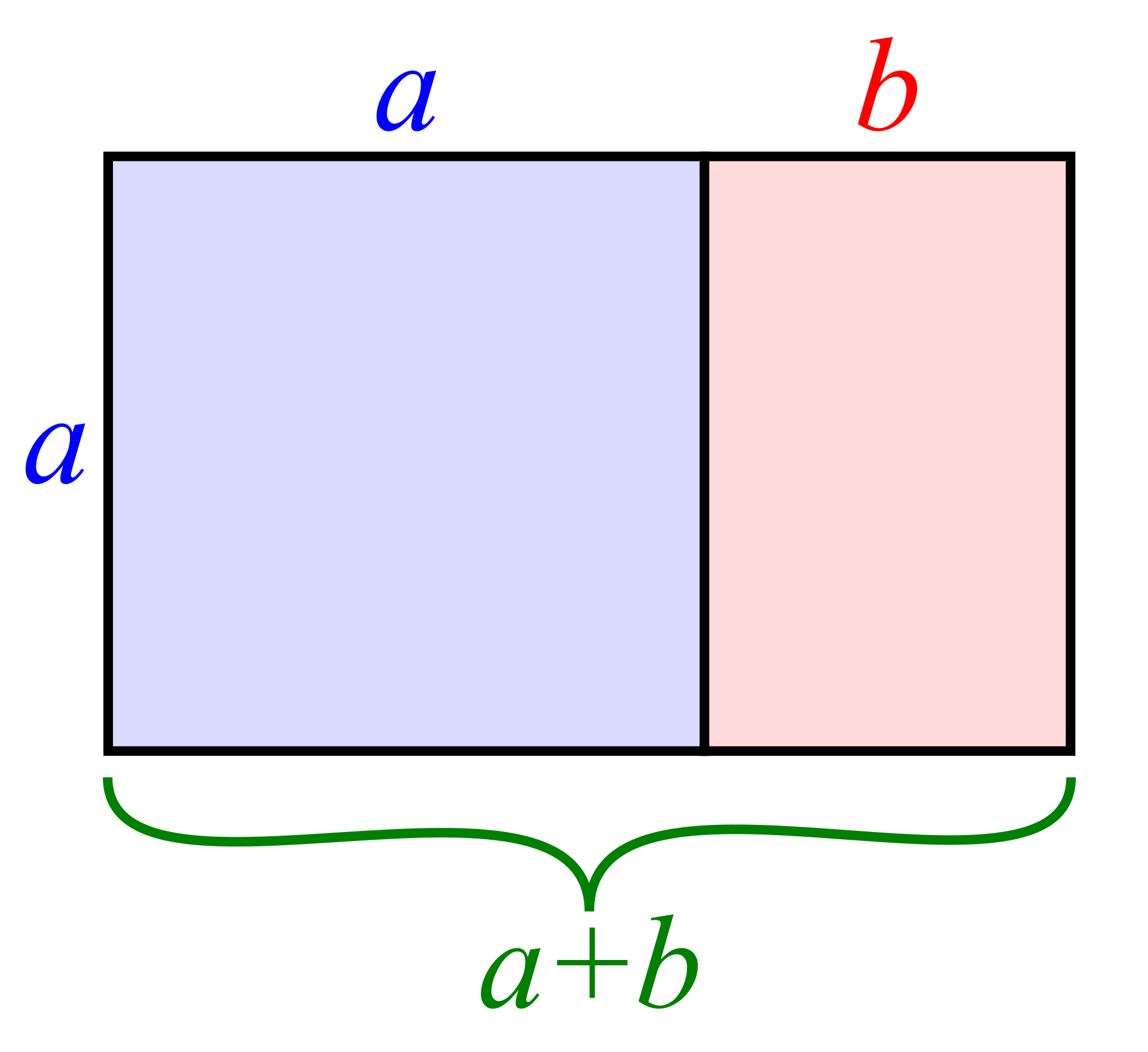
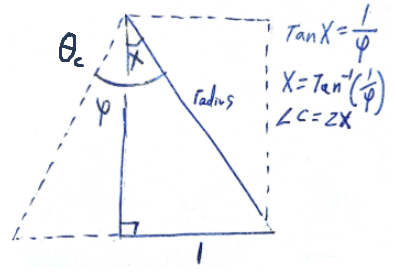
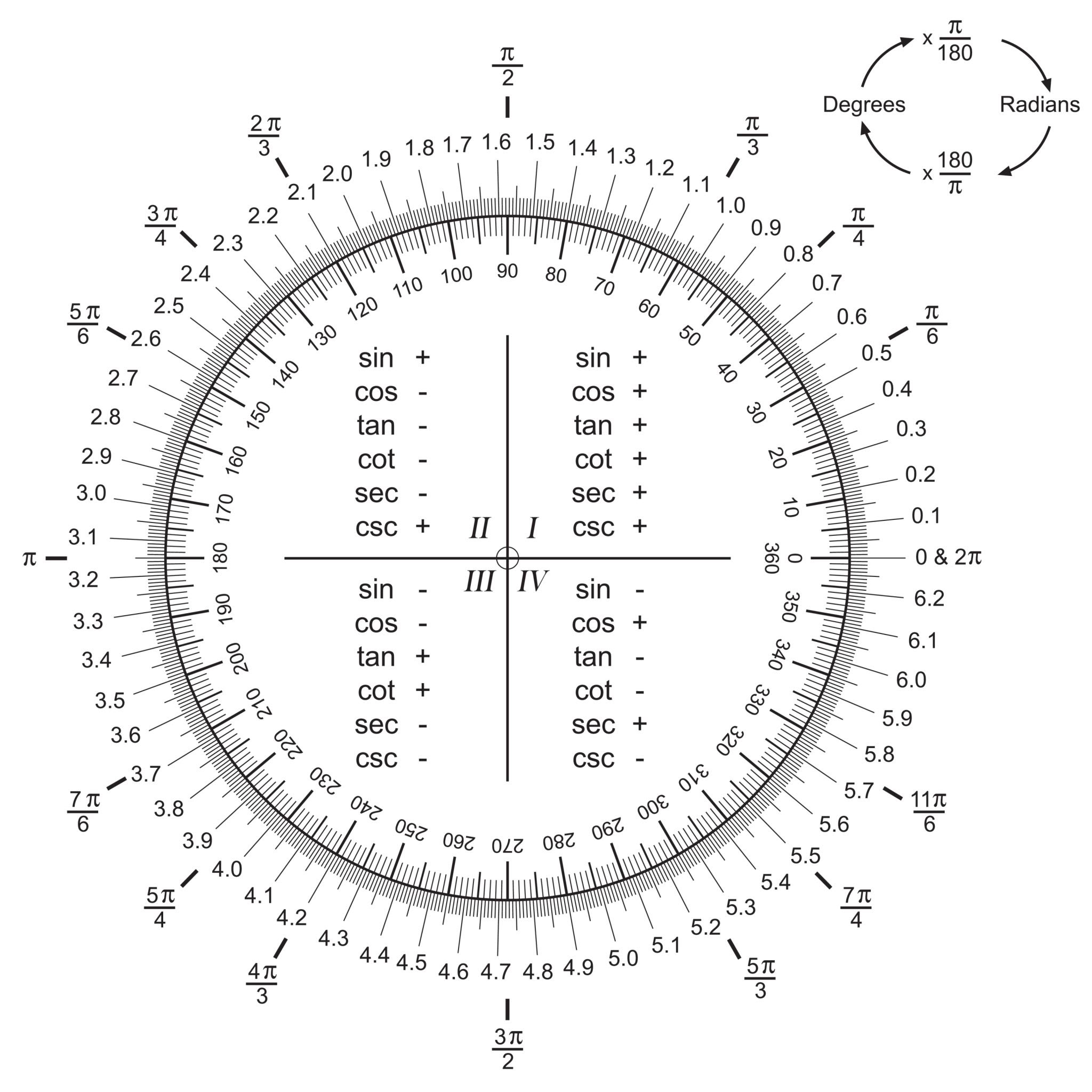
# Initial Setup:

Golden Ratio: (*a*+*b*)/*a* = φ = (1+√5)/2



A regular icosahedron contains three golden rectangles set perpendicular to each other. The central angle of edge 𝛼 is then the central angle for every edge of a regular icosahedron. This angle, 𝛳c = atan(1/φ) \* 2 = 2 \* atan((√5-1)/2).  
  
𝛳c will be the most important angle value for the purpose of this software. Ten of the twelve initial points on the icosahedron will have a latitude of ∓𝛳c. The centroid distance between each of these points to their closest neighbors will also be c . Every point calculated from here on out will also be a divided derivative of this value.  
  
Subdividing the edges into halves is the next step.  
The formulas are as follows (using Excel format)  
  
Bx=cos(Lat2)\*cos(Long2-Long1)  
By=cos(Lat2)\*sin(Long2-Long1)

Lat.mid=atan2(sqrt((cos(Lat1)+Bx)^2+By^2), sin(Lat1)+sin(Lat2))  
Long.mid=atan2(cos(Lat1)+Bx, By)

Lat1 = Latitude of point 1 in radians

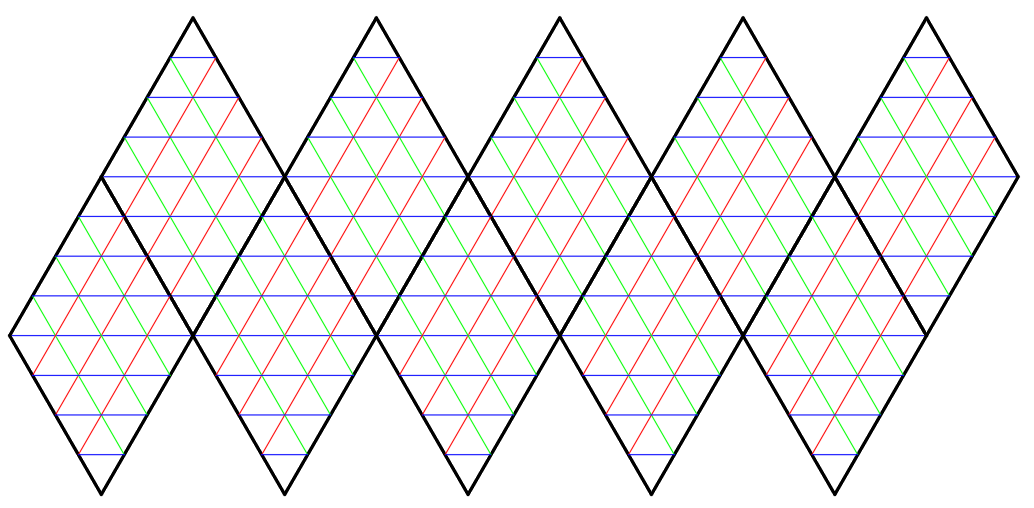
Lat2 = Latitude of point 2 in radians

Long1 = Longitude of point 1 in radians

Long2 = Longitude of point 2 in radians

Lat.mid & Long.mid = Lat/Long of calculated midpoint.

Note: There’s a difference with ATAN2 between standard notation and Excel or C++. Standard notation assumes ATAN2(x,y) but Excel and C++ use ATAN2(y,x). The equations above assume the latter.



# Subdivision:

I’m debating where in the tessellation process that geographic coordinates get converted into 3D cartesian coordinates. Generation speed is a factor and having the system generate both systems as each vertex is calculated might be faster but would make the memory used of the data array significantly larger before writing the file. At this point, waiting until the entire mesh is generated before conversion might be the better option.

VertexArray

EdgeArray

FaceArray\_current

FaceArray\_new

ScratchArray

The VertexArray will contain all the information on each of the existing vertices. This array will store the Vertex Index Number, the Latitude, the Longitude, and the Height of each vertex.

The “Vertex Index Number” (VIN) of each vertex is a numerical count of each vertex that gets incremented as each new vertex is generated.

Further iterations of this algorithm can also include normals, biome data, or whatever other information is generated by the planet generation code.

The EdgeArray will list every edge that has been subdivided so far. When subdividing a new edge, it will check the EdgeArray to see if the two parent vertices have already been subdivided. If so, it will skip the calculation but load the calculated vertex for face association into the ScratchArray.

Edge Array: Va,Vb;Vc

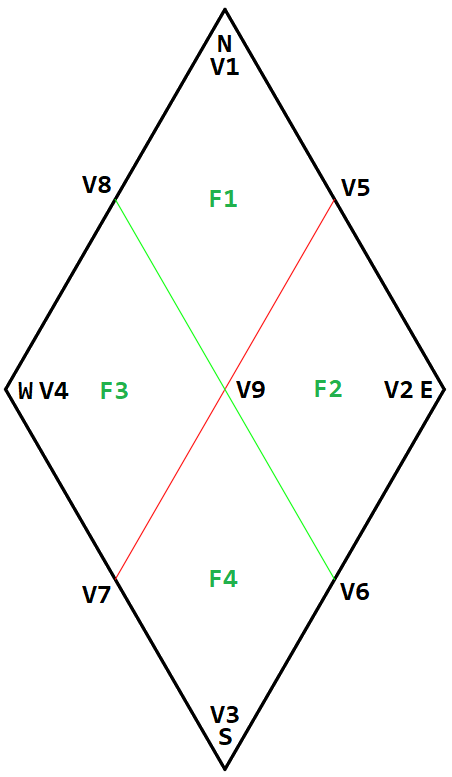
Va & Vb are an index comprised of the paired parent vertices derived from the Vertex Index Number of both. Vc is Vertex Index Number of the child vertex.

FaceArray\_current is a data array that lists each face yet to be subdivided by “Face Index Number” (FIN) and the four vertices that comprise the Face by VIN.

FaceArray\_new is a data array that lists each face that has been subdivided so far. New faces are given sequential index numbers as they are generated in this order: North, East, South, West. Given the slant of the rhombuses that the faces consist of, it’s essentially Left-to-Right, Top-to-Bottom following the five primary bands getting tessellated from North-to-South then East around the globe.

Both FaceArray\_current and FaceArray\_new will list the vertices of each face in a clock-wise manner with the Northern-most vertex as the first. (North, East, South, West)

ScratchArray is the data array that holds the vertex data of the face currently being subdivided. This includes the four parent vertex data and the five new vertices that will be calculated. The data in this array will be used to temporarily hold the vertex information to load FaceArray\_new with the required data.

The vertex order will be: The first four will be in the same order as the vertices of the parent face (V1, V2, V3, V4). V5 is the vertex between V1 & V2. V6 is the vertex between V2 & V3. V7 is the vertex between V3 & V4. V8 is the vertex between V4 & V1. V9 is the vertex between V2 & V4.  
  
Step 1:  
 Load face data with indexes for all 4 vertices from FaceArray\_current & VertexArray into ScratchArray.  
  
Step 2:  
 Sub-divide the 4 edges of the face. These will be V5, V6, V7, & V8. Add to EdgeArray & ScratchArray.  
  
Step 3:  
 A rhombus has two pairs of opposite points, one set is closer to each other than the other. Subdivide the inferred edge between the East vertex and the West vertex. (Because the face is a Rhombus, the distance between the East and West vertices is shorter than the North and South vertices.) This point is the center of the parent face that will be the common point to the four new faces.  
  
This point does not need to be added to the EdgeArray as there are no shared edges with the remaining faces that need to be subdivided.  
 Add vertex to ScratchArray as V9.

Step 4:

Add V5 through V9 from the ScratchArray to the VertexArray. Number them sequentially starting after the largest number in the vertex array.

Step 5:

Associate the vertices to the new faces in FaceArray\_new following the established method of the North point/face as the first and working left-to-right and top-to-bottom. F1-4 is shown for local use but FaceArray\_new will have them numbered sequentially as they are generated.

F1 = V1, V5, V9, V6

F2 = V5, V2, V9, V7

F3 = V6, V9, V3, V8

F4 = V9, V7, V8, V4

Step 6:

Purge ScratchArray

Step 7a:

Load next parent face and continue from Step 1.

Step 7c:

If all parent faces have been subdivided, do the following:

1. Check if the tessalation level has been met.
   1. If true goto step 8.
   2. If false continue.
2. Purge FaceArray\_current
3. copy FaceArray\_new to FaceArray\_current
4. purge FaceArray\_new
5. Load first face from FaceArray\_current and continue from Step 1

Step 8:

Once desired tessellation level has been reached, convert all vertices in VertexArray from geographic coordinates to 3D Cartesian coordinates. (lat, long, & height to X, Y, & Z).

Use a new array, VertexArray\_cart to store converted data. Purge VertexArray once done.

The equations to do this are as follows:

X = r \* cos(lat) \* cos(long)

Y = r \* cos(lat) \* sin(long)

Z = r \* sin(lat)

Note: Software has a difficult time when angles are multiples of 90 degrees when Sin and Cos of those angles equals zero. Some manual exceptions, perhaps through rounding, might be required to have the software assume a value of zero if it’s “close enough”.

Step 9:

Use FaceArray\_current & VertexArray\_cart to generate the 3D model file.

Observation: When determining the central angle of a fixed chord length, I’ve discovered that this can be easily determined with the ratio of CHORD LENGTH / RADIUS = Central Angle in radians. I can use this to determine the number of voxels that run from the North Pole to the South Pole given the length of a voxel. Since the voxels don’t run North-South for half the distance, I can determine the number of voxel that run along the edge of an initial triangle and multiply by 4.

Note: There will be plenty of Trig functions used during this process. In C++ and other programming languages, there’s an issue about floating-point rounding errors in regard to this. As the solution to Trig functions approach either 1 or 0 (usually when the angle approaches a multiple of 90 degrees), it might not return an exact value for 1 or 0 but have a slight rounding error because of how floating-point variables are calculated. The primary method I’ve found to combat this is to introduce an “Epsilon Comparison” to the Trig equation. Instead of checking for exact equality (e.g. if (x == 1)), compare floating-point values within a small tolerance (epsilon).

Ex: if (fabs(x – 1.0) < 1e-6).

Example Epsilon Comparison:

*// C++*

#include <cmath>

#include <iostream>

int main() {

double angle = 0.0;

double sinValue = sin(angle);

*// Instead of:*

*// if (sinValue == 0) { ... }*

*// use epsilon comparison:*

const double epsilon = 1e-6; *// A small tolerance value*

if (fabs(sinValue) < epsilon) {

std::cout << “sin( “ << angle << “) is approximately zero.” << std::endl;

} else {

std::cout << “sin( “ << angle << “) is not approximately zero.” << std::endl;

}

return 0;

}