```c
/*
 * ME2055 - Computational Fluid Dynamics
 *
 * Heat transfer in a cylindrical coordinate system.
 *
 * Author: Dustin (Ting-Hsuan) Ma
 *
 * Compile: gcc cpuHeat.c -O3 -lm -o CPU.exe
 * Execute: ./CPU <Number of Iteration>
 * Clang: clang-format -i cpuHeat.c
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

// Constants
#define LRL ( REAL ) 10.f   /* mm */
#define LRS ( REAL ) 7.f    /* mm */
#define ANG1 ( REAL ) 25.f  /* Degree */
#define ANG2 ( REAL ) 70.f  /* Degree */
#define T1 ( REAL ) 300.0f  /* Kelvin */
#define T2 ( REAL ) 200.0f  /* Kelvin */
#define TNOT ( REAL ) 0.0f  /* Kevlin  */
#define QP3 ( REAL ) 5.0e-2 /* W/mm^3  */
#define K ( REAL ) 20.0e-3  /* W/mm*K */
#define H ( REAL ) 1.0      /* Unit depht 1 mm */
#define RESIDUAL ( REAL ) 1e-10

// Mesh
#define NORMFACTOR ( INT ) 1
#define NR (INT)((NORMFACTOR * 10) + 1)
#define NANG (INT)((NORMFACTOR * 14) + 1)
#define DR (REAL) LRL / (( REAL ) NR - 1.f)
#define DANG (REAL)(ANG2) / (( REAL ) NANG - 1.f)

// Calculation index
#define IC i + j *NR
#define IP1 (i + 1) + j *NR
#define IM1 (i - 1) + j *NR
#define JP1 i + (j + 1) * NR
#define JM1 i + (j - 1) * NR

// Conversion Factors
#define CONV M_PI / 180.f

#define DEBUG 0
#define GAUSS 0

typedef double REAL;
typedef int    INT;

#if (GAUSS)
void gaussMethod(REAL *phi, REAL *out)
{
    for (INT j = 1; j < NANG; j++) {
        for (INT i = 1; i < NR - 1; i++) {
            REAL r      = i * DR;
            REAL dtheta = DANG * CONV;
            REAL a      = 2 / (DR * DR) + 2 / (r * r * dtheta * dtheta);
            REAL b      = (1 / (DR * DR) - 1 / (2 * r * DR)) / a;
            REAL c      = (1 / (DR * DR) + 1 / (2 * r * DR)) / a;
            REAL d      = (1 / (r * r * dtheta * dtheta)) / a;
            REAL f      = (QP3 / K) / a;
            out[ IC ]   = b * phi[ IM1 ] + c * phi[ IP1 ] + d * phi[ JP1 ] + d * phi[ JM1 ] + f;
            phi[ IC ]   = out[ IC ];
        }
    }
}
#else
void jacobiMethod(const REAL *phi, REAL *out, const REAL *rGrid)
{
    for (INT j = 1; j < NANG; j++) {
        for (INT i = 1; i < NR - 1; i++) {
            REAL r      = i * DR;
            REAL dtheta = DANG * CONV;
            REAL a      = 2 / (DR * DR) + 2 / (r * r * dtheta * dtheta);
            REAL b      = (1 / (DR * DR) - 1 / (2 * r * DR)) / a;
            REAL c      = (1 / (DR * DR) + 1 / (2 * r * DR)) / a;
            REAL d      = (1 / (r * r * dtheta * dtheta)) / a;
            REAL f      = (QP3 / K) / a;
            out[ IC ]   = b * phi[ IM1 ] + c * phi[ IP1 ] + d * phi[ JP1 ] + d * phi[ JM1 ] + f;
        }
    }
}
```



```c
#endif

void boundaryConditionsD(REAL *phi, REAL *rGrid, REAL *angGrid)
{
    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 0; i < NR; i++) {
            if (angGrid[ IC ] <= ANG1) {
                //if (angGrid[ IC ] == 0) phi[ JM1 ] = phi[ JP1 ];                // Insulated Side 1
                //if (rGrid[ IC ] == LRS) phi[ IP1 ] = phi[ IM1 ];                // Insulated Side 2
                if (angGrid[ IC ] == ANG1 && rGrid[ IC ] >= LRS) phi[ IC ] = T2; // T2 Side
            } else {
                if (angGrid[ IC ] == ANG2) phi[ IC ] = T1; // T1 side
                if (rGrid[ IC ] == LRL) phi[ IC ] = T1;    // T1 side
            }
            if (rGrid[ IC ] == 0) phi[ IC ] = T1;                          // default
            if (rGrid[ IC ] > LRS && angGrid[ IC ] < ANG1) phi[ IC ] = 0; // NULL section
        }
    }
}
void boundaryConditionsN(REAL *phi, REAL *rGrid, REAL *angGrid)
{
    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 0; i < NR; i++) {
            if (angGrid[ IC ] < ANG1) {
                if (angGrid[ IC ] == 0) phi[ JM1 ] = phi[ JP1 ];                // Insulated Side 1
                if (rGrid[ IC ] == LRS) phi[ IP1 ] = phi[ IM1 ];                // Insulated Side 2
            }
        }
        }
}

void verifyingPhysics(const REAL *phi, REAL *rGrid, REAL *angGrid, REAL *qPrime)
{
    REAL side1 = 0.f;
    REAL side2 = 0.f;
    REAL side3 = 0.f;

    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 1; i < NR; i++) {
            if (angGrid[ IC ] == ANG1 && rGrid[ IC ] >= LRS) { // 200K side
                side1 += -K * DR * (phi[ IC ] - phi[ JP1 ]) / (rGrid[ IC ] * DANG * CONV);
            }
            if (angGrid[ IC ] == ANG2) { // 300K side - Straight
                side2 += -K * DR * (phi[ IC ] - phi[ JM1 ]) / (rGrid[ IC ] * DANG * CONV);
            }
            if (rGrid[ IC ] == LRL && angGrid[ IC ] >= ANG1) { // 300K side - Curved
                side3 += -K * DANG * CONV * (phi[ IC ] - phi[ IM1 ]) / DR;
            }
        }
    }
    printf("side1 = %f, side2 = %f, side3 = %f \n", side1, side2, side3);
    *qPrime = side1 + side2 + side3;
}

void l2Norm(const REAL *phi, REAL *norm)
{
    REAL sum = 0.f;
    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 1; i < NR - 1; i++) {
            sum += abs(phi[ IC ] * phi[ IC ]);
        }
    }
    *norm = sqrt(sum);
}

void meshGrid(REAL *rGrid, REAL *angGrid, REAL *xGrid, REAL *yGrid)
{
    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 0; i < NR; i++) {
            rGrid[ IC ]   = i * DR;
            angGrid[ IC ] = (j - 1) * DANG;
            xGrid[ IC ]   = rGrid[ IC ] * cos(angGrid[ IC ] * CONV);
            yGrid[ IC ]   = rGrid[ IC ] * sin(angGrid[ IC ] * CONV);
        }
    }
}

void outputMatrix(const REAL *in, char *name)
{
    FILE *file = fopen(name, "w");
    for (INT j = 1; j < NANG + 1; j++) {
        for (INT i = 0; i < NR; i++) {
            fprintf(file, "%6.2f", in[ IC ]);
        }
```

```c
        fprintf(file, "\n");
    }
    fprintf(file, "\n");
    fclose(file);
}

INT main(int argc, char *argv[])
{
    // Allocating memory
    REAL *rGrid     = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*rGrid));     // Radius Grid
    REAL *angGrid   = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*angGrid));   // Angle Grid
    REAL *xGrid     = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*xGrid));     // X Grid
    REAL *yGrid     = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*yGrid));     // Y Grid
    REAL *theta     = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*theta));     // Theta phi
    REAL *theta_new = ( REAL * ) calloc(NR * (NANG + 1), sizeof(*theta_new)); // Next Theta
    REAL *norm      = ( REAL * ) calloc(2, sizeof(*norm));
    REAL *tmp, qPrime[ 0 ];

    // Initializing matrix boundaries
    meshGrid(rGrid, angGrid, xGrid, yGrid);
    boundaryConditionsD(theta, rGrid, angGrid);
    outputMatrix(xGrid, "xGrid.csv");
    outputMatrix(yGrid, "yGrid.csv");
    outputMatrix(rGrid, "rGrid.csv");
    outputMatrix(angGrid, "angGrid.csv");

    REAL diff = 1.f;
    INT  iter = 0;

    while (RESIDUAL < diff) {
#if (GAUSS)
        gaussMethod(theta, theta_new, rGrid);
#else
        jacobiMethod(theta, theta_new, rGrid);
#endif
        boundaryConditionsD(theta_new, rGrid, angGrid);
        boundaryConditionsN(theta_new, rGrid, angGrid);

        norm[ 0 ] = norm[ 1 ];
        l2Norm(theta_new, &norm[ 1 ]);
        diff = norm[ 1 ] - norm[ 0 ];

        tmp       = theta;
        theta     = theta_new;
        theta_new = tmp;
        iter++;
    }

    boundaryConditionsD(theta_new, rGrid, angGrid);


    // Verifying Physics and Output Final values
    verifyingPhysics(theta_new, rGrid, angGrid, &qPrime[ 0 ]);
    printf("===== Answer with %d Iterations ===== \n", iter);
    outputMatrix(theta_new, "Temperature.csv");
    printf("q' = %5.4f [kW], Residual = %f \n", qPrime[ 0 ], diff);

    // Deallocating Memory
    free(theta);
    free(theta_new);
    free(rGrid);
    free(angGrid);
    free(norm);
    theta     = NULL;
    theta_new = NULL;
    rGrid     = NULL;
    angGrid   = NULL;
    norm      = NULL;
    return EXIT_SUCCESS;
}
```