MAE 267 – Project 1
Serial, Single-Block, Finite-Volume Methods For Solving 2D Heat Conduction

Logan Halstrom
PhD Graduate Student Researcher
Center for Human/Robot/Vehicle Integration and Performance
Department of Mechanical and Aerospace Engineering
University of California, Davis
Davis, California 95616
Email: ldhalstrom@ucdavis.edu

1 Statement of Problem

This analysis details the solution of the steady-state temperature distribution on a 1m x 1m block of steel with Dirichlet boundary conditions (Eqn 2). Solutions were performed on square, non-uniform grids rotatated in the positive z-direction by $rot = 30^o$. Two grids of 101x101 points and 501x501 points were used to solve the equation of heat transfer. Temperature was uniformly initialized to a value of 3.5 and the solution was iterated until the maximum residual found was less than $1.0x10^{-5}$. The equation for heat conduction (Eqn 1) was solved using an explicit, node-centered, finite-volume scheme, with an alternative distributive scheme for the second-derivative operator. Steady-state temperature distribution was saved in a PLOT3D unformatted file, and CPU wall time of the solver was recorded.

2 Equations and Algorithms

The solver developed for this analysis utilizes a finite-volume numerical solution method to solve the transient heat conduction equation (Eqn 1).

$$\rho c_p \frac{\partial T}{\partial t} = k\left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right] \tag{1}$$

The solution is initialized with the Dirichlet boundary conditions (Eqn 2).

$$T = \begin{cases} 5.0\left[\sin\left(\pi x_p\right) + 1.0\right] & \text{for } j = j_{max} \\ \left|\cos\left(\pi x_p\right)\right| + 1.0 & \text{for } j = 0 \\ 3.0 y_p + 2.0 & \text{for } i = 0, i_{max} \end{cases} \tag{2}$$

Grids were generated according to the following (Eqn 3)

$$\begin{aligned} rot &= 30.0\frac{\pi}{180.0} \\ x_p &= \cos\left[0.5\pi\frac{i_{max}-i}{i_{max}-1}\right] \\ y_p &= \cos\left[0.5\pi\frac{j_{max}-j}{j_{max}-1}\right] \\ x(i,j) &= x_p\cos(rot) + (1.0 - y_p)\sin(rot) \\ y(i,j) &= y_p\cos(rot) + x_p\sin(rot) \end{aligned} \tag{3}$$

To solve Eqn 1 numerically, the equation is discretized according to a node-centered finite-volume scheme, where first-derivatives at the nodes are found using Green's theorem integrating around the secondary control volumes. Trapezoidal, counter-clockwise integration for the first-derivative in the x-direction is achieved with Eqn 4.

$$\begin{aligned} \frac{\partial T}{\partial x} = \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} &\left[(T_{i+1,j} + T_{i+1,j+1})Ayi_{i+1,j}\right. \\ &- (T_{i,j} + T_{i,j+1})Ayi_{i,j} \\ &- (T_{i,j+1} + T_{i+1,j+1})Ayi_{i,j+1} \\ &\left.- (T_{i,j} + T_{i+1,j})Ayi_{i,j}\right] \end{aligned} \tag{4}$$

A similar scheme is used to find the first-derivative in the y-direction.

3 Results and Discussion

Both grids used in this analysis were non-uniformly distributed according to the same function and can be observed in Figs 1 and 2

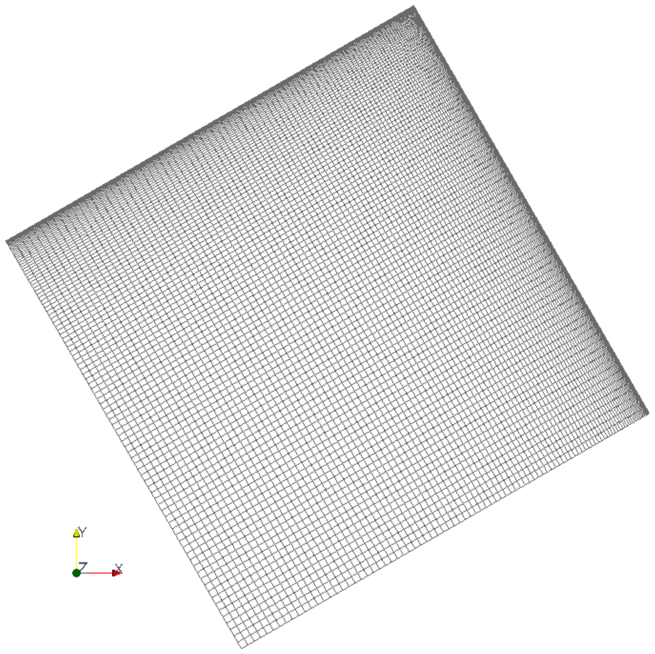

Fig. 1: 101x101 point mesh

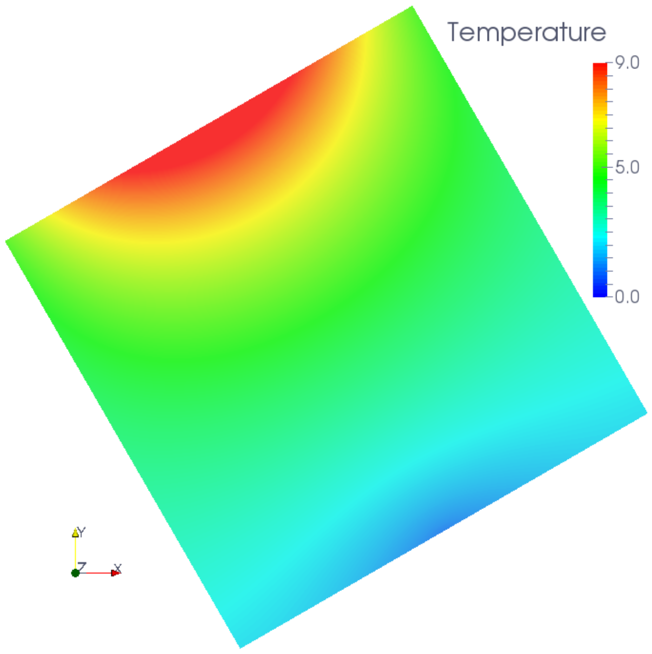

Fig. 3: Steady-state temperature distribution on a 101x101 mesh

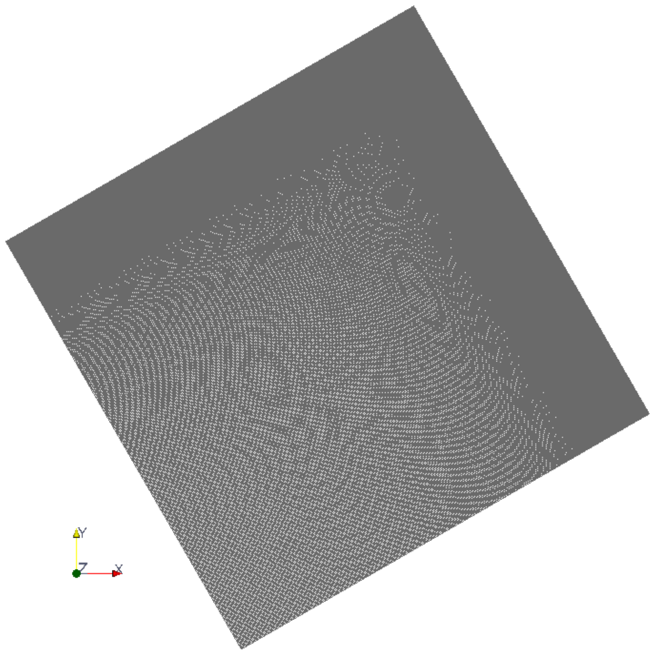

Fig. 2: 501x501 point mesh

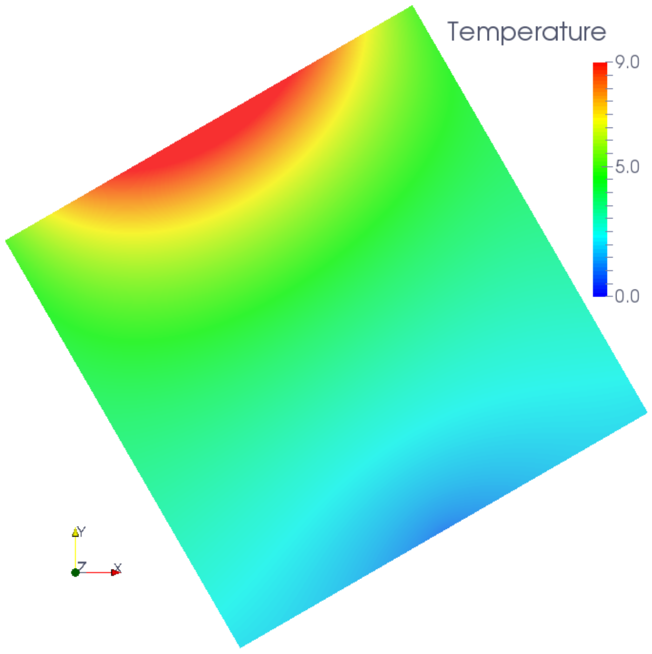

Fig. 4: Steady-state temperature distribution on a 501x501 mesh

It can be seen that the mesh becomes more refined nearer i_{max} and j_{max}, and that the 501x501 point mesh is significantly more dense than its counterpart, which resulted in much longer wall times for solutions.

Figs 3 and 4 show the steady-state temperature distribution on the steel plate for each mesh. Very little difference is apparent, with the 501x501 mesh being slightly more dissipative near the hot/cold boundaries.

4 Conclusion

This project has produced a functional algorithm for solving steady-state heat conduction in serial. Though CPU wall time of the 101x101 point grid was minimal (16.88 seconds), significant wall time was required to converge the solution for the 501x501 point grid (4763 seconds). See Appendix A for more run parameter output. Wall time could be reduced by parallelizing the code.

Appendix A: Sample Output

```
 Running a         101 by         101 grid took:
       15987 iterations
   16.881096124649048      seconds (Total CPU walltime)
   16.868424892425537      seconds (Solver CPU walltime)

 Found max residual of    9.9976378598399043E-006
 At ij of           39          39
```

Listing 1: Sample output for 101x101 grid solution

```
 Running a         501 by         501 grid took:
      176325 iterations
   4763.3411269187927      seconds (Total CPU walltime)
   4763.1624689102173      seconds (Solver CPU walltime)

 Found max residual of    9.9999431423345320E-006
 At ij of          196         207
```

Listing 2: Sample output for 501x501 grid solution

Appendix B: Source Code

```fortran
! MAE 267
! PROJECT 1
! LOGAN HALSTROM
! 12 OCTOBER 2015


! DESCRIPTION:  Solve heat conduction equation for single block of steel.
! To compile: mpif90 -o main modules.f90 plot3D_module.f90 subroutines.f90 main.f90
    ! makes executable file 'main'
    ! run with ./main or ./runjob.sh
! 'rm *.mod' afterward to clean up unneeded compiled files

PROGRAM heatTrans
!     USE CLOCK
    USE CONSTANTS
    USE subroutines
    USE plot3D_module

    IMPLICIT NONE

    ! GRID
    TYPE(MESHTYPE), TARGET, ALLOCATABLE :: mesh(:,:)
    TYPE(CELLTYPE), TARGET, ALLOCATABLE :: cell(:,:)
    ! ITERATION PARAMETERS
    ! Minimum Residual
    REAL(KIND=8) :: min_res = 0.00001D0
    ! Maximum number of iterations
    INTEGER :: max_iter = 1000000, iter = 0

    INCLUDE "mpif.h"
    REAL(KIND=8) :: start_total, end_total
    REAL(KIND=8) :: start_solve, end_solve
    ! CLOCK TOTAL TIME OF RUN
    start_total = MPI_Wtime()


    ! MAKE GRID
    ! Set grid size
    CALL GRIDSIZE(101)
    ALLOCATE(mesh(1:IMAX, 1:JMAX))
    ALLOCATE(cell(1:IMAX-1, 1:JMAX-1))

    ! INIITIALIZE SOLUTION
    WRITE(*,*) 'Making mesh...'
    CALL init(mesh, cell)

    ! MEASURE WALL TIME FOR OVERALL SOLUTION
!     WRITE(*,*) 'Starting clock for solver...'
! !     CALL start_clock()
!     start_solve = MPI_Wtime()

    ! SOLVE
    WRITE(*,*) 'Solving heat conduction...'
    CALL solve(mesh, cell, min_res, max_iter, iter)

!     CALL end_clock()
!     end_solve = MPI_Wtime()
!     end_total = MPI_Wtime()
!     wall_time_solve = start_solve - end_solve
!     wall_time_total = start_total - end_total

    WRITE(*,*) 'Writing results...'
    ! SAVE SOLUTION AS PLOT3D FILES
    CALL plot3D(mesh)
    ! CALC TOTAL WALL TIME
    end_total = MPI_Wtime()
    wall_time_total = start_total - end_total
    ! SAVE SOLVER PERFORMANCE PARAMETERS
    CALL output(mesh, iter)


    ! CLEAN UP
    DEALLOCATE(mesh)
    DEALLOCATE(cell)
    WRITE(*,*) 'Done!'


END PROGRAM heatTrans
```

Listing 3: Wrapper program for solution of 2D heat conduction

```fortran
! MAE 267
! PROJECT 1
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  Subroutines used for solving heat conduction of steel plate.
! Utilizes modules from 'modules.f90'
! CONTENTS:
! init --> Initialize the solution with dirichlet B.C.s
! solve --> Solve heat conduction equation with finite volume scheme
! output --> Save solution parameters to file

MODULE subroutines
    USE CONSTANTS
    USE MESHMOD
    USE CELLMOD
    USE TEMPERATURE
!     USE CLOCK

    IMPLICIT NONE

CONTAINS
    SUBROUTINE init(mesh, cell)
        ! Initialize the solution with dirichlet B.C.s
        TYPE(MESHTYPE), TARGET :: mesh(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE), TARGET :: cell(1:IMAX-1, 1:JMAX-1)
        INTEGER :: i, j

        ! INITIALIZE MESH
        CALL init_mesh(mesh)
        ! INITIALIZE CELLS
        CALL init_cells(mesh, cell)
        ! CALC SECONDARY AREAS OF INTEGRATION
        CALL calc_2nd_areas(mesh, cell)
        ! CALC CONSTANTS OF INTEGRATION
        CALL calc_constants(mesh, cell)

        ! INITIALIZE TEMPERATURE WITH DIRICHLET B.C.
        !PUT DEBUG BC HERE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        DO j = 1, JMAX
            CALL init_temp(mesh(1,j), 3.D0 * mesh(1,j)%yp + 2.D0)
            CALL init_temp(mesh(IMAX,j), 3.D0 * mesh(IMAX,j)%yp + 2.D0)
        END DO

        DO i = 1, IMAX
            CALL init_temp(mesh(i,1), ABS(COS(pi * mesh(i,1)%xp)) + 1.D0)
            CALL init_temp(mesh(i,JMAX), 5.D0 * (SIN(pi * mesh(i,JMAX)%xp) + 1.D0))
        END DO
    END SUBROUTINE init

    SUBROUTINE solve(mesh, cell, min_res, max_iter, iter)
        ! Solve heat conduction equation with finite volume scheme
        TYPE(MESHTYPE) :: mesh(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE) :: cell(1:IMAX-1, 1:JMAX-1)
        ! Minimum residual criteria for iteration, actual residual
        REAL(KIND=8) :: min_res, res = 1000.D0
        ! iteration number, maximum number of iterations
        ! iter in function inputs so it can be returned to main
        INTEGER :: iter, max_iter
        INTEGER :: i, j

        INCLUDE "mpif.h"
        REAL(KIND=8) :: start_solve, end_solve
        WRITE(*,*) 'Starting clock for solver...'
        start_solve = MPI_Wtime()

        iter_loop: DO WHILE (res >= min_res .AND. iter <= max_iter)
            ! Iterate FV solver until residual becomes less than cutoff or
            ! iteration count reaches given maximum


!             ! CLOCK TOTAL TIME OF iteration loop
!             start_iter = MPI_Wtime()

            ! INCREMENT ITERATION COUNT
            iter = iter + 1
            ! CALC NEW TEMPERATURE AT ALL POINTS
            CALL derivatives(mesh, cell)
            ! SAVE NEW TEMPERATURE DISTRIBUTION
            DO j = 2, JMAX - 1
                DO i = 2, IMAX - 1
                    mesh(i,j)%T = mesh(i,j)%T + mesh(i,j)%Ttmp
                END DO
            END DO

!             end_iter = MPI_Wtime()
!             IF (iter < 6) THEN
!                 wall_time_iter(iter) = end_iter - start_iter
!             END IF

            ! CALC RESIDUAL
            res = MAXVAL(ABS(mesh(2:IMAX-1, 2:JMAX-1)%Ttmp))
        END DO iter_loop

        ! CACL SOLVER WALL CLOCK TIME
        end_solve = MPI_Wtime()
        wall_time_solve = end_solve - start_solve

        ! SUMMARIZE OUTPUT
        IF (iter > max_iter) THEN
```

```fortran
          WRITE(*,*) 'DID NOT CONVERGE (NUMBER OF ITERATIONS:', iter, ')'
        ELSE
          WRITE(*,*) 'CONVERGED (NUMBER OF ITERATIONS:', iter, ')'
        END IF
    END SUBROUTINE solve

    SUBROUTINE output(mesh, iter)
        ! Save solution parameters to file
        TYPE(MESHTYPE), TARGET :: mesh(1:IMAX, 1:JMAX)
        REAL(KIND=8), POINTER :: Temperature(:,:), tempTemperature(:,:)
        INTEGER :: iter, i, j

        Temperature => mesh(2:IMAX-1, 2:JMAX-1)%T
        tempTemperature => mesh(2:IMAX-1, 2:JMAX-1)%Ttmp
        ! Let's find the last cell to change temperature and write some output.
        ! Write down the 'steady state' configuration.
        OPEN(UNIT = 1, FILE = "SteadySoln.dat")
        DO i = 1, IMAX
            DO j = 1, JMAX
                WRITE(1,'(F10.7, 5X, F10.7, 5X, F10.7, I5, F10.7)'), mesh(i,j)%x, mesh(i,j)%y, mesh(i,j)%T
            END DO
        END DO
        CLOSE (1)

        ! Output to the screen so we know something happened.
        WRITE (*,*), "IMAX/JMAX", IMAX, JMAX
        WRITE (*,*), "iters", iter
        WRITE (*,*), "residual", MAXVAL(tempTemperature)
        WRITE (*,*), "ij", MAXLOC(tempTemperature)

        ! Write down info for project
        OPEN (UNIT = 2, FILE = "SolnInfo.dat")
        WRITE (2,*), "Running a", IMAX, "by", JMAX, "grid took:"
        WRITE (2,*), iter, "iterations"
        WRITE (2,*), wall_time_total, "seconds (Total CPU walltime)"
        WRITE (2,*), wall_time_solve, "seconds (Solver CPU walltime)"
!         WRITE (2,*), wall_time_iter, "seconds (Iteration CPU walltime)"
        WRITE (2,*)
        WRITE (2,*), "Found max residual of ", MAXVAL(tempTemperature)
        WRITE (2,*), "At ij of ", MAXLOC(tempTemperature)
        CLOSE (2)
    END SUBROUTINE output
END MODULE subroutines
```

Listing 4: Main subroutines for solver (initialization/solution/output)

```fortran
! MAE 267
! PROJECT 1
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  Modules used for solving heat conduction of steel plate.
! Initialize and store constants used in all subroutines.

! CONTENTS:
! CONSTANTS --> Initializes constants for simulation.  Sets grid size.
! CLOCK --> Calculates clock wall-time of a process.
! MAKEGRID --> Initialize grid with correct number of points and rotation,
!                  set boundary conditions, etc.
! CELLS -->  Initialize finite volume cells and do associated calculations
! TEMPERATURE --> Calculate and store new temperature distribution
!                      for given iteration

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! CONSTANTS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE CONSTANTS
    ! Initialize constants for simulation.  Set grid size.
    IMPLICIT NONE
    ! CFL number, for convergence (D0 is double-precision, scientific notation)
    REAL(KIND=8), PARAMETER :: CFL = 0.5D0
    ! Material constants (steel): thermal conductivity [W/(m*K)],
                                ! density [kg/m^3],
                                ! specific heat ratio [J/(kg*K)]
    REAL(KIND=8), PARAMETER :: k = 18.8D0, rho = 8000.D0, cp = 500.D0
    ! Thermal diffusivity [m^2/s]
    REAL(KIND=8), PARAMETER :: alpha = k / (cp * rho)
    ! Pi, grid rotation angle (30 deg)
    REAL(KIND=8), PARAMETER :: pi = 3.141592654D0, rot = 30.D0*pi/180.D0
    ! CPU Wall Times
    REAL(KIND=8) :: wall_time_total, wall_time_solve, wall_time_iter(1:5)
    ! Grid size
    INTEGER :: IMAX, JMAX

CONTAINS
    SUBROUTINE GRIDSIZE(n)
        ! Set size of grid (square)
        INTEGER :: n
        IMAX = n
        JMAX = n
    END SUBROUTINE GRIDSIZE
END MODULE CONSTANTS

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! WALL CLOCK TIME !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE CLOCK
    ! Calculates clock wall-time of a process.
    INTEGER clock_start, clock_end, clock_max, clock_rate
    REAL(KIND=8) wall_time

CONTAINS
    SUBROUTINE start_clock()
        ! get clock parameters
        CALL SYSTEM_CLOCK(count_max=clock_max, count_rate=clock_rate)
        ! Get start time
        CALL SYSTEM_CLOCK(clock_start)
    END SUBROUTINE start_clock

    SUBROUTINE end_clock()
        ! Get end time
        CALL SYSTEM_CLOCK(clock_end)
        wall_time = DFLOAT(clock_end - clock_start) / DFLOAT(clock_rate)
        PRINT*, 'Solver wall clock time (seconds):', wall_time
    END SUBROUTINE end_clock
END MODULE CLOCK

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! INITIALIZE GRID !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE MESHMOD
    ! Initialize grid with correct number of points and rotation,
    ! set boundary conditions, etc.
    USE CONSTANTS

    IMPLICIT NONE
    PUBLIC

    TYPE MESHTYPE
        ! DERIVED DATA TYPE
        INTEGER :: i, j
        ! Grid points, see cooridinate rotaion equations in problem statement
        REAL(KIND=8) :: xp, yp, x, y
        ! Temperature at each point, temporary variable to hold temperature sum
        REAL(KIND=8) :: T, Ttmp
        ! Iteration Parameters: timestep, secondary cell volume,
                                    ! equation constant term
        REAL(KIND=8) :: dt, V2nd, term
    END TYPE MESHTYPE

CONTAINS
    SUBROUTINE init_mesh(mesh)
        ! Mesh points (derived data type)
        TYPE(MESHTYPE), TARGET :: mesh(1:IMAX, 1:JMAX)
        ! Pointer for mesh points
        TYPE(MESHTYPE), POINTER :: m
        INTEGER :: i, j

        DO j = 1, JMAX
            DO i = 1, IMAX
                m => mesh(i, j)
                ! 'p' points to 'mesh', i is variable in derived data type
                    ! accessed by '%'
                ! MAKE SQUARE GRID
                m%i = i
                m%j = j
                ! ROTATE GRID
                m%xp = COS( 0.5D0 * pi * DFLOAT(IMAX - i) / DFLOAT(IMAX - 1) )
                m%yp = COS( 0.5D0 * pi * DFLOAT(JMAX - j) / DFLOAT(JMAX - 1) )

                m%x = m%xp * COS(rot) + (1.D0 - m%yp ) * SIN(rot)
                m%y = m%yp * COS(rot) + (m%xp) * SIN(rot)
            END DO
        END DO
    END SUBROUTINE init_mesh

    SUBROUTINE init_temp(m, T)
        ! Initialize temperature across mesh
        ! m --> pointer for mesh vector
        ! T --> initial temperature profile
        TYPE(MESHTYPE), INTENT(INOUT) :: m
        REAL(KIND=8) :: T
        ! SET MESH POINTS WITH INITIAL TEMPERATURE PROFILE
        m%T = T
    END SUBROUTINE init_temp
END MODULE MESHMOD

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! CELLS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran

MODULE CELLMOD
    ! Initialize finite volume cells and do associated calculations
    USE MESHMOD

    IMPLICIT NONE
    PUBLIC

    TYPE CELLTYPE
        ! Cell volumes
        REAL(KIND=8) :: V
        ! Second-derivative weighting factors for alternative distribution scheme
        REAL(KIND=8) :: yPP, yNP, yNN, yPN
        REAL(KIND=8) :: xNN, xPN, xPP, xNP
    END TYPE CELLTYPE

CONTAINS
    SUBROUTINE init_cells(mesh, cell)
        ! cell --> derived data type containing cell info
        ! mesh --> derived data type containing mesh point info
        TYPE(MESHTYPE) :: mesh(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE), TARGET :: cell(1:IMAX-1,1:JMAX-1)
        INTEGER :: i, j

        DO j = 1, JMAX-1
            DO i = 1, IMAX-1
                ! CALC CELL VOLUMES
                    ! (length in x-dir times length in y-dir)
                cell(i,j)%V = ( (mesh(i+1,j)%xp - mesh(i,j)%xp) ) &
                                    * ( mesh(i,j+1)%yp - mesh(i,j)%yp )
            END DO
        END DO
    END SUBROUTINE init_cells

    SUBROUTINE calc_2nd_areas(m, cell)
        ! calculate areas for secondary fluxes.
        ! cell --> derived data type with cell data, target for c
        ! m --> mesh points
        TYPE(MESHTYPE), TARGET :: m(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE), TARGET :: cell(1:IMAX-1, 1:JMAX-1)
        TYPE(CELLTYPE), POINTER :: c
        INTEGER :: i, j
        ! Areas used in alternative scheme to get fluxes for second-derivative
        REAL(KIND=8) :: Ayi, Axi, Ayj, Axj
        ! Areas used in counter-clockwise trapezoidal integration to get
        ! x and y first-derivatives for center of each cell (Green's thm)
        REAL(KIND=8) :: Ayi_half, Axi_half, Ayj_half, Axj_half

        ! CALC CELL AREAS
        Axi(i,j) = m(i,j+1)%x - m(i,j)%x
        Axj(i,j) = m(i+1,j)%x - m(i,j)%x
        Ayi(i,j) = m(i,j+1)%y - m(i,j)%y
        Ayj(i,j) = m(i+1,j)%y - m(i,j)%y

        Axi_half(i,j) = ( Axi(i+1,j) + Axi(i,j) ) * 0.25D0
        Axj_half(i,j) = ( Axj(i,j+1) + Axj(i,j) ) * 0.25D0
        Ayi_half(i,j) = ( Ayi(i+1,j) + Ayi(i,j) ) * 0.25D0
        Ayj_half(i,j) = ( Ayj(i,j+1) + Ayj(i,j) ) * 0.25D0

        ! Actual finite-volume scheme equation parameters
        DO j = 1, JMAX-1
            DO i = 1, IMAX-1
                c => cell(i, j)
                ! (NN = 'negative-negative', PN = 'positive-negative',
                    ! see how fluxes are summed)
                c%xNN = ( -Axi_half(i,j) - Axj_half(i,j) )
                c%xPN = (  Axi_half(i,j) - Axj_half(i,j) )
                c%xPP = (  Axi_half(i,j) + Axj_half(i,j) )
                c%xNP = ( -Axi_half(i,j) + Axj_half(i,j) )

                c%yPP = (  Ayi_half(i,j) + Ayj_half(i,j) )
                c%yNP = ( -Ayi_half(i,j) + Ayj_half(i,j) )
                c%yNN = ( -Ayi_half(i,j) - Ayj_half(i,j) )
                c%yPN = (  Ayi_half(i,j) - Ayj_half(i,j) )
            END DO
        END DO
    END SUBROUTINE calc_2nd_areas

    SUBROUTINE calc_constants(mesh, cell)
        ! Calculate constants for a given iteration loop.  This way,
        ! they don't need to be calculated within the loop at each iteration
        TYPE(MESHTYPE), TARGET :: mesh(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE), TARGET :: cell(1:IMAX-1, 1:JMAX-1)
        INTEGER :: i, j
        DO j = 2, JMAX - 1
            DO i = 2, IMAX - 1
                ! CALC TIMESTEP FROM CFL
                mesh(i,j)%dt = ((CFL * 0.5D0) / alpha) * cell(i,j)%V ** 2 &
                                / ( (mesh(i+1,j)%xp - mesh(i,j)%xp)**2 &
                                    + (mesh(i,j+1)%yp - mesh(i,j)%yp)**2 )
                ! CALC SECONDARY VOLUMES
                ! (for rectangular mesh, just average volumes of the 4 cells
                !  surrounding the point)
                mesh(i,j)%V2nd = ( cell(i,j)%V &
                                    + cell(i-1,j)%V + cell(i,j-1)%V &
                                    + cell(i-1,j-1)%V ) * 0.25D0
                ! CALC CONSTANT TERM
                ! (this term remains constant in the equation regardless of
                !  iteration number, so only calculate once here,
                !  instead of in loop)
                mesh(i,j)%term = mesh(i,j)%dt * alpha / mesh(i,j)%V2nd
            END DO
        END DO
    END SUBROUTINE calc_constants
END MODULE CELLMOD

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! CALCULATE TEMPERATURE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE TEMPERATURE
    ! Calculate and store new temperature distribution for given iteration
    USE MESHMOD
    USE CELLMOD

    IMPLICIT NONE
    PUBLIC

CONTAINS
    SUBROUTINE derivatives(m, c)
        ! Calculate first and second derivatives for finite-volume scheme
        TYPE(MESHTYPE), INTENT(INOUT) :: m(1:IMAX, 1:JMAX)
        TYPE(CELLTYPE), INTENT(INOUT) :: c(1:IMAX-1, 1:JMAX-1)
        ! Areas for first derivatives
        REAL(KIND=8) :: Ayi, Axi, Ayj, Axj
        ! First partial derivatives of temperature in x and y directions
        REAL(KIND=8) :: dTdx, dTdy
        INTEGER :: i, j

        ! CALC CELL AREAS
        Axi(i,j) = m(i,j+1)%x - m(i,j)%x
        Axj(i,j) = m(i+1,j)%x - m(i,j)%x
        Ayi(i,j) = m(i,j+1)%y - m(i,j)%y
        Ayj(i,j) = m(i+1,j)%y - m(i,j)%y

        ! RESET SUMMATION
        m%Ttmp = 0.D0

        DO j = 1, JMAX - 1
            DO i = 1, IMAX - 1
                ! CALC FIRST DERIVATIVES
                dTdx = + 0.5d0 &
                            * (( m(i+1,j)%T + m(i+1,j+1)%T ) * Ayi(i+1,j) &
                            -  ( m(i,   j)%T + m(i,   j+1)%T ) * Ayi(i,   j) &
                            -  ( m(i,j+1)%T + m(i+1,j+1)%T ) * Ayj(i,j+1) &
                            +  ( m(i,   j)%T + m(i+1,   j)%T ) * Ayj(i,   j) &
                                ) / c(i,j)%V
                dTdy = - 0.5d0 &
                            * (( m(i+1,j)%T + m(i+1,j+1)%T ) * Axi(i+1,j) &
                            -  ( m(i,   j)%T + m(i,   j+1)%T ) * Axi(i,   j) &
                            -  ( m(i,j+1)%T + m(i+1,j+1)%T ) * Axj(i,j+1) &
                            +  ( m(i,   j)%T + m(i+1,   j)%T ) * Axj(i,   j) &
                                ) / c(i,j)%V

                ! Alternate distributive scheme second-derivative operator.
                m(i+1,   j)%Ttmp = m(i+1,   j)%Ttmp + m(i+1,   j)%term * ( c(i,j)%yNN * dTdx + c(i,j)%xPP * dTdy )
                m(i,     j)%Ttmp = m(i,     j)%Ttmp + m(i,     j)%term * ( c(i,j)%yPN * dTdx + c(i,j)%xNP * dTdy )
                m(i,   j+1)%Ttmp = m(i,   j+1)%Ttmp + m(i,   j+1)%term * ( c(i,j)%yPP * dTdx + c(i,j)%xNN * dTdy )
                m(i+1,j+1)%Ttmp = m(i+1,j+1)%Ttmp + m(i+1,j+1)%term * ( c(i,j)%yNP * dTdx + c(i,j)%xPN * dTdy )
            END DO
        END DO
    END SUBROUTINE derivatives
END MODULE TEMPERATURE
```

Listing 5: Modules used by solver

```fortran
! MAE 267
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  This module creates a grid and temperature file in
!               the plot3D format for steady state solution

MODULE plot3D_module
    USE CONSTANTS
    USE MESHMOD
    IMPLICIT NONE

    ! VARIABLES
    INTEGER :: gridUnit  = 30   ! Unit for grid file
    INTEGER :: tempUnit = 21    ! Unit for temp file
    REAL(KIND=8) :: tRef = 1.D0          ! tRef number
    REAL(KIND=8) :: dum = 0.D0          ! dummy values
    INTEGER :: nBlocks = 1      ! number of blocks

    CONTAINS
    SUBROUTINE plot3D(mesh)
        IMPLICIT NONE

        TYPE(MESHTYPE) :: mesh(1:IMAX, 1:JMAX)
        INTEGER :: i, j

        ! FORMAT STATEMENTS
        10     FORMAT(I10)
        20     FORMAT(10I10)
        30     FORMAT(10E20.8)

!         ! OPEN FILES
!         OPEN(UNIT=gridUnit,FILE='grid.xyz',FORM='formatted')
!         OPEN(UNIT=tempUnit,FILE='temperature.dat',FORM='formatted')

!         ! WRITE TO GRID FILE (FORMATTED)
!         WRITE(gridUnit,10) nBlocks
!         WRITE(gridUnit,20) IMAX,JMAX
!         WRITE(gridUnit,30) ((mesh(i,j)%x,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%y,i=1,IMAX),j=1,JMAX)

!         ! WRITE TO TEMPERATURE FILE
!          WRITE(tempUnit,10) nBlocks
!          WRITE(tempUnit,20) IMAX,JMAX
!          WRITE(tempUnit,30) tRef,dum,dum,dum
!          WRITE(tempUnit,30) ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), &
!                             ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX)

        ! OPEN FILES
        OPEN(UNIT=gridUnit,FILE='grid.xyz',FORM='unformatted')
        OPEN(UNIT=tempUnit,FILE='temperature.dat',FORM='unformatted')

        ! WRITE TO GRID FILE (UNFORMATTED)
            ! (Paraview likes unformatted better)
        WRITE(gridUnit) nBlocks
        WRITE(gridUnit) IMAX,JMAX
        WRITE(gridUnit) ((mesh(i,j)%x,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%y,i=1,IMAX),j=1,JMAX)

        ! WRITE TO TEMPERATURE FILE
            ! When read in paraview, 'density' will be equivalent to temperature
        WRITE(tempUnit) nBlocks
        WRITE(tempUnit) IMAX,JMAX
        WRITE(tempUnit) tRef,dum,dum,dum
        WRITE(tempUnit) ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), &
                           ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX), ((mesh(i,j)%T,i=1,IMAX),j=1,JMAX)

        ! CLOSE FILES
        CLOSE(gridUnit)
        CLOSE(tempUnit)
    END SUBROUTINE plot3D
END MODULE plot3D_module
```

Listing 6: PLOT3D file output module (compatible with ParaView)