```
::::::::::::::
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:
//
//    2012.05.11
//
//  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 );

//****************************************************************************
//  Draw Binary Angle Tree and Cumulative Binary Angle Tree
//****************************************************************************
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);
  void    setThreshold(double threshold);

  int     getNIter();
  int     getNAngle();
  double  getThreshold();


  //****************************************************************************
  //   Plot angle vectors on a unit circle
```

```
  //-------------------------------------------------------------------------
  void plot_circle_angle ();

  //-------------------------------------------------------------------------
  //   Plot angle vectors on a linear scal
  //-------------------------------------------------------------------------
  void plot_line_angle ();

  //-------------------------------------------------------------------------
  //   plot residual errors
  //   Residuals-Angle Plot and Residuals-Index Plot
  //-------------------------------------------------------------------------
  void plot_residual_errors ();

  // ------------------------------------------------
  //   Find Angles Statistics  --> member data
  //   Delta Distribution Plot and  Angle-Delta Plot
  // ------------------------------------------------
  void calc_statistics ();

  void calc_uscale_statistics (double, double);
  void plot_uscale_statistics ();
  void plot_uscale_residual_errors ();
  void plot_uscale_residual_errors (double, double);


private:

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

  double  delta_avg;
  double  delta_std;
  double  min_angle;
  double  max_angle;

  double  ssr;      // sum of the squares of the residuals
  double  mse;      // mean squared error
  double  rms;      // root mean square error
  double  max_err;  // maximum of squared errors

  double threshold;

};


::::::::::::::
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:
//
//    2012.05.11
//
//  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 Binary Angle Tree and Cumulative Binary 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));


  //---------------------------------------
  // Binary Angle Tree
  //---------------------------------------

  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 << " " <<  i << " 0.0 1.0" << endl;
    }
  }

  myout.close();


  // writing gnuplot commands
  myout.open("command.gp");
  myout << "set title \"Binary Angle Tree\" " << endl;
  myout << "set xlabel \"Angles in degree\" " << endl;
  myout << "set ylabel \"Levels \" " << endl;
  myout << "set format x \"%.0f\" " << endl;
  myout << "set format y \"%.0f\" " << endl;
  myout << "plot 'angle.dat' using 1:2:3:4  ";
  myout << "with vectors head filled lt 3" << endl;
  myout << "pause mouse keypress" << endl;
  myout.close();


  system("gnuplot command.gp");

  //---------------------------------------
  // Cumulative 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 << " " <<  i << " 0.0 1.0" << endl;
      }
    }

  }

  myout.close();


  // writing gnuplot commands
  myout.open("command.gp");
  myout << "set title \"Cumulative Binary Angle Tree\" " << endl;
  myout << "set xlabel \"Angles in degree\" " << endl;
  myout << "set ylabel \"Levels \" " << endl;
  myout << "set format x \"%.0f\" " << endl;
  myout << "set format y \"%.0f\" " << endl;
  myout << "plot 'angle.dat' using 1:2:3:4  ";
  myout << "with vectors head filled lt 4" << 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;

  delta_avg = delta_std = min_angle = max_angle = 0.0;
  ssr = mse = rms = max_err = 0.0;
}


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

  cout << "(nIter = " << nIter << ", ";
  cout << "nAngle = " << nAngle << ")" <<endl;

  delta_avg = delta_std = min_angle = max_angle = 0.0;
  ssr = mse = rms = max_err = 0.0;

  threshold = 0.0;
```

```
}


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


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

void Angles::setThreshold(double threshold)
{
  threshold = threshold;
}

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


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


double Angles::getThreshold()
{
  return threshold;
}



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

  cout << "* plot_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");
  if (Leaf) myout << "set title \"Leaf Angles on a unit circle \" " << endl;
  else myout << "set title \"All Angles on a unit circle \" " << endl;
  myout << "set xlabel \"x\" " << endl;
```

```
  myout << "set ylabel \"y\" " << endl;
  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 3" << endl;
  myout << "pause mouse keypress" << endl;
  myout.close();


  system("gnuplot command.gp");

  return;

}


//-----------------------------------------------------------------------------
//     Plot angle vectors on a linear scale
//-----------------------------------------------------------------------------
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 1.0" << endl;
  }

  myout.close();


  // writing gnuplot commands
  myout.open("command.gp");
  if (Leaf) myout << "set title \"Leaf Angles on a linear scale\" " << endl;
  else      myout << "set title \"All Angles on a linear scale\" " << endl;
  myout << "set xlabel \"angles in radian\" " << endl;
  myout << "set ylabel \"\" " << endl;
  myout << "set yrange [0:+2]" << endl;
  myout << "plot 'angle.dat' using 1:2:3:4  ";
  myout << "with vectors head filled lt 3" << endl;
  myout << "pause mouse keypress" << endl;
  myout.close();


  system("gnuplot command.gp");

  return;

}


//-----------------------------------------------------------------------------
//   plot residual errors
```

```
//   Residuals-Angle Plot and Residuals-Index Plot
//-------------------------------------------------------------------------
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;
  }


  // B : sorted angles array
  vector <double> B;
  vector <double> ::iterator first, last;

  for (int i=0; i < nAngle; ++i)
    B.push_back(A[i]);

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


  // I=0 Use A[i] for Residuals-Index Plot
  // I=1 Use b[i] for Residuals-Angle Plot
  //.............................................................
  for (int I=0; I<2; I++) {
  //.............................................................

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

  int nBreak =0;

  // not member but local variables
  double se, ssr, mse, rms, max_err;
  se = ssr = mse = rms = max_err = 0.0;

  for (i=0; i<nAngle; i++) {
    x = 1 / K;
    y = 0.0;
    if (I == 0) z = A[i];
    else        z = B[i];

    cordic(&x, &y, &z, nIter, &nBreak, i, threshold);

    se = z * z;
    ssr += se;
    if (se > max_err) max_err = 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);
    if (I==0) myout <<  A[i];
    else      myout <<  B[i];
    myout << fixed << right << setw(12) << setprecision(7) << z << endl;

  }
```

```
  mse = ssr / nAngle;
  rms = sqrt(mse);

  max_err = sqrt(max_err);


  cout << "  No of points = " << nAngle ;
  cout << " (nBreak = " << nBreak << " : " ;
  cout <<  100. * nBreak / nAngle << " % )" << endl;

  cout << "  SSR: Sum of Squared Residuals    = " ;
  cout << fixed << right << setw(12) << setprecision(7) << ssr << endl;
  cout << "  MSR: Mean Squared Residuals      = " ;
  cout << fixed << right << setw(12) << setprecision(7) << mse << endl;
  cout << "  RMS: Root Mean Squared Residuals = " ;
  cout << fixed << right << setw(12) << setprecision(7) << rms << endl;
  cout << "  Max Residual Error               = " ;
  cout << fixed << right << setw(12) << setprecision(7) << max_err << endl;


  myout.close();


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

  if (I==0) {
    if (Leaf) myout << "set title \"Residual-Index Plot (Leaf) \" " << endl;
    else myout << "set title \"Residual-Index Plot (All) \" " << endl;
    myout << "set xlabel \"Index\" " << endl;
    myout << "set ylabel \"Residuals\" " << endl;
    myout << "plot 'angle.dat' using 1:3 with linespoints " << endl;
  } else {
    if (Leaf) myout << "set title \"Residual-Angle Plot (Leaf) \" " << endl;
    else myout << "set title \"Residual-Angle Plot of (All)\" " << endl;
    myout << "set xlabel \"Angles\" " << endl;
    myout << "set ylabel \"Residuals\" " << endl;
    myout << "plot 'angle.dat' using 2:3 with linespoints " << endl;
  }
  myout << "pause mouse keypress" << endl;

  myout.close();


  system("gnuplot command.gp");


  //.............................................................
  }
  //.............................................................

  return;

}


//---------------------------------------------------------------------------
//   Find Angles Statistics  --> member data
//   Delta Distribution Plot and  Angle-Delta Plot
//---------------------------------------------------------------------------
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)" << " nAngle = " << nAngle << endl;
else cout << "(AllAngles)" << " 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;

double udelta = (B[B.size()-1] - B[0]) /  nAngle; // computed unifrom delta

double min_diff = D[0];
double max_diff = D[D.size()-1];

cout << "  min angle      = " << min_angle << endl;
cout << "  max angle      = " << max_angle << endl;
cout << "  min difference = " << min_diff << endl;
cout << "  max difference = " << max_diff << endl;
cout << "  uniform delta  = " << udelta ;
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();


cout << "  + Delta Distribution Plot \n" ;

// writing gnuplot commands
myout.open("command.gp");
if (Leaf) myout << "set title \"Delta Distribution of Leaf Angles\" " << endl;
else      myout << "set title \"Delta Distribution of All Angles\" " << endl;
myout << "set xlabel \"Delta (Adjacent Angle Difference)\" " << endl;
myout << "set ylabel \"probability\" " << endl;
myout << "set yrange [0:+1]" << endl;

myout << "set arrow from " << delta_avg << ", 0";
myout << " to " << delta_avg << ", 0.7" << endl;
myout << "set label \"delta_avg \" at " << delta_avg;
myout << ", 0.7 right" << endl;

myout << "set arrow from " << udelta << ", 0";
myout << " to " << udelta << ", 0.5" << endl;
myout << "set label \"uniform delta \" at " << udelta;
myout << ", 0.5 right" << endl;

myout << "plot 'angle.dat' with linespoints" << endl;
myout << "pause mouse keypress" << endl;
myout.close();


system("gnuplot command.gp");



cout << "  + Angle-Delta Plot \n" ;

// 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");
if (Leaf) myout << "set title \"Angle-Delta Plot of Leaf Angles\" " << endl;
else      myout << "set title \"Angle-Delta Plot of All Angles\" " << endl;
myout << "set xlabel \"Angles in radian\" " << endl;
myout << "set ylabel \"Delta (Adj Angle Diff) \" " << endl;

myout << "set arrow from " << "-1.0, " << delta_avg;
myout << " to " << "+1.0, " << delta_avg << endl;
myout << "set label \"delta_avg \" at " << "+0.0, ";
myout << delta_avg << " left" << endl;

myout << "set arrow from " << "-1.0, " << udelta;
myout << " to " << "+1.0, " << udelta << endl;
myout << "set label \"uniform delta \" at " << "+1.0, ";
myout << udelta << " left" << endl;

myout << "set arrow from " << "-0.7853, " << min_diff;
myout << "0 to -0.7853, " << max_diff  << endl;
myout << "set label \"-pi/4 \" at " << "-0.7853, " ;
myout << min_diff << " right " << endl;

myout << "set arrow from " << "+0.7853, " << min_diff;
myout << "0 to +0.7853, " << max_diff  << endl;
myout << "set label \"+pi/4 \" at " << "+0.7853, " ;
myout << min_diff << " left " << endl;
```

```
  myout << "plot 'angle.dat' with linespoints" << endl;
  myout << "pause mouse keypress" << endl;
  myout.close();


  system("gnuplot command.gp");


  return;
}


//-----------------------------------------------------------------------------
//   Calculate statistics on the uniform scale
//-----------------------------------------------------------------------------
//   ssr     : sum of the squares of the residuals
//   mse     : mean squared error
//   rms     : root mean square error
//   max_err : maximum of squared errors
//-----------------------------------------------------------------------------
//   ssr     : sum of the squares of the residuals
//   ang =  min_angle + delta_avg * offFactor ;
//   ang += (delta_avg / resFactor);
//-----------------------------------------------------------------------------
void Angles::calc_uscale_statistics (double resFactor, double offFactor)
{

  int n;
  double x, y, z;
  ofstream myout;

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


  // sr      : square error of a data point

  double ang = min_angle + delta_avg * offFactor ;
  double se = 0.0 ;
  int nBreak =0;
  n = 0;


  ssr = mse = rms = max_err = 0.0;

  while (ang < max_angle) {
    x = 1 / K;
    y = 0.0;
    z = ang;


    cordic(&x, &y, &z, nIter, &nBreak, n, threshold);


    se = (z * z);
    ssr += se;
    if (se > max_err) max_err = se;

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

    ang += (delta_avg / resFactor);
    n++;

  }

  mse = ssr / n;
  rms = sqrt(mse);
```

```
  max_err = sqrt(max_err);

  cout << "  Angles = (" ;
  cout << min_angle << " : " << delta_avg << " : " << max_angle << ")" ;
  cout << endl << "  --> total " << n << " points" ;
  cout << " (nBreak = " << nBreak << " : " ;
  cout <<  100. * nBreak / n << " % )" << endl;

  cout << "  SSR: Sum of Squared Residuals    = " ;
  cout << fixed << right << setw(12) << setprecision(7) << ssr << endl;
  cout << "  MSR: Mean Squared Residuals      = " ;
  cout << fixed << right << setw(12) << setprecision(7) << mse << endl;
  cout << "  RMS: Root Mean Squared Residuals = " ;
  cout << fixed << right << setw(12) << setprecision(7) << rms << endl;
  cout << "  Max Residual Error               = " ;
  cout << fixed << right << setw(12) << setprecision(7) << max_err << endl;


  return;

}


//-------------------------------------------------------------------------------
//   Plot uniform scale statistics
//-------------------------------------------------------------------------------
//   ang =  min_angle + delta_avg * offFactor ;
//   ang += (delta_avg / resFactor);
//-------------------------------------------------------------------------------
void Angles::plot_uscale_statistics ( )
{

  int    i, j;
  double resFactor, offFactor;
  ofstream myout;

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

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

  // ssr : sum of the squares of the residuals
  // mse : mean squared error
  // rms : root mean square error
  // sr  : square error of a data point
  // max_err : maximum of squared errors


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

  for (i=0; i<4; i++) {
    resFactor = i + 1.0;

    myout << fixed << right << setw(10) << resFactor;

    for (j=0; j<4; j++) {
      offFactor = (j+1) / 4.;

      cout << " ============== " << i << " === " << j ;
      cout << " ============== " << endl;

      calc_uscale_statistics(resFactor, offFactor);

      myout << fixed << right << setw(22) << setprecision(7) << ssr;
    }
```

```
    myout << endl;

  }

  myout.close();



  // writing gnuplot commands
  myout.open("command.gp");
  myout << "set autoscale y" << endl;
  myout << "plot 'angle.dat' using 1:2 with linespoints,  " ;
  myout << " 'angle.dat' using 1:3 with linespoints,  " ;
  myout << " 'angle.dat' using 1:4 with linespoints,  " ;
  myout << " 'angle.dat' using 1:5 with linespoints " << endl;
  myout << "pause mouse keypress" << endl;
  myout.close();


  system("gnuplot command.gp");

  return;

}



//-------------------------------------------------------------------------
//   Plot residual errors on the uniform scale
//-------------------------------------------------------------------------
//   ang =  min_angle + delta_avg * offFactor ;
//   ang += (delta_avg / resFactor);
//-------------------------------------------------------------------------
void Angles::plot_uscale_residual_errors
(double resFactor, double offFactor )
{

  int n;
  double x, y, z;
  ofstream myout;

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

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

  double ang = min_angle + delta_avg * offFactor ;
  double se = 0.0 ;
  int nBreak =0;
  n = 0;

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

  while (ang < max_angle) {
    x = 1 / K;
    y = 0.0;
    z = ang;


    cordic(&x, &y, &z, nIter, &nBreak, n, threshold);


    se = (z * z);
```

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

    myout << fixed << right << setw(10) << n;
    myout << fixed << right << setw(22) << setprecision(7) << ang;
    myout << fixed << right << setw(22) << setprecision(7) << z;
    myout << fixed << right << setw(22) << setprecision(7) << se << endl;

    ang += (delta_avg / resFactor);
    n++;

  }

  cout << "  Angles = (" ;
  cout << min_angle << " : " << delta_avg << " : " << max_angle << ")" ;
  cout << endl << "  --> total " << n << " points" ;
  cout << " (nBreak = " << nBreak << " : " ;
  cout <<  100. * nBreak / n << " % )" << 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;

}



void Angles::plot_uscale_residual_errors ()
{

  plot_uscale_residual_errors (1.0, 1.0);

}


/*****


  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:
//
//    2012.05.11
//
//
//  Author:
//
//    Young Won Lim
//
//  Parameters:
//
//----------------------------------------------------------------------------



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

  // ---------------------------------------------------------------
  // nIter   : Number of Iteration = Height of binary angle tree
  // nAngle  : Number of Angles    = Number of Leaf Nodes
  // ---------------------------------------------------------------
  int    nIter = 3;
  int    nAngle = 1 << nIter;


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

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


  // ---------------------------------------------------------------
  // A   : contains the angles of leaf nodes in binary angle tree
  // All : contains the angles of all nodes in binary angle tree
  // ---------------------------------------------------------------
  double *A, *All;
  int    level, leaves;
  int    i, j, k;

  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;
}


// ------------------------------------------------------------
// LeafAngles : Angles Class for leaf nodes only
// AllAngles  : Angles Class for all nodes (internal nodes included)
// ------------------------------------------------------------
Angles LeafAngles(A, nIter, nAngle);
Angles AllAngles(All, nIter, 2*nAngle-1);


// ------------------------------------------------
//   Plot Binary Angle Tree
// ------------------------------------------------
// draw_angle_tree (nIter, nAngle);

// ------------------------------------------------
//   Plot angle vectors on a unit circle
// ------------------------------------------------
// LeafAngles.plot_circle_angle();
// AllAngles.plot_circle_angle();

// ------------------------------------------------
//   Plot angle vectors on a linear scale
// ------------------------------------------------
// LeafAngles.plot_line_angle();
// AllAngles.plot_line_angle();



LeafAngles.setThreshold(0.0);
AllAngles.setThreshold(0.0);


// ------------------------------------------------
//   Find Angles Statistics  --> member data
//   Delta Distribution Plot and  Angle-Delta Plot
// ------------------------------------------------
LeafAngles.calc_statistics();
AllAngles.calc_statistics();


// ------------------------------------------------
//   Calculate Uniform Scale Statistics --> member data
// ------------------------------------------------
LeafAngles.calc_uscale_statistics(1.0, 1.0);
AllAngles.calc_uscale_statistics(1.0, 1.0);



// ------------------------------------------------
//   plot residual errors
//   Residuals-Angle Plot and Residuals-Index Plot
// ------------------------------------------------
LeafAngles.plot_residual_errors();
```

```
  AllAngles.plot_residual_errors();


  return 0;



  // ------------------------------------------------
  //   Plot Uniform Scale Statistics
  // ------------------------------------------------
  LeafAngles.plot_uscale_statistics();
  AllAngles.plot_uscale_statistics();

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









  return 0;

}


:::::::::::::::
cordic.hpp
:::::::::::::::
void cordic ( double *x, double *y, double *z, int n, int *, int, double =0.0 );
:::::::::::::::
cordic.cpp
:::::::::::::::
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <cmath>
# include <ctime>

using namespace std;

# include "cordic.hpp"


//****************************************************************************80

void cordic ( double *x, double *y, double *z, int n,
              int *nBreak, int nBreakInit, double threshold)

//****************************************************************************80
//  CORDIC returns the sine and cosine using the CORDIC method.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2012.04.17
//
//  Author:
//
//    Based on MATLAB code in a Wikipedia article.
//
```

```
//    Modifications by John Burkardt
//
//    Further modified by Young W. Lim
//
//  Parameters:
//
//    Input:
//      *x: x coord of an init vector
//      *y: y coord of an init vector
//      *z: angle (-90 <= angle <= +90)
//       n: number of iteration
//         A value of 10 is low.  Good accuracy is achieved
//         with 20 or more iterations.
//
//    Output:
//      *xo: x coord of a final vector
//      *yo: y coord of a final vector
//      *zo: angle residue
//
//  Local Parameters:
//
//    Local, real ANGLES(60) = arctan ( (1/2)^(0:59) );
//
//    Local, real KPROD(33), KPROD(j) = product ( 0 <= i <= j ) K(i),
//    K(i) = 1 / sqrt ( 1 + (1/2)^(2i) ).
//
//
{
# define ANGLES_LENGTH 60
# define KPROD_LENGTH 33

  double angle;
  double angles[ANGLES_LENGTH] = {
    7.85398163397448309628E-01,
    4.63647609000806116214E-01,
    2.44978663126864154170E-01,
    1.24354994546761435030E-01,
    6.24188099959573484739E-02,
    3.12398334302682762540E-02,
    1.56237286204768308028E-02,
    7.81234106010111129650E-03,
    3.90623013196697182760E-03,
    1.95312251647881868510E-03,
    9.76562189559319430400E-04,
    4.88281211194898275470E-04,
    2.44140620149361764020E-04,
    1.22070311893670204240E-04,
    6.10351561742087750220E-05,
    3.05175781155260968620E-05,
    1.52587890613157621070E-05,
    7.62939453110197026340E-06,
    3.81469726560649628290E-06,
    1.90734863281018703540E-06,
    9.53674316405960879420E-07,
    4.76837158203088859930E-07,
    2.38418579101557982490E-07,
    1.19209289550780685310E-07,
    5.96046447753905544140E-08,
    2.98023223876953036770E-08,
    1.49011611938476551470E-08,
    7.45058059692382798710E-09,
    3.72529029846191404530E-09,
    1.86264514923095702910E-09,
    9.31322574615478515360E-10,
    4.65661287307739257780E-10,
    2.32830643653869628900E-10,
    1.16415321826934814450E-10,
    5.82076609134674072260E-11,
    2.91038304567337036130E-11,
    1.45519152283668518070E-11,
```

```
  7.2759576141834259033E-12,
  3.6379788070917129517E-12,
  1.8189894035458564758E-12,
  9.0949470177292823792E-13,
  4.5474735088646411896E-13,
  2.2737367544323205948E-13,
  1.1368683772161602974E-13,
  5.6843418860808014870E-14,
  2.8421709430404007435E-14,
  1.4210854715202003717E-14,
  7.1054273576010018587E-15,
  3.5527136788005009294E-15,
  1.7763568394002504647E-15,
  8.8817841970012523234E-16,
  4.4408920985006261617E-16,
  2.2204460492503130808E-16,
  1.1102230246251565404E-16,
  5.5511151231257827021E-17,
  2.7755575615628913511E-17,
  1.3877787807814456755E-17,
  6.9388939039072283776E-18,
  3.4694469519536141888E-18,
  1.7347234759768070944E-18 };
double c2;
double factor;
int j;
double kprod[KPROD_LENGTH] = {
  0.70710678118654752440,
  0.63245553203367586640,
  0.61357199107789634961,
  0.60883391251775242102,
  0.60764825625616820093,
  0.60735177014129595905,
  0.60727764409352599905,
  0.60725911229889273006,
  0.60725447933256232972,
  0.60725332108987516334,
  0.60725303152913433540,
  0.60725295913894481363,
  0.60725294104139716351,
  0.60725293651701023413,
  0.60725293538591350073,
  0.60725293510313931731,
  0.60725293503244577146,
  0.60725293501477238499,
  0.60725293501035403837,
  0.60725293500924945172,
  0.60725293500897330506,
  0.60725293500890426839,
  0.60725293500888700922,
  0.60725293500888269443,
  0.60725293500888161574,
  0.60725293500888134606,
  0.60725293500888127864,
  0.60725293500888126179,
  0.60725293500888125757,
  0.60725293500888125652,
  0.60725293500888125626,
  0.60725293500888125619,
  0.60725293500888125617 };
double pi = 3.141592653589793;
double poweroftwo;
double s2;
double sigma;
double sign_factor;
double theta;

double xn, yn;
```

```
//
//  Initialize loop variables:
//
  theta = *z;

  xn = *x;
  yn = *y;

  poweroftwo = 1.0;
  angle = angles[0];

  angle = atan( 1. );

//
//  Iterations
//

  for ( j = 1; j <= n; j++ )
  {
    if ( theta < 0.0 )
    {
      sigma = -1.0;
    }
    else
    {
      sigma = 1.0;
    }

    //
    // scaling factor is not yet corrected
    //
    static int cntBreak = 0;
    if (nBreakInit == 0) cntBreak = 0;
    if (abs(*z) < threshold) {
      *nBreak = ++cntBreak;

      // cout << "cntBreak= " << cntBreak;
      // cout << "  z= " << *z;
      // cout << " < " << threshold << endl;
      break;
    }

    factor = sigma * poweroftwo;

    *x =          xn - factor * yn;
    *y = factor * xn +          yn;

    xn = *x;
    yn = *y;

//
//  Update the remaining angle.
//
    theta = theta - sigma * angle;

    poweroftwo = poweroftwo / 2.0;
//
//  Update the angle from table, or eventually by just dividing by two.
//
    if ( ANGLES_LENGTH < j + 1 )
    {
      angle = angle / 2.0;
    }
    else
    {
      angle = angles[j];
    }

    angle = atan( 1. / (1 << j));
```

```
    *z = theta;
  }
//
//  Adjust length of output vector to be [cos(beta), sin(beta)]
//
//  KPROD is essentially constant after a certain point, so if N is
//  large, just take the last available value.
//
//   if ( 0 < n )
//   {
//     *c = *c * kprod [ i4_min ( n, KPROD_LENGTH ) - 1 ];
//     *s = *s * kprod [ i4_min ( n, KPROD_LENGTH ) - 1 ];
//   }
//
//  Adjust for possible sign change because angle was originally
//  not in quadrant 1 or 4.
//
//   *c = sign_factor * *c;
//   *s = sign_factor * *s;

  return;
# undef ANGLES_LENGTH
# undef KPROD_LENGTH
}
```