The difference between each cpps are described in write-up. Thank you!

Monte Carlo:

```
#include<iostream>
using namespace std;
#include "math.h"
#include "time.h"

static double d;
static double runtimes;
int timesin = 0;//in circle times

clock_t start, finish;//clock

//for statistic
double mean = 0.0;
double sum = 0.0;
double var = 0.0;
double wide = 0.0;

int listtimes = 0;//times of each "big" run
double lower = 0.0;
double higher = 0.0;

//simple list
class arraylist
{
private:
    int n;//present number
    double *list;
public:
    arraylist(int temp)
    {
        list = new double[temp];
        n = 0;
    }
    void addLast(double temp)
    {
        list[n] = temp;
        n++;
    }
    double getLast()
    {
        return list[n-1];
    }
    double get(int number)
    {
        return list[number];
    }
};
arraylist List(1000000);

//1-dimension
bool ifincircle1(double temp)
{
    if (temp >= -1.0&&temp <= 1.0)
    {
        return true;
    }
    return false;
}

void MontePointd1()
{
    for (int i = 0; i < runtimes; i++)
    {
        double temp = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        if (ifincircle1(temp))
        {
        timesin++;
        }
    }
}

//2-dimension
bool ifincircle2(double tempx, double tempy)
{
    if (tempx*tempx + tempy*tempy <= 1.0)
    {
        return true;
    }
    return false;
}

void MontePointd2()
{
    for (int i = 0; i < runtimes; i++)
    {
    /*  double x = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double y = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);*/
        double x = (-1.0 + ((1.0-(float)rand() / (RAND_MAX + 1)) )* 2);
        double y = (-1.0 + ((1.0 - (float)rand() / (RAND_MAX + 1))) * 2);
        if (ifincircle2(x, y))
        {
            timesin++;
        }
    }
}

//3-dimension
bool ifincircle3(double tempx, double tempy, double tempz)
{
    if ((tempx*tempx) + (tempy*tempy) + (tempz*tempz) <= 1.0)
    {
        return true;
    }
    return false;
}

void MontePointd3()
{
    for (int i = 0; i < runtimes; i++)
    {
        double x = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double y = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double z = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);

        if (ifincircle3(x, y, z))
        {
            timesin++;
        }
    }
}

//4-dimension
bool ifincircle4(double tempx, double tempy, double tempz,double tempo)
{
    if ((tempx*tempx) + (tempy*tempy) + (tempz*tempz) +(tempo*tempo)<= 1.0)
    {
        return true;
    }
    return false;
}

void MontePointd4()
{
    for (int i = 0; i < runtimes; i++)
    {
        double x = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double y = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double z = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double o = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);

        if (ifincircle4(x, y, z,o))
        {
            timesin++;
        }
    }
}

//5-dimension
bool ifincircle5(double tempx, double tempy, double tempz, double tempo,double tempq)
{
    if ((tempx*tempx) + (tempy*tempy) + (tempz*tempz) + (tempo*tempo)+(tempq*tempq) <= 1.0)
    {
        return true;
    }
    return false;
}

void MontePointd5()
```

```
{
    for (int i = 0; i < runtimes; i++)
    {
        double x = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double y = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double z = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double o = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);
        double q = (-1.0 + ((float)rand() / (RAND_MAX + 1)) * 2);

        if (ifincircle5(x, y, z, o,q))
        {
            timesin++;
        }
    }
}


void statistics()
{
    double temp = 0.0;
    if (d == 1)
    {
        temp = (timesin / runtimes)*2.0;
    }
    else if (d == 2)
    {
        temp = (timesin / runtimes)*4.0;
    }
    else if (d==3)
    {
        temp = (timesin / runtimes)*8.0;
    }
    else if (d==4)
    {
        temp = (timesin / runtimes)*16.0;
    }
    else if (d == 5)
    {
        temp = (timesin / runtimes)*32.0;
    }
    cout << "The volume of " << (int)d << "-dimensional (hyper-)sphere of redius r=1 centered on the origin is "   << temp << endl;
    List.addLast(temp);
}

void main()
{
    cout << "(Monte Carlo):Insert the dimension you want to calculate:";
    cin >> d;
    start = clock();
    //define runtimes
    if (d == 1)
    {
        runtimes = 1000000;
    }
    else if (d == 2)
    {
        runtimes = 1000000;
    }
    else if (d == 3)
    {
        runtimes = 1000000;
    }
    else if (d == 4)
    {
        runtimes = 1000000;
    }
    else if (d == 5)
    {
        runtimes = 1000000;
    }
    bool iffirsttime = true;

    for (;iffirsttime == true || ((int)(higher * 10000000) != (int)(lower * 10000000));)
    {
        if (d == 1)
        {
            MontePointd1();
        }
        else if (d == 2)
        {
            MontePointd2();
        }
        else if (d == 3)
        {
            MontePointd3();
        }
        else if (d == 4)
        {
            MontePointd4();
        }
        else if (d == 5)
        {
            MontePointd5();
        }

        statistics();
        sum += List.getLast();
        listtimes++;

        mean = sum / listtimes;

        for (int i = 0; i<listtimes; i++)
        {
            var += (List.get(i) - mean)*(List.get(i) - mean);
        }
        var = var / listtimes;

        lower = mean - 2.58*sqrt(var / listtimes);
        higher = mean + 2.58*sqrt(var / listtimes);
        cout <<"interval:"<< lower << "-" << higher << endl;

        timesin = 0;//initialize
        if (listtimes > 1)
        {
            iffirsttime = false;
        }
    }
    finish = clock();
    cout<< "The volume of " << (int)d << "-dimensional (hyper-)sphere of redius r=1 centered
on the origin with 99% confidence is " <<lower<<"-"<<higher<<"  acuurate result:"<< mean<<endl;
    cout << "Run Times:" << listtimes << endl;
    cout << "Run Time is:" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
    system("Pause");
}
```

Cube Based Version 3.0:

```
#include<iostream>
using namespace std;
#include "math.h"
#include "time.h"

//just cal the far distance from circle

#define EPSILON 0.000000001

double d;//dimention
long long runtimes;//cube number
clock_t start, finish;//time

double smallcube;//cube edge length
double halfcube;//half cube edge length

//use square to compare, more fast and accurate
double act_dis; //actuall distance
double dis; //judge dis
long long outcircle = 0;
long long incircle = 0;
long long crosscircle = 0;

class Coordinate
{
public:
    double x, y, z,h,g;
    Coordinate(double xx, double yy, double zz,double hh,double gg)
    {
        x = xx;
        y = yy;
        z = zz;
        h = hh;
        g = gg;
    }
};
//initialize small cube coordinate
Coordinate cube(-1.0, 1.0, -1.0,1.0,-1.0);

//return positive or negative
double posnegcal(double temp)
{
    if (temp<0)
    {
        return -1.0;
    }
    else if (temp>0)
    {
        return 1.0;
    }
    return 0.0;
}

void nextcube1()//dimention 1
{
    cube.x += smallcube;
}

void ifincircle1()
{
    act_dis = abs(cube.x);//no sqrt
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis >= dis)
    {
        outcircle++;
    }
}

void CubeSim1()
{
    ifincircle1();
    nextcube1();
}

void nextcube2()
{
    if (cube.x - 1.0<-EPSILON)//cube.x<1.0
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        cube.y -= smallcube;
    }
    if (cube.x <= EPSILON&&cube.x >= -EPSILON)//judge cube.x==0?
    {
        cube.x += smallcube;
    }
    if (cube.y <= EPSILON&&cube.y >= -EPSILON)
    {
        cube.y -= smallcube;
    }
}

void ifincircle2()
{
    act_dis = cube.x*cube.x + cube.y*cube.y;//no sqrt
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis >= dis)
    {
        outcircle++;
    }
    else//more accurate
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = l*l + m*m;
        if (n < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim2()
{
    ifincircle2();
    nextcube2();
}

void nextcube3()
{
    if (cube.x - 1.0<-EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y + 1.0> EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.z += smallcube;
            cube.y = 1.0;
        }
    }
    if (cube.x <= EPSILON&&cube.x >= -EPSILON)//judge cube.x==0?
    {
        cube.x += smallcube;
    }
    if (cube.y <= EPSILON&&cube.y >= -EPSILON)
    {
        cube.y -= smallcube;
    }
    if (cube.z <= EPSILON&&cube.z >= -EPSILON)
    {
        cube.z += smallcube;
    }
}

void ifincircle3()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z;// dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = cube.z - posnegcal(cube.z)*smallcube;
        double o = l*l + m*m + n*n;

        if (o < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim3()
{
    ifincircle3();
    nextcube3();
}

void nextcube4()
{
    if (cube.x - 1.0<-EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y + 1.0> EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.y = 1.0;
            if (cube.z - 1.0 < -EPSILON)
            {
                cube.z += smallcube;
            }
            else
            {
                cube.z = -1.0;
                cube.h -= smallcube;
            }
        }
    }
    if (cube.x <= EPSILON&&cube.x >= -EPSILON)//judge cube.x==0?change vertex
    {
        cube.x += smallcube;
    }
    if (cube.y <= EPSILON&&cube.y >= -EPSILON)
    {
        cube.y -= smallcube;
    }
    if (cube.z <= EPSILON&&cube.z >= -EPSILON)
    {
        cube.z += smallcube;
    }
    if (cube.h <= EPSILON&&cube.h >= -EPSILON)
    {
        cube.h -= smallcube;
    }
}

void ifincircle4()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z+cube.h*cube.h;// dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = cube.z - posnegcal(cube.z)*smallcube;
        double o = cube.h - posnegcal(cube.h)*smallcube;
        double p = l*l + m*m + n*n+o*o;

        if (p < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim4()
{
    ifincircle4();
    nextcube4();
```

```
}

void nextcube5()
{
    if (cube.x - 1.0<-EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y + 1.0> EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.y = 1.0;
            if (cube.z - 1.0 < -EPSILON)
            {
                cube.z += smallcube;
            }
            else
            {
                cube.z = -1.0;
                if (cube.h + 1.0 > EPSILON)
                {
                    cube.h -= smallcube;
                }
                else
                {
                    cube.h = 1.0;
                    cube.g += smallcube;
                }
            }
        }
    }
    if (cube.x <= EPSILON&&cube.x >= -EPSILON)//judge cube.x==0?change vertex
    {
        cube.x += smallcube;
    }
    if (cube.y <= EPSILON&&cube.y >= -EPSILON)
    {
        cube.y -= smallcube;
    }
    if (cube.z <= EPSILON&&cube.z >= -EPSILON)
    {
        cube.z += smallcube;
    }
    if (cube.h <= EPSILON&&cube.h >= -EPSILON)
    {
        cube.h -= smallcube;
    }
    if (cube.g <= EPSILON&&cube.g >= -EPSILON)
    {
        cube.g += smallcube;
    }
}

void ifincircle5()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z + cube.h*cube.h+cube.g*cube.g;//dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = cube.z - posnegcal(cube.z)*smallcube;
        double o = cube.h - posnegcal(cube.h)*smallcube;
        double p = cube.g - posnegcal(cube.g)*smallcube;
        double q = l*l + m*m + n*n + o*o+p*p;

        if (q < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim5()
{
    ifincircle5();
    nextcube5();
}



void statistics()
{
    double min;
    double max;
    double accurate ;
    if (d == 1)
    {
        min = 2.0*incircle / runtimes;
        max = 2.0*(incircle + crosscircle) / runtimes;
        accurate = 2.0*(incircle + crosscircle / 2) / runtimes;
    }
    if (d == 2)
    {
        min = 4.0*incircle / runtimes;
        max = 4.0*(incircle + crosscircle) / runtimes;
        accurate = 4.0*(incircle + crosscircle / 2) / runtimes;
    }
    if (d == 3)
    {
        min = 8.0*incircle / runtimes;
        max = 8.0*(incircle + crosscircle) / runtimes;
        accurate= 8.0*(incircle + crosscircle/2) / runtimes;
    }
    if (d == 4)
    {
        min = 16.0*incircle / runtimes;
        max = 16.0*(incircle + crosscircle) / runtimes;
        accurate = 16.0*(incircle + crosscircle / 2) / runtimes;
    }
    if (d == 5)
    {
        min = 32.0*incircle / runtimes;
        max = 32.0*(incircle + crosscircle) / runtimes;
        accurate = 32.0*(incircle + crosscircle / 2) / runtimes;
    }
    cout << "(Cube Based) The volume of " << (int)d << "-dimensional (hyper-)sphere of redius r=1 centered on the origin is between " << min << "-" << max<<" ("<<accurate<<") " << endl;
    cout << "Run time: " << (double)(finish - start) / CLOCKS_PER_SEC << endl;
}

void main()
{
    cout << "(Cube Based) Insert the dimension: ";
    cin >> d;
    start = clock();
    if (d == 1)
    {
        //initialize cube coordinate from left-up
        runtimes = 1000000;
        smallcube = 2 / runtimes;
        halfcube = 1 / runtimes;
        dis = 1.0;
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim1();
        }
        finish = clock();
    }
    if (d == 2)
    {
        //initialize cube coordinate from left-up
        runtimes = 1000000;
        smallcube = 2 / sqrt(runtimes);
        halfcube = 1 / sqrt(runtimes);
        dis = (1.0 + sqrt(2 * smallcube*smallcube))*(1.0 + sqrt(2 * smallcube*smallcube));//no sqrt, more accurate and fast
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim2();
        }
        finish = clock();
    }
    if (d == 3)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 1000000;
        double st = 100.0;
        smallcube = 2.0 / st;
        halfcube = 1.0 / st;
        dis = (1.0 + sqrt(3 * smallcube* smallcube))*(1.0 + sqrt(3 * smallcube* smallcube));
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim3();
        }
        finish = clock();
    }
    if (d == 4)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 1000000;
        smallcube = 2.0 / sqrt(sqrt(runtimes));
        halfcube = 1.0 / sqrt(sqrt(runtimes));
        dis = (1.0 + sqrt(4 * smallcube* smallcube))*(1.0 + sqrt(4 * smallcube* smallcube));
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim4();
        }
        cout << sqrt(sqrt(runtimes));
        finish = clock();
    }
    if (d == 5)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 1048576;
        double st = 16.0;
        smallcube = 2.0 / st;
        halfcube = 1.0 / st;
        dis = (1.0 + sqrt(5 * smallcube* smallcube))*(1.0 + sqrt(5 * smallcube* smallcube));
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim5();
        }
        finish = clock();
    }
    statistics();
    system("Pause");
}
```

Cube Based Test:(note:this is just for test, use the symmetry of “circle”, and you may carefully insert number in main)

```
#include<iostream>
using namespace std;
#include "math.h"
#include "time.h"

//just cal the far distance from circle

#define EPSILON 0.000000001

double d;//dimention
long long runtimes;//cube number
clock_t start, finish;//time

double smallcube;//cube edge length
double halfcube;//half cube edge length

//use square to compare, more fast and accurate
double act_dis; //actuall distance
double dis; //judge dis
double diagonal;//little cube diagonal
long long outcircle = 0;
long long incircle = 0;
long long crosscircle = 0;
```

```
class Coordinate
{
public:
    double x, y, z, h, g;
    Coordinate(double xx, double yy, double zz, double hh, double gg)
    {
        x = xx;
        y = yy;
        z = zz;
        h = hh;
        g = gg;
    }
};
//initialize small cube coordinate
Coordinate cube(-1.0, 1.0, -1.0, 1.0, -1.0);

//return positive or negative
double posnegcal(double temp)
{
    if (temp<0)
    {
        return -1.0;
    }
    else if (temp>0)
    {
        return 1.0;
    }
    return 0.0;
}

void nextcube2()
{
    if (cube.x - 1.0<-EPSILON)//cube.x<1.0
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        cube.y -= smallcube;
    }
    if (cube.x <= EPSILON&&cube.x >= -EPSILON)//judge cube.x==0?
    {
        cube.x += smallcube;
    }
    if (cube.y <= EPSILON&&cube.y >= -EPSILON)
    {
        cube.y -= smallcube;
    }
}

void ifincircle2()
{
    act_dis = cube.x*cube.x + cube.y*cube.y;//no sqrt
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis >= dis)
    {
        outcircle++;
    }
    else//more accurate
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = l*l + m*m;

        if (n < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim2()
{
    ifincircle2();
    nextcube2();
}

void nextcube3()
{
    if (cube.x <-smallcube - EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y > smallcube +EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.z += smallcube;
            cube.y = 1.0;
        }
    }
}

void ifincircle3()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z;// dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else//more accurate
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double o = cube.z - posnegcal(cube.z)*smallcube;
        double n = l*l + m*m+o*o;

        if (n < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim3()
{
    ifincircle3();
    nextcube3();
}

void nextcube4()
{
    if (cube.x <-smallcube - EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y > smallcube+EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.y = 1.0;
            if (cube.z <-smallcube - EPSILON)
            {
                cube.z += smallcube;
            }
            else
            {
                cube.z = -1.0;
                cube.h -= smallcube;
            }
        }
    }
}

void ifincircle4()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z + cube.h*cube.h;// dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = cube.z - posnegcal(cube.z)*smallcube;
        double o = cube.h - posnegcal(cube.h)*smallcube;
        double p = l*l + m*m + n*n + o*o;

        if (p < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim4()
{
    ifincircle4();
    nextcube4();
```

```
}

void nextcube5()
{
    if (cube.x <-smallcube-EPSILON)
    {
        cube.x += smallcube;
    }
    else//cube.x==1.0-halfcube
    {
        cube.x = -1.0;
        if (cube.y >smallcube+ EPSILON)//cube.y > -1.0
        {
            cube.y -= smallcube;
        }
        else
        {
            cube.y = 1.0;
            if (cube.z <-smallcube -EPSILON)
            {
                cube.z += smallcube;
            }
            else
            {
                cube.z = -1.0;
                if (cube.h>smallcube+ EPSILON)
                {
                    cube.h -= smallcube;
                }
                else
                {
                    cube.h = 1.0;
                    cube.g += smallcube;
                }
            }
        }
    }
}

void ifincircle5()
{
    act_dis = cube.x*cube.x + cube.y*cube.y + cube.z*cube.z + cube.h*cube.h + cube.g*cube.g;//
dis square
    if (act_dis <= 1.0)
    {
        incircle++;
    }
    else if (act_dis > dis)
    {
        outcircle++;
    }
    else
    {
        double l = cube.x - posnegcal(cube.x)*smallcube;
        double m = cube.y - posnegcal(cube.y)*smallcube;
        double n = cube.z - posnegcal(cube.z)*smallcube;
        double o = cube.h - posnegcal(cube.h)*smallcube;
        double p = cube.g - posnegcal(cube.g)*smallcube;
        double q = l*l + m*m + n*n + o*o + p*p;

        if (q < 1.0)
        {
            crosscircle++;
        }
        else
        {
            outcircle++;
        }
    }
}

void CubeSim5()
{
    ifincircle5();
    nextcube5();
}


void statistics()
{
    double min;
    double max;
    double accurate;
    crosscircle = runtimes - incircle-outcircle;
    if (d == 2)
    {
        min = 4.0*incircle / runtimes;
        max = 4.0*(runtimes - outcircle) / runtimes;
        accurate=4.0*(incircle + crosscircle / 2) / runtimes;
    }
    if (d == 3)
    {
        min = 8.0*incircle / runtimes;
        max = 8.0*(runtimes- outcircle) / runtimes;
        accurate = 8.0*(incircle + crosscircle/2) / runtimes;
    }
    if (d == 4)
    {
        min = 16.0*incircle / runtimes;
        max = 16.0*(runtimes - outcircle) / runtimes;
        accurate = 16.0*(incircle + crosscircle / 2) / runtimes;
    }
    if (d == 5)
    {
        min = 32.0*incircle / runtimes;
        max = 32.0*(runtimes - outcircle) / runtimes;
        accurate = 32.0*(incircle + crosscircle / 2) / runtimes;
    }
    cout << "(Cube Based) The volume of " << (int)d << "-dimensional (hyper-)sphere of redius r=1 centered on the origin is between " << min << "-" << max<<" ("<<accurate<<") " << endl;
    cout << "Run time: " << (double)(finish - start) / CLOCKS_PER_SEC << endl;
}

void main()
{
    cout << "(Cube Based) Insert the dimension: ";
    cin >> d;
    start = clock();
    if (d == 2)
    {
        //initialize cube coordinate from left-up
        runtimes = 225000000;// you need to put total runtimes/4 here
        double st = 15000.0;// the k value
        smallcube = 2 / st;
        halfcube = 1 / st;
        dis = (1.0 + sqrt(2 * smallcube*smallcube))*(1.0 + sqrt(2 * smallcube*smallcube));//no sqrt, more accurate and fast
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim2();
        }
        finish = clock();
    }
    if (d == 3)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 125000000; // you need to put total runtimes/8 here
        double st = 1000.0;// the k value
        smallcube = 2.0 / st;
        halfcube = 1.0 / st;
        diagonal = sqrt(3 * smallcube*smallcube);
        dis = (1.0 + sqrt(3 * smallcube* smallcube))*(1.0 + sqrt(3 * smallcube* smallcube));
        for (;cube.z<0.0;)
        {
            CubeSim3();
        }
        finish = clock();
    }
    if (d == 4)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 1600000000;// you need to put total runtimes/16 here
        double st = 400.0;
        smallcube = 2.0 / st;
        halfcube = 1.0 / st;
        dis = (1.0 + sqrt(4 * smallcube* smallcube))*(1.0 + sqrt(4 * smallcube* smallcube));
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim4();
        }
        finish = clock();
    }
    if (d == 5)
    {
        //initialize cube coordinate from back-left-up
        runtimes = 10000000000;// you need to put total runtimes/32 here
        double st = 200.0;
        smallcube = 2.0 / st;
        halfcube = 1.0 / st;
        dis = (1.0 + sqrt(5 * smallcube* smallcube))*(1.0 + sqrt(5 * smallcube* smallcube));
        for (int i = 0; i < runtimes; i++)
        {
            CubeSim5();
        }
        finish = clock();
    }
    statistics();
    system("Pause");
}
```