

```

::::::::::::
Angles.hpp
::::::::::::
# include <iostream>
# include <iomanip>
# include <fstream>
// # include <cstdlib>
// # include <cmath>
// # include <vector>
// # include <algorithm>

using namespace std;

//-----
// Purpose:
//
// Class Angles Interface Files
//
// Discussion:
//
// Licensing:
//
// This code is distributed under the GNU LGPL license.
//
// Modified:
//
// 2012.05.02
//
// Author:
//
// Young Won Lim
//
// Parameters:
//
//-----

double compute_angle ( int idx, int nIter );
void draw_angle_tree (int nIter, int nAngle);

class Angles
{
public:
Angles();
Angles(double *A, int nIter, int nAngle);

void setA(double *A);
void setNIter(int nIter);
void setNAngle(int nAngle);
int getNIter();
int getNAngle();

void plot_unit_circle_angle ();
void plot_line_angle ();
void plot_residual_errors ();
void calc_statistics ();
void plot_uniform_scale_residual_errors ();

private:

double *A;
int nIter;
int nAngle;
int Leaf;

double delta_avg;

```

```
double delta_std;
double min_angle;
double max_angle;
```

```
};
```

```
:::::::::::::
```

```
Angles.cpp
```

```
:::::::::::::
```

```
# include <iostream>
# include <iomanip>
# include <cstdlib>
# include <cmath>
# include <fstream>
# include <vector>
# include <algorithm>
```

```
using namespace std;
```

```
# include "Angles.hpp"
# include "cordic.hpp"
```

```
double pi = 3.141592653589793;
double K = 1.646760258121;
```

```
//-----
// Purpose:
//
// Class Angles Implementation Files
//
// Discussion:
//
//
// Licensing:
//
// This code is distributed under the GNU LGPL license.
//
// Modified:
//
// 2012.05.02
//
// Author:
//
// Young Won Lim
//
// Parameters:
//
//-----
```

```
//-----
// Compute Angles based on the binary tree
// idx - index for leaf nodes of the binary tree
// nIter - no of iteration (corresponds to the level of the tree)
//-----
```

```
double compute_angle ( int idx, int nIter )
{
    double angle = 0.0;
    char s[32];
    int i, j;
```

```
    // i - bit position starting from lsb
    // j = 2^i
    // (idx & (1 << i)) - i-th bit of idx
    // if each bit is '1', add atan(1/2^i)
    // if each bit is '0', sub atan(1/2^i)
```

```

// s[32] contains the binary representation of idx
for (i=0; i<nIter; i++) {
    j = 1 << i;
    if (idx & (1 << i)) {
        angle += atan( 1. / j );
        s[nIter-i-1] = '1';
    } else {
        angle -= atan( 1. / j );
        s[nIter-i-1] = '0';
    }

    // cout << "i=" << i << " j=" << j << " 1/j=" << 1./j
    //      << " atan(1/j)=" << atan(1./j)*180/3.1416 << endl;

}
s[nIter] = '\0';

// cout << nIter << " " << idx << " " << s
//      << " ----> " << angle*180/3.1416 << endl;

return angle;
}

//-----
// Draw Angle Tree
//-----
void draw_angle_tree (int nIter, int nAngle)
{
    int level = nIter;
    int i, j, k;
    ofstream myout;
    double *A;

    if (nIter > 10) {
        cout << "nIter = " << nIter << " is too large to plot! " << endl;
        return;
    }

    // cout << "nIter = " << nIter << endl;
    // cout << "nAngle = " << nAngle << endl;

    A = (double *) malloc(nAngle * sizeof (double));

    myout.open("angle.dat");

    for (i=0; i<level; ++i) {
        nIter = i;
        nAngle = 1 << nIter;

        for (j=0; j<nAngle; ++j) {
            A[j] = compute_angle(j, nIter);

            // cout << "A[" << j << "] = " << A[j] << endl;
            myout << A[j]*180/pi << " " << 0.5*i << " 0.0 0.5" << endl;
        }
    }

    myout.close();

    // writing gnuplot commands
    myout.open("command.gp");

```

```

myout << "plot 'angle.dat' using 1:2:3:4 ";
myout << "with vectors head filled lt 2" << endl;
myout << "pause mouse keypress" << endl;
myout.close();

```

```

system("gnuplot command.gp");

```

```

//-----
// Accumulated Angle Tree
//-----

```

```

myout.open("angle.dat");

```

```

for (i=0; i<level; ++i) {
    for (k=0; k<=i; k++) {
        nIter = k;
        nAngle = 1 << nIter;

        for (j=0; j<nAngle; ++j) {
            A[j] = compute_angle(j, nIter);

            //cout << "A[" << j << "] = " << A[j] << endl;
            myout << A[j]*180/pi << " " << 0.5*i << " 0.0 0.5" << endl;
        }
    }
}

```

```

myout.close();

```

```

// writing gnuplot commands
myout.open("command.gp");
myout << "plot 'angle.dat' using 1:2:3:4 ";
myout << "with vectors head filled lt 2" << endl;
myout << "pause mouse keypress" << endl;
myout.close();

```

```

system("gnuplot command.gp");

```

```

free (A);
return;

```

```

}

```

```

//-----
// Class Angles' Member Functions
//-----

```

```

Angles::Angles() : A(NULL), nIter(3), nAngle(8)
{
    Leaf = 1;

    cout << "A is not initialized " << endl;
    cout << "nIter = " << nIter << endl;
    cout << "nAngle = " << nAngle << endl;
}

```

```

Angles::Angles(double *A, int nIter, int nAngle) :
    A(A), nIter(nIter), nAngle(nAngle)
{
    if (nAngle == (1 << nIter)) {
        Leaf = 1;
    }
}

```

```

    cout << "A LeafAngles Object is created" << endl;
} else {
    Leaf = 0;
    cout << "An AllAngles Object is created" << endl;
}

cout << "nIter = " << nIter << endl;
cout << "nAngle = " << nAngle << endl;
}

void Angles::setNIter(int nIter)
{
    nIter = nIter;
}

void Angles::setNAngle(int nAngle)
{
    nAngle = nAngle;
}

int Angles::getNIter()
{
    return nIter;
}

int Angles::getNAngle()
{
    return nAngle;
}

//-----
//      Plot angle vectors on the unit circle
//-----
void Angles::plot_unit_circle_angle ()
{
    int i;
    ofstream myout;

    cout << "* plot_unit_circle_angle ... " ;
    if (Leaf) cout << "(LeafAngles)" << endl;
    else cout << "(AllAngles)" << endl;

    if (nIter > 10) {
        cout << "nIter = " << nIter << " is too large to plot! " << endl;
        return;
    }

    // writing angle data on a unit circle
    myout.open("angle.dat");
    for (i=0; i<nAngle; i++) {
        myout << "0.0 0.0 " << cos(A[i]) << " " << sin(A[i]) << " " << endl;
    }
    myout.close();

    // writing gnuplot commands
    myout.open("command.gp");
    myout << "set size square" << endl;
    myout << "set xrange [-1:+1]" << endl;
    myout << "set yrange [-1:+1]" << endl;
    myout << "set object 1 circle at 0, 0 radius 1" << endl;
    myout << "plot 'angle.dat' using 1:2:3:4 ";
    myout << "with vectors head filled lt 2" << endl;
    myout << "pause mouse keypress" << endl;
}

```

```

myout.close();

system("gnuplot command.gp");

return;
}

//-----
//   Plot angle vectors on the line
//-----
void Angles::plot_line_angle ()
{
    ofstream myout;

    cout << "* plot_line_angle ... ";
    if (Leaf) cout << "(LeafAngles)" << endl;
    else cout << "(AllAngles)" << endl;

    if (nIter > 10) {
        cout << "nIter = " << nIter << " is too large to plot! " << endl;
        return;
    }

    // cout << "nIter = " << nIter << endl;
    // cout << "nAngle = " << nAngle << endl;

    myout.open("angle.dat");

    for (int i=0; i<nAngle; ++i) {
        // cout << "A[" << i << "] = " << A[i] << endl;
        myout << A[i] << " 0.0 0.0 0.5" << endl;
    }

    myout.close();

    // writing gnuplot commands
    myout.open("command.gp");
    myout << "set yrange [0:+2]" << endl;
    myout << "plot 'angle.dat' using 1:2:3:4 ";
    myout << "with vectors head filled lt 2" << endl;
    myout << "pause mouse keypress" << endl;
    myout.close();

    system("gnuplot command.gp");

    return;
}

//-----
//   plot residual errors
//-----
void Angles::plot_residual_errors ()
{
    int i;
    double x, y, z;
    ofstream myout;

    cout << "* plot_residual_errors ... ";
    if (Leaf) cout << "(LeafAngles)" << endl;
    else cout << "(AllAngles)" << endl;
}

```

```

if (nIter > 10) {
    cout << "nIter = " << nIter << " is too large to plot! " << endl;
    return;
}

// writing residue errors
myout.open("angle.dat");

double mse = 0.0, max_se = 0.0, se = 0.0;
for (i=0; i<nAngle; i++) {
    x = 1 / K;
    y = 0.0;
    z = A[i];

    cordic(&x, &y, &z, nIter);

    se = z * z;
    mse += se;
    if (se > max_se) max_se = se;

    // cout << "A[" << i << "]=" ;
    // cout << fixed << right << setw(10) << setprecision(7) << A[i];
    // cout << " z= " ;
    // cout << fixed << right << setw(10) << setprecision(7) << z << endl;

    myout << fixed << right << setw(10) << i;
    myout << fixed << right << setw(12) << setprecision(7) << A[i];
    myout << fixed << right << setw(12) << setprecision(7) << z << endl;
}

// mse /= nAngle;
// mse = sqrt(mse);

cout << "* No of points = " ;
cout << fixed << right << setw(10) << nAngle << endl;
cout << "* Mean Squared Residual Errors = " ;
cout << fixed << right << setw(12) << setprecision(7) << mse << endl;
cout << "* Max Squared Residual Error (Uniform Scale) = " ;
cout << fixed << right << setw(12) << setprecision(7) << max_se << endl;

myout.close();

// writing gnuplot commands
myout.open("command.gp");
myout << "set autoscale y" << endl;
myout << "plot 'angle.dat' using 1:3 with linespoints " << endl;
myout << "pause mouse keypress" << endl;
myout.close();

system("gnuplot command.gp");

return;
}

//-----
// Calculate leaf node angle statistics
//-----
void Angles::calc_statistics ()
{
    vector <double> B, D;
    vector <double> ::iterator first, last;
    double mean, std;
    ofstream myout;

```

```

cout << "* calc_statistics... ";
if (Leaf) cout << "(LeafAngles)" << endl;
else cout << "(AllAngles)" << endl;

cout << "nAngle = " << nAngle << endl;

for (int i=0; i < nAngle; ++i) {
    // cout << "A[" << i << "]= " << setw(12) << setprecision(8) << A[i] << endl;
    // cout << "B[" << i << "]= " << setw(12) << setprecision(8) << B[i] << endl;
}

// B : sorted angles array
for (int i=0; i < nAngle; ++i)
    B.push_back(A[i]);

sort(B.begin(), B.end());

// D : difference angle array
for (int i=0; i < nAngle-1; ++i)
    D.push_back(B[i+1]- B[i]);

sort(D.begin(), D.end());

mean = 0.0;
for (int i=0; i < D.size(); ++i)
    mean += D[i];
mean /= D.size();

std = 0.0;
for (int i=0; i < D.size(); ++i)
    std += ((D[i]-mean) * (D[i]-mean));
std /= D.size();
std = sqrt(std);

min_angle = B[0];
max_angle = B[B.size()-1];
delta_avg = mean;
delta_std = std;

cout << "max angle      = " << B[0] << endl;
cout << "min angle      = " << B[B.size()-1] << endl;
cout << "delta computed = " << (B[B.size()-1] - B[0]) / nAngle ;
cout << " = (max-min) / nAngle " << endl;
cout << "delta mean = " << mean << endl;
cout << "delta std = " << std << endl;

// write histogram data from delta array
myout.open("angle.dat");
double pb ;
for (int i=0, j, k; i<nAngle-2; i++) {
    j = i; k = 1;
    while ((D[j+1] - D[j])/D[j] < 0.01) {
        k++;
        j++;
    }
    pb = (double) k / D.size();
    myout << fixed << right << setw(12) << setprecision(7) << D[i] ;
    myout << " " << pb << endl;
    i = j;
}
myout.close();

// writing gnuplot commands

```



```

myout.open("command.gp");
myout << "set boxwidth 0.9 relative" << endl;
myout << "set style fill solid 1.0 border lt -1" << endl;
myout << "plot 'angle.dat' with linespoints" << endl;
myout << "pause mouse keypress" << endl;
myout.close();

system("gnuplot command.gp");

// write angle-delta data
myout.open("angle.dat");
for (int i=0; i<B.size()-1; i++) {
    myout << B[i] << " " << B[i+1] - B[i] << endl;
}
myout.close();

// writing gnuplot commands
myout.open("command.gp");
myout << "set boxwidth 0.9 relative" << endl;
myout << "set style fill solid 1.0 border lt -1" << endl;
myout << "plot 'angle.dat' with linespoints" << endl;
myout << "pause mouse keypress" << endl;
myout.close();

system("gnuplot command.gp");

return;
}

//-----
// Calculate leaf node angle statistics
//-----
void Angles::plot_uniform_scale_residual_errors ()
{
    int i;
    double x, y, z;
    ofstream myout;

    cout << "* plot_residual_errors ... ";
    if (Leaf) cout << "(LeafAngles)" << endl;
    else cout << "(AllAngles)" << endl;

    if (nIter > 10) {
        cout << "nIter = " << nIter << " is too large to plot! " << endl;
        return;
    }

    // writing residue errors
    myout.open("angle.dat");

    double ang = min_angle + delta_avg*0.5;
    double mse = 0.0, max_se = 0.0, se = 0.0 ;
    i = 0;
    while (ang < max_angle) {
        x = 1 / K;
        y = 0.0;
        z = ang;

        cordic(&x, &y, &z, nIter);

```

```

// cout << fixed << right << setw(10) << setprecision(7) << A[i];
// cout << fixed << right << setw(10) << setprecision(7) << z << endl;

myout << fixed << right << setw(10) << i;
myout << fixed << right << setw(12) << setprecision(7) << ang;
myout << fixed << right << setw(12) << setprecision(7) << z << endl;

se = z* z;
mse += se;
if (se > max_se) max_se = se;

ang += (delta_avg);
i++;

}

// mse /= i;
// mse = sqrt(mse);

cout << "* No of points (Uniform Scale) = " ;
cout << fixed << right << setw(10) << i << endl;
cout << "* Mean Squared Residual Errors (Uniform Scale) = " ;
cout << fixed << right << setw(12) << setprecision(7) << mse << endl;
cout << "* Max Squared Residual Error (Uniform Scale) = " ;
cout << fixed << right << setw(12) << setprecision(7) << max_se << endl;

myout.close();

// writing gnuplot commands
myout.open("command.gp");
myout << "set autoscale y" << endl;
myout << "plot 'angle.dat' using 1:3 with linespoints " << endl;
myout << "pause mouse keypress" << endl;
myout.close();

system("gnuplot command.gp");

return;

}

/*****

for (i=0; i<20; i+=4) {
    for (j=0; j<4; ++j) {
        r = atan( 1. / (1 << (i+j)) ) / atan( 1. / (1 << i) ) * 100;
        cout << "index = " << i+j << " --> r = " << r << endl;
    }
}

return 0;

}

*****/

:~::~:
Angles_tb.cpp
:~::~:

```

```

# include <cstdlib>
# include <cmath>
# include <iostream>
# include <iomanip>
# include <fstream>

using namespace std;

# include "cordic.hpp"
# include "Angles.hpp"

//-----
// Purpose:
//
// Explore Angles Space using Class Angles
//
// Discussion:
//
//
// Licensing:
//
// This code is distributed under the GNU LGPL license.
//
// Modified:
//
// 2012.05.02
//
// Author:
//
// Young Won Lim
//
// Parameters:
//
//-----

int main (int argc, char * argv[])
{
    int    nIter = 3;
    int    nAngle = 1 << nIter;
    int    i, j, k;
    int    level, leaves;
    double *A, *All;

    if (argc > 1 ) {
        nIter = atoi(argv[1]);
        nAngle = 1 << nIter;
    }

    // cout << "nIter = " << nIter << endl;
    // cout << "nAngle = " << nAngle << endl;

    A = (double *) malloc ((1<<nIter) * sizeof (double));
    All = (double *) malloc (2* (1<<nIter) * sizeof (double));

    for (j=0; j<nAngle; ++j) {
        A[j] = compute_angle(j, nIter);
        // cout << "A[" << j << "]= " << setw(12) << setprecision(8) << A[j] << endl;
    }

    for (i=0, k=0; i<=nIter; ++i) {
        level = i;
        leaves = 1 << level;

```

```

// cout << "level = " << level << "leaves = " << leaves << endl;
for (j=0; j<leaves; ++j) {
    All[j+k] = compute_angle(j, level);
    // cout << "All[" << j+k << "] = " << All[j+k] << endl;
}
k += leaves;
}

```

```

Angles LeafAngles(A, nIter, nAngle);
Angles AllAngles(All, nIter, 2*nAngle-1);

```

```

// -----
// Plot angle vectors on the unit circle
// -----
//LeafAngles.plot_unit_circle_angle();
//AllAngles.plot_unit_circle_angle();

```

```

// -----
// Plot angle on the line axis
// -----
// LeafAngles.plot_line_angle();
// AllAngles.plot_line_angle();

```

```

// -----
// Plot residue errors at the leaf node angles
// -----
LeafAngles.plot_residual_errors();
AllAngles.plot_residual_errors();

```

```

// -----
// Find Angles Statistics
// -----
LeafAngles.calc_statistics();
AllAngles.calc_statistics();

```

```

// -----
// Plot residue errors at the leaf node angles
// -----
LeafAngles.plot_uniform_scale_residual_errors();
AllAngles.plot_uniform_scale_residual_errors();

```

```

// -----
// Plot residue errors at the leaf node angles
// -----
//draw_angle_tree (nIter, nAngle);

```

```

return 0;

```

```

}

```