NONLINEAR ACOUSTIC STREAMING IN STRAIGHT AND TAPERED TUBES

A Dissertation in
Acoustics
by
Brian C. Tuttle

© 2012 Brian C. Tuttle

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2012
The dissertation of Brian C. Tuttle was reviewed and approved* by the following:

Victor W. Sparrow  
Professor of Acoustics  
Interim Chair of the Graduate Program in Acoustics  
Dissertation Advisor, Chair of Committee

Anthony A. Atchley  
Professor of Acoustics  
Associate Dean of the College of Engineering

Robert M. Keolian  
Associate Professor of Acoustics

Philip J. Morris  
Boeing/A.D. Welliver Professor of Aerospace Engineering

*Signatures are on file in the Graduate School.
Abstract

In thermoacoustic and Stirling devices such as the pulse-tube refrigerator, efficiency is diminished by the formation of a second-order mean velocity known as Rayleigh streaming. This flow emerges from the interaction of the working gas with the wall of the tube in a thin boundary layer. Recent studies have suggested that streaming velocity can be decreased in a tube by tapering it slightly. This research investigates that claim through the development of a numerical model of Rayleigh streaming in variously tapered tubes. It is found that the numerical simulation of streaming in a straight tube compares well with theory, and the application of different thermal boundary conditions at the tube wall shows that for pressurized helium, inner streaming vortices which appear near an adiabatic tube wall do not develop near an isothermal wall. An order analysis indicates that the temperature dependence of viscosity and thermal conductivity contributes appreciably to an accurate numerical model of streaming. Comparison of Rayleigh streaming in tapered tubes shows the effects of taper angle on the circulation and velocity of the mean flow.
Table of Contents

List of Figures viii
List of Tables xi
List of Symbols xii
Acknowledgments xviii

Chapter 1
Introduction 1
1.1 Types of Streaming ............................................. 1
1.1.1 Boundary-Layer Interactions ................................. 2
1.1.2 Bulk Dissipation ............................................. 2
1.1.3 Traveling-Wave Phasing .................................. 3
1.1.4 Jet Dynamics ................................................. 4
1.2 Previous Work .................................................... 4
1.2.1 Early Observations and Theory ............................. 5
1.2.2 Heat Conduction and Temperature Gradients ............. 7
1.2.3 Fluid Inertia ................................................... 9
1.2.4 Variations in Tube Radius ................................ 11
1.2.5 Other Contributions ........................................ 14
1.3 Motivation and Goals .......................................... 15
1.3.1 Challenges ..................................................... 16
1.4 Thesis Overview ................................................ 17

Chapter 2
Numerical Model 19
2.1 Experimental Basis .............................................. 19
2.2 Model Geometry ............................................... 21
2.3 Model Equations ............................................... 22
Chapter 3  
Numerical Implementation  
3.1 Grid Refinement and Generation  
3.1.1 Grid-Refinement Method  
3.1.2 Grid Generation  
3.1.2.1 Cylindrical Coordinates  
3.1.2.2 Spherical Coordinates  
3.1.3 Conformal Mapping  
3.2 Finite-Difference Scheme  
3.2.1 Split-Step Method  
3.2.2 Second Derivatives and Source Terms  
3.3 Boundary Treatment  
3.3.1 One-Sided Axisymmetric Boundary  
3.3.1.1 Dirichlet Condition for an Offset Boundary  
3.3.1.2 Neumann Condition for an Offset Boundary  
3.4 Multi-Processor Computation  
3.5 Summary  

Chapter 4  
Simulation Method  
4.1 Excitation of Standing Waves  
4.1.1 Line Source  
4.1.2 Two-Source Method  
4.1.3 Reflected-Source Method  
4.2 Model Tube Dimensions  
4.3 Calculation of Streaming Velocity  
4.4 Normalization  
4.5 Summary  

Chapter 5  
Results  
5.1 Model Verification  
5.2 Adiabatic vs. Isothermal Boundary Condition at the Tube Wall  
5.3 Effects of High-Order Terms on Streaming Development
### List of Figures

1.1 Boundary layer streaming in a Kundt’s tube ........................................... 2  
1.2 Dissipation-driven streaming from an ultrasonic source in an unbounded medium ................................................................. 3  
1.3 Jet-driven streaming ................................................................................ 3  
1.4 Diagram of inner and outer streaming: $x =$ distance along the surface, $y =$ distance above the surface, $\lambda =$ acoustic wavelength ........................................... 7  

2.1 Cylindrical and spherical coordinate geometries ..................................... 22  
2.2 Cylindrical coordinates ............................................................................. 25  
2.3 Spherical coordinates ................................................................................ 33  

3.1 Hyperbolic tangent function mapping regular grid $\Xi_N$ to refined grid $X_N$ ................................................................................................. 39  
3.2 Grid near the axis showing an offset of $\frac{\Delta r}{2}$ to avoid a singularity at $r = 0$ ................................................................................................. 42  
3.3 Axisymmetric boundary ............................................................................. 51  
3.4 Decomposition of computational domain into $n$ nodes, labeled 0 to $n-1$ ................................................................................................. 54  
3.5 Message passing between adjacent subdomains A and B ....................... 54  

4.1 Normalized velocity and pressure of a standing wave generated by two line sources at $z = 0.25\lambda$ and $z = 1.75\lambda$. The curves are shown at times of maximum excursion indicated by the phase angle $\phi$. ................................................... 59  
4.2 Normalized velocity and pressure of a standing wave generated by a single source at $z/\lambda = 0.25$ in a straight tube with one rigid end and one non-reflecting end. The curves are shown at times of maximum excursion indicated by the phase angle $\phi$. ......................... 60  
4.3 Normalized velocity and pressure of a standing wave generated by a single source at $r/\lambda = 0.56$ in a tapered tube with one rigid end and one non-reflecting end. The curves are shown at times of maximum excursion indicated by the phase angle $\phi$. ......................... 61
4.4 Parameters defining tapered tube geometry. The rigid end is at $r = R_1$, and the non-reflecting end is on the opposite side at $r = R_1 + L$. 

5.1 Streaming circulation in a two-wavelength-long tube with rigid adiabatic wall, averaged over 10 periods from cycles 540 to 550.

5.2 Profile of axial streaming velocity comparing the development of the numerical simulation to the theoretical prediction from Schuster and Matz.

5.3 Transverse profile of axial Lagrangian streaming velocity. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.

5.4 Eulerian and velocity transport components of axial streaming velocity.

5.5 Eulerian and velocity transport components of axial streaming velocity near the tube wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.

5.6 Axial Lagrangian streaming profiles of varying order near an adiabatic wall.

5.7 Axial Lagrangian streaming profiles of varying order near an adiabatic wall.

5.8 Streaming circulation in a cylindrical tube with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.

5.9 Streaming circulation in a 0.4° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 30 to 40.

5.10 Streaming circulation in a 0.8° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 30 to 40.

5.11 Streaming circulation in a 1.2° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.

5.12 Streaming circulation in a 1.6° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.

5.13 Axial Lagrangian streaming profile near adiabatic wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.
5.14 Profile of axial Lagrangian streaming velocity adjacent to an adiabatic wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.

5.15 Profile of axial Lagrangian streaming velocity adjacent to an isothermal wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.

5.16 Positive axial streaming velocity contribution (solid line) to the spatially averaged velocity $\langle u_{st}^+ \rangle$. Only the upper 10% of the radial profile is shown.

5.17 Comparison of mean positive streaming velocity in the axial direction as an indicator of relative circulation strength between variously tapered tubes.

C.1 Streaming velocity profiles calculated using three grids of varying radial resolution. Positions A and B are indicated by the vertical lines.

C.2 Streaming velocity profiles calculated over three grids. The extrapolated solution is shown for positions A and B.

C.3 Streaming velocity profiles calculated over three grids. The extrapolated solution is shown for positions A and B.
List of Tables

4.1 Parameters of Tube Geometries ............................................. 65

C.1 Streaming velocities calculated over three grids for positions A and
   B found through cubic spline interpolation .................................. 127

C.2 Relative error magnitudes and Grid Convergence Indices for points
   A and B found through cubic spline interpolation ........................ 127

C.3 Streaming velocities calculated over three grids for positions A and
   B found through linear interpolation ......................................... 128

C.4 Relative error magnitudes and Grid Convergence Indices for points
   A and B found through linear interpolation .................................. 129
List of Symbols

A  grid refinement parameter, p. 40, cross-sectional area [m²]

A  generic vector, p. 25, array of coefficients, p. 114

b  exponent describing temperature dependent viscosity, p. 20

B  grid refinement parameter, p. 40

B  array of coefficients, p. 114

c₀  speed of sound [m/s]

cₚ  specific heat at constant pressure [J/(kg · K)]

cᵥ  specific heat at constant volume [J/(kg · K)]

C  Courant number, p. 46

C₁, C₂  reference temperatures in Sutherland’s formula [K], p. 23

C  vector of inhomogeneous terms, p. 114

d  direction of single-step spatial difference operation, p. 47

e  specific internal energy [J/kg], p. 23

E_N(x)  discretization error, p. 98

F  vector of axial flux terms

g(ε)  gauge function, p. 98

g  acceleration due to gravity [m/s²]
vector of radial flux terms

\( i \) computational grid index in the \( \xi \)-direction

\( j \) computational grid index in the \( \eta \)-direction

\( j_0(x), j_1(x) \) spherical Bessel functions, p. 64

\( J \) Jacobian of grid transformation

\( k \) wavenumber \([m^{-1}]\)

\( l_i \) left eigenvectors, p. 115

\( L \) tube length \([m]\)

\( L_\eta, L_\xi \) finite-difference operators in the \( \eta \)- and \( \xi \)-direction, p. 47

\( \mathcal{L}_i \) boundary conditions, p. 115

\( \mathcal{L}'_i \) boundary conditions to first order, p. 116

\( \mathcal{L}''_i \) boundary conditions to second order, p. 116

\( \dot{m} \) mass flux \([kg/s]\), p. 68

\( n_0(x), n_1(x) \) spherical Neumann functions, p. 64

\( O \) Landau symbol, p. 98

\( p \) pressure \([Pa]\),

order of convergence, App. C

\( P_0 \) acoustic pressure amplitude \([Pa]\)

\( \mathbf{q} \) heat-flux vector \([W/m^2]\)

\( Q \) heat load \([J]\)

\( r \) radial distance in cylindrical coordinates \([m]\), p. 25,
radial distance in spherical coordinates \([m]\), p. 33

\( R \) specific gas constant, p. 23,
tube radius \([m]\)

\( R_1 \) distance from spherical origin to narrow end of tapered tube \([m]\),
p. 63
\( R_{ST} \) distance from spherical origin to streaming profile in tapered tube [m], p. 63

\( R_s \) distance from spherical origin to source in tapered tube [m], p. 63

\( Re_{nt} \) nonlinear Reynolds number, p. 10

\( S \) source vector, p. 57

\( S_p \) time derivative of pressure source [Pa/s], p. 57

\( S_0, S_1 \) ratios of regular to refined grid spacing, p. 40

\( SF \) vector of axial source terms

\( SG \) vector of radial source terms

\( t \) time [s]

\( T \) temperature [K]

\( u \) grid refinement function, p. 40,
   general model variable, p. 51,
   axial component of acoustic velocity [m/s]

\( u_{st} \) axial component of Lagrangian streaming velocity [m/s], p. 66

\( U \) acoustic velocity amplitude [m/s]

\( U_{Rep} \) representative acoustic velocity [m/s], p. 68

\( U_N \) normalization velocity [m/s], p. 68

\( v \) radial component of acoustic velocity [m/s]

\( v_{st} \) radial component of Lagrangian streaming velocity [m/s], p. 66

\( \mathbf{v} \) velocity vector [m/s]

\( \mathbf{v}_{st} \) Lagrangian streaming velocity vector [m/s], p. 66

\( \mathbf{v}_T \) velocity transform vector [m/s], p. 66

\( \mathbf{w} \) solution vector of primitive variables

\( x \) generic axial physical grid component, p. 44
$x_\xi, x_\eta$ inverse metrics of grid transformation, p. 45

$X$ generic refined physical grid, p. 39

$y$ generic radial physical grid component, p. 44

$y_\xi, y_\eta$ inverse metrics of grid transformation, p. 45

$z$ height in cylindrical coordinates [m], p. 25

$z_s$ axial position of source in straight tube [m], p. 57

$Z_{ST}$ axial position of streaming profile in straight tube [m], p. 67

$\alpha$ arc length [m], p. 33,
boundary value, p. 52,
width of Gaussian function, p. 57

$\gamma$ ratio of specific heats, p. 20

$\delta_e$ thermal penetration depth [m]

$\delta_\nu$ viscous penetration depth [m], p. 10

$\delta_{ij}$ Kronecker delta function, p. 24

$\Delta$ discrete step size

$\Delta y$ grid refinement variable, p. 40

$\epsilon$ small ordering parameter, p. 98

$\varepsilon$ Mach number, p. 102

$\eta$ Stokes number, p. 104,
ardial computational grid variable, p. 41

$\eta_x, \eta_y$ metrics of grid transformation, p. 45

$\theta$ phase angle between acoustic pressure and velocity [rad], p. 20,
$polar angle in cylindrical coordinates [rad], p. 25,
polar angle in spherical coordinates [rad], p. 33

$\Theta$ maximum polar angle of tapered tube [rad], p. 43

$\kappa$ thermal conductivity [W/(m · K)], p. 23
\[ \lambda \] dilatational viscosity \([\text{kg}/(\text{m} \cdot \text{s})]\), p. 23, wavelength \([\text{m}]\)

\[ \lambda_i \] eigenvalues, p. 114

\[ \mu \] shear viscosity \([\text{kg}/(\text{m} \cdot \text{s})]\), p. 23

\[ \xi \] axial computational grid variable, p. 41

\[ \xi_x, \xi_y \] metrics of grid transformation, p. 45

\[ \Xi \] generic computational grid, p. 39

\[ \rho \] density \([\text{kg}/\text{m}^3]\)

\[ \sigma \] Prandtl number, p. 20, grid refinement ratio, App. C

\[ \sigma_{ij} \] component of stress tensor \([\text{Pa}]\), p. 24

\[ \phi \] taper angle \([\text{rad}]\), phase of sinusoidal oscillation \([\text{rad}]\)

\[ \phi_{ij} \] component of rate-of-shear tensor \([\text{s}^{-1}]\), p. 24

\[ \varphi \] azimuthal angle in spherical coordinates \([\text{rad}]\), p. 33

\[ \Phi \] generic scalar, p. 25, velocity potential, p. 100

\[ \omega \] angular frequency \([\text{rad}/\text{s}]\)

**Subscripts**

\[ N \] number of elements in a sequence, p. 98, normalized, Ch. 3

\[ ch \] characteristic value

\[ r \] \(\hat{r}\)-direction, axial in spherical coordinates, radial in cylindrical coordinates

\[ z \] \(\hat{z}\)-direction, axial in cylindrical coordinates

\[ \theta \] \(\hat{\theta}\)-direction, radial in spherical coordinates
0 zero-order, ambient quantity
1 first-order, acoustic quantity
2 second-order, nonlinear quantity

**Superscripts**

\( a' \) fluctuating part of quantity \( a \)

\( \hat{a} \) coordinate vector \( a \), Ch. 2,
   nondimensional variable \( a \), App. A

\( a^+ \) non-negative part of quantity \( a \), Section 5.5
Acknowledgments

Such endeavors are not accomplished without the support and guidance of a host of family, friends, and mentors. First and foremost I am grateful to my advisor, Dr. Vic Sparrow, for his patient counsel and to my committee members for their constructive critiques of my work. To my family I am indebted for their endless encouragement and support. Many thanks to friends both near and far, past and present who have illuminated this journey with their laughter and wisdom. And thanks especially to Alexandra for all of her love and encouragement.
Dedication

This work is dedicated to Patience and to all those who exercise that virtue.
Acoustic streaming is a circulation generated by acoustic oscillations in a fluid medium. In most cases the circulation arises from small second-order stresses in the fluid that normally have negligible effect in linear acoustics but come into play with oscillations of sufficiently high amplitude. Over many oscillations these stresses accumulate to propel a steady flow of mass within the fluid. In the context of thermoacoustic refrigeration, streaming represents an unpredictable dynamic. The goal of thermoacoustic refrigeration is to move heat from one area of a resonant enclosure to another by way of strong acoustic oscillations. Any steady mass flow from streaming could cause heat convection in a direction that is counterproductive to this effort. Recent studies have suggested that streaming velocity may be decreased by varying the geometry of the resonant enclosure [1, 2, 3, 4, 5]. The purpose of this research is to construct a time-accurate numerical model to compare the effects of tapered versus straight waveguides on acoustic streaming.

1.1 Types of Streaming

There are many mechanisms that can give rise to streaming velocity in an acoustic field. In addition to scenarios where streaming occurs in an unbounded medium or circulates within enclosures, acoustic streaming can develop near vibrating objects or around stationary obstacles in an oscillating flow. For instance, special attention has been given to streaming around bubbles and other compressible structures for studies of biological processes [6, 7, 8]. The present study concentrates on streaming within resonant enclosures and does not address flows around objects. This
circulation can be subdivided further according to the classification of streaming generation mechanisms proposed by Boluriaan and Morris [9]. These are streaming driven by boundary-layer interactions, bulk dissipation, traveling-wave phasing, and jet dynamics. All four types are described here because in practice streaming circulation may result from a combination of these mechanisms.

### 1.1.1 Boundary-Layer Interactions

The main focus of this research is streaming driven by boundary-layer interactions in tubes of circular cross section, such as a Kundt’s tube. This circulation, known as Rayleigh streaming, develops from the influence of viscous stresses within the boundary layer when a standing wave is excited within the tube. As illustrated in Figure 1.1, the main streaming velocity is directed along the wall away from the velocity antinodes and toward the nodes. Rayleigh streaming is discussed in detail throughout the following sections and chapters.

![Figure 1.1. Boundary layer streaming in a Kundt’s tube](image)

### 1.1.2 Bulk Dissipation

Fluid flow that develops in the presence of a high-intensity sound beam is known as bulk-dissipation-driven streaming, Eckart streaming, or “quartz wind.” It is set into motion by gradients in the momentum flux that are caused by a dissipation of energy along the sound beam [10]. Eckart provided the first mathematical analysis of this type of streaming [11]. It can occur with no boundary interactions and typically results from ultrasonic excitation as shown in the diagram of Figure 1.2.
1.1.3 Traveling-Wave Phasing

For an acoustic field that is not a pure standing wave there is a nonzero mass flux component that propagates mean mass flow in the direction of energy flow. In a standing wave the phase between first-order density and velocity completely cancels and eliminates the mass flux vector. Otherwise this vector is present and finds its maximum magnitude when density and velocity are in phase as in the case of a traveling wave. This mass flux component provides a mechanism for streaming that does not depend on viscosity for momentum flux through energy dissipation or shear stress. Traveling-wave streaming can occur in an inviscid fluid as discussed by Boluriaan and Morris [9]. It is one of the mechanisms for streaming within thermoacoustic devices that rely on traveling waves, and in that context it is referred to as Gedeon streaming [12].
1.1.4 Jet Dynamics

Jet-driven streaming is propelled by fluid exiting and entering an abrupt constric-
tion or orifice. The mean flow develops from a difference in behavior between the
outflow and the inflow. In the outward direction, shown in Figure 1.3a, the shear
layer separates from the surface surrounding the orifice, and the flow tends to con-
tinue in the direction that it is ejected. On the inflow portion of the cycle shown
in Figure 1.3b, fluid enters the constriction from all directions as in a sink. The
combination of these actions averaged over many cycles results in a circulation
similar to that shown in Figure 1.3c. Fluid gathers around the perimeter of the
opening and flows outward from it.

The magnitude of the velocity in jet-driven streaming is markedly higher than
in the other types mentioned here. Jet-driven streaming velocity is on the order of
acoustic particle velocity and can sometimes exceed it. While the velocity of other
streaming mechanisms scales as the square of acoustic particle velocity, jet-driven
streaming velocity scales linearly. This suggests that jet-driven streaming may
be considered a first-order effect, whereas Rayleigh, Eckart, and traveling-wave
streaming are derived from second-order interactions [9].

1.2 Previous Work

Many authors who have written about acoustic streaming have provided excellent
overviews of its history and state of the art at the time of publication. Wesley
Nyborg gave one such review including discussions of near-boundary streaming,
quartz wind, streaming around barriers, and the development of streaming theory
up to 1965 [7]. A later review by Boluriaan and Morris, “Acoustic Streaming:
From Rayleigh to Today,” thoroughly covers the known types of streaming and
the efforts to investigate the underlying mechanisms [9]. Most recently Aktas and
Ozgumus summarized a comprehensive list of publications on Rayleigh streaming
in channels and tubes [13].

The following review of previous research in acoustic streaming focuses on cir-
culation due to boundary-layer interactions in tubes, or Rayleigh streaming. It
begins chronologically with early observations and theory and continues with de-
criptions of three common themes in the literature: the effects of heat conduction and temperature gradients, fluid inertia, and variations in tube radius. Other important work is cited in the final section.

1.2.1 Early Observations and Theory

In 1831 Michael Faraday made the earliest recorded observations of steady currents of air caused by acoustic vibrations [14]. He noted earlier experiments by Chladni in which sand tended to collect in the nodal lines of vibrating plates, and by Savart who observed that very fine powder when introduced onto a vibrating surface would hover above the areas of strongest vibration, eventually settling on those places. Faraday proceeded with an array of experiments demonstrating that circulating currents of air were induced above the vibrating surfaces which motivated the particles into their various patterns. Dvořák in 1876 made a similar observation of the movement of dust in sounding tubes [15]. When a standing wave was excited in a tube, the dust tended to travel toward the velocity nodes and away from the antinodes. It was this report that Lord Rayleigh cited in his 1883 analysis of boundary-layer-driven streaming in standing-wave resonators, henceforth known as Rayleigh streaming [16].

Rayleigh considered the circulation of air between two parallel plates when the air oscillates in a direction parallel to those surfaces. He also provided a variation on the analysis applicable to cylindrical tubes which Schuster and Matz later elaborated to describe streaming velocity in a Kundt’s tube [17]. Using a method of successive approximations Rayleigh developed a theory that predicted streaming motion and showed that the circulation arose from second-order nonlinear interactions in the fluid near the surfaces. This method of successive approximations first solves the equations of motion to first order and then uses that solution to obtain a result valid to second order, the second-order mean velocity being the relevant quantity to describe streaming. For this analysis Rayleigh neglected heat conduction, considering its effect to be minimal. He assumed that the fluid was compressible and limited his analysis to a geometry where the tube radius was much smaller than the acoustic wavelength and the viscous penetration depth was much smaller than the tube radius. Although his result accurately imposes a no-
slip condition at the surface and predicts zero streaming velocity in a direction parallel to it, the streaming velocity in the normal direction at the surface is allowed a finite value. This draws into question the validity of the solution within a viscous penetration depth of the surface [9]. Outside of this depth Rayleigh’s solution provides a valid and useful description of acoustic streaming.

It was not until many years later in 1931 that Andrade sought to observe Rayleigh’s theory in a controlled experiment [18]. Noting that Dvořák’s evidence of dust motion lacked the steady standing wave that Rayleigh’s theory held necessary for fully developed streaming, Andrade set about to create such an environment using the latest in thermionic valve and electromagnetic diaphragm technology, i.e. an amplifier and loudspeaker. He took photographs of the patterns formed by smoke particles circulating with the vibrating air in a glass tube, which was enclosed by a water-filled jacket to maintain steady temperature. He found that the patterns compared favorably with Rayleigh’s predictions. He also observed that at high frequencies the particles were less inclined to follow the movement of the air and instead acted as obstacles likely to cause extra vortices in the flow due to their presence. Thus he recognized a main challenge in the accurate measurement of acoustic streaming, that the observation technique not interfere with the motion being observed.

Further developments in the theory of streaming followed with a variety of refinements to Rayleigh’s analysis. In 1932 Schlichting examined streaming within the boundary layer of incompressible oscillating flow over a flat plate. He predicted vortices adjacent to the surface with a depth of approximately 1.9 times the viscous penetration depth and a width of 1/4 the acoustic wavelength [9]. This thin vortex, called inner streaming, rotates counter to the neighboring outer streaming circulation predicted by Rayleigh. Inner and outer streaming are illustrated schematically in Figure 1.4.

Westervelt developed a generalized theory of streaming that is valid for rotational as well as irrotational motion. Unlike Rayleigh’s theory, Westervelt’s analysis does not assume solenoidal motion of first-order velocity and is valid for an inhomogeneous medium. He rederived Rayleigh’s calculation of streaming velocity for the more general case and showed that it agreed with Rayleigh’s solution when the restriction of solenoidal motion was applied [19].
In his 1978 lecture delivered to the Institute of Acoustics, Lighthill spoke about the distinction between Eulerian and Lagrangian motion [10]. In the Eulerian frame of reference velocity is described at fixed points in space as the fluid passes by. This results in a depiction of the fluid motion as a velocity field at a specific moment in time. The Lagrangian frame of reference follows the motion of a small parcel of fluid as it moves through space. Considering many small parcels over time gives a depiction of the fluid motion as a set of particle paths. In linear acoustic theory these two frames of reference show the same motion, but when second-order effects are taken into account, as in the calculation of streaming, they become distinct. It is Lagrangian velocity that corresponds to the motion observed when smoke particles are used to reveal streaming circulation.

In 1989 Sharpe et al. made use of this distinction when they measured the circulation of smoke in a cylindrical tube by a new technique, particle image velocimetry. They isolated the experiment thermally with polyurethane foam, covering all but the working area. Their resulting observations of outer streaming showed good agreement with Rayleigh’s prediction [20].

1.2.2 Heat Conduction and Temperature Gradients

Early theoretical analyses of Rayleigh streaming neglected the role of heat conduction and variations in temperature. Nikolaus Rott addressed this issue in his
1974 paper where he included the effects of thermal conduction in the calculation of streaming velocity [21]. He considered a tube with isothermal walls where the mean temperature was allowed to vary along the length of the tube. He also allowed the coefficients of viscosity and thermal conductivity to depend on the acoustically fluctuating temperature. Thompson and Atchley noted that the streaming velocity calculated for air in the axial direction along the center of the tube was 3% higher using Rott’s equations than that calculated by Schuster and Matz’s equations which do not include thermal effects [22].

Qi et al. arrived at a similar expression for streaming velocity in a tube. The focus of their research was to clarify the importance of fluid compressibility in the analysis of streaming near a boundary [23]. They included heat conduction but considered the coefficients of viscosity and thermal conductivity to be independent of temperature [24]. Consequently their equations of streaming velocity differ from Rott’s by one term, and when applied to a calculation of streaming in air in a tube, they produced an axial velocity approximately 13% higher than that predicted by Schuster and Matz [22].

In research that investigated streaming in wide and narrow channels as well as in cylindrical tubes, Hamilton et al. showed the importance of heat conduction and temperature-dependent viscosity. In their analysis they considered the temperature dependence of thermal conductivity to be of higher order, so that coefficient was held constant. They found that in channels of width between 10 and 20 times the viscous penetration depth, heat conduction had a pronounced effect on the magnitude of streaming velocity. In channels much wider or narrower than that range the effect was smaller. Their analysis of streaming in cylindrical tubes showed qualitatively the same behavior [25, 26].

Thompson and Atchley were able to provide an empirical study of heat conduction effects by using laser Doppler anemometry to measure simultaneously the acoustic velocity and streaming velocity in a tube. They chose a glass cylinder thermally isolated from its surroundings by an air-filled box and collected slices of velocity data in the axial and radial directions. They compared axial Lagrangian streaming velocity to theoretical predictions from both Rott and Qi and found better agreement with Rott’s theory [22].

In another experiment Thompson and Atchley allowed a small temperature
gradient to develop thermoacoustically in the wall of the tube. They measured the axial streaming velocity along the center of the tube but found little agreement with Rott’s theory or any other existing prediction of streaming. They offered as a possible explanation the effect of convective heat transfer which had been seen in other experiments but had not been taken into account theoretically [27]. Thompson and Atchley’s experiments showed the importance of temperature-dependent viscosity and thermal conductivity in the prediction of streaming. Existing theory, however, could not explain the observed effect of temperature gradient.

1.2.3 Fluid Inertia

The importance of fluid inertia in the prediction of streaming was first suggested by Stuart. He noticed that the streaming velocity outside the boundary layer of an oscillating cylinder tended to decrease for large Reynolds number [28]. This nondimensional number is defined as the ratio of inertial forces to viscous forces, and its size indicates the relative importance of those two effects.

Lighthill in his 1978 lecture made the argument that the terms representing fluid inertia should be included in the time-averaged momentum equations that are used to describe streaming velocity. These terms are usually neglected since they are fourth order in a perturbation analysis and are considered insignificant in comparison with second-order terms. Lighthill pointed out that the magnitude of these quantities matters more than their mathematical order. Since the second-order viscous terms include the small coefficient of viscosity, they are not much larger in magnitude than the fourth-order inertial terms. For flow conditions where the Reynolds number is large the fluid inertia is not insignificant in the streaming calculation [10]. Lighthill’s remarks focused on the phenomenon of turbulent jets created by acoustic oscillations, but the argument applies to the analysis of Rayleigh streaming as well.

Menguy and Gilbert put this idea into their analysis and simulation of Rayleigh streaming in a cylindrical waveguide. To distinguish between streaming that experiences the effect of fluid inertia and streaming that does not, they defined a nonlinear Reynolds number in terms of Mach number and Shear number, \( \text{Re}_{nl} = \frac{M^2}{Sh^2} \). The Mach number is given as a characteristic acoustic velocity \( U \) di-
vided by the speed of sound $c_0$, while the Shear number in this case is the ratio of viscous penetration depth $\delta_v$ to the tube radius $R$ which is then divided by $\sqrt{2}$. The nonlinear Reynolds number can be expressed as

$$Re_{nl} = \left(\frac{U}{c_0}\right)^2 \left(\frac{R}{\delta_v/\sqrt{2}}\right)^2.$$  

(1.1)

For $Re_{nl} > 1$ the effect of fluid inertia becomes more significant, and the streaming velocity is considered fast or nonlinear. For lower Reynolds numbers the inertial effect is not as important, and the streaming is considered slow. A numerical study comparing fast and slow streaming showed that for $Re_{nl} = 2$ and $Re_{nl} = 4$ the effect of inertia produced a distortion in the axial streaming velocity along the centerline of the tube which caused the maximum velocity location to shift toward the velocity antinode. A transverse profile of the axial streaming velocity showed a slight deviation from the classical parabolic shape predicted by slow streaming theory [29].

Boluriaan and Morris investigated both thermal effects and inertial effects on streaming using a time-accurate numerical simulation of the full Navier-Stokes equations. They included temperature-dependent viscosity and thermal conductivity, applied isothermal and adiabatic wall boundary conditions, and simulated streaming at various Reynolds numbers in a cylindrical tube and between parallel walls. They confirmed that the transverse profile of axial streaming deviated from a parabolic shape at high Reynolds numbers, becoming more flattened in the center. They also noted that the variations in mean temperature along the axis of the tube differed between adiabatic and isothermal wall boundary conditions. For the simulation of streaming between parallel plates at a Reynolds number of 20, they observed the emergence of a second inner-streaming vortex [30].

Observations of streaming for various Reynolds numbers were provided by Thompson and Atchley’s experiments. They imposed an isothermal boundary by placing the tube of the apparatus inside a larger tube through which water flowed at a constant temperature. This minimized temperature gradients along the tube in an attempt to observe only the effect of inertia on streaming. At a Reynolds number of 4, the resulting axial streaming velocity showed better agreement with Rott’s theory than with that of Menguy and Gilbert, suggesting that the effects
of inertia are less important than the effects of thermal conditions [27].

Most recently Moreau et al. were able to measure inner as well as outer streaming in a cylindrical tube [31]. They used laser Doppler velocimetry to measure both slow and fast streaming with an uncontrolled thermal boundary. Their observations of slow outer streaming showed good agreement with the parabolic profile predicted by Rayleigh. For a Reynolds number of 98, axial streaming along the center of the tube was reduced. Near the tube wall at radial positions less than thirty viscous penetration depths from the wall, they observed slow inner streaming that was similar in form to the predictions of Bailliet [32]. With increasing Reynolds number the inner streaming profile changed and gave the strong suggestion of the development of second and third inner vortices [31]. While the decrease in outer streaming velocity that was observed in these experiments is consistent with the effect of fluid inertia at high Reynolds number, the extra vortices emerging within the boundary layer suggest a transition to turbulence, as mentioned below in Section 1.2.5.

1.2.4 Variations in Tube Radius

The antecedents for the present research on streaming in straight and tapered tubes lie in studies of streaming in waveguides with varying cross-sectional area. The first of these was done by Charles Thompson in 1984 which resulted in an analytical solution for streaming in a rectangular duct with gradually changing height [33]. His analysis did not include temperature-dependent viscosity. Bailliet et al. did include thermal effects in their analysis of streaming between parallel plates and in a cylindrical tube. They included heat conduction and temperature-dependent viscosity and studied the effect of an axial temperature gradient on streaming for different channel widths. They found that the temperature gradient had less effect on streaming in narrow channels, characterized by a small ratio of channel height to viscous penetration thickness, than it did in wide channels [32]. Hamilton et al. also looked at streaming in channels with different widths. Their analysis showed that the relative size of inner and outer streaming vortices depends on the width of the channel relative to the viscous penetration depth. As the channel decreases in width the outer streaming vortex becomes less dominant, and for channel widths
less than 10 times the penetration depth only the inner vortex remains [26].

Of these studies only the earlier one by Thompson involved a gradually changing cross-sectional area. The first team to consider the theory of Rayleigh streaming in tapered channels was Olson and Swift in 1997 [1]. Motivated by their work in thermoacoustic refrigeration and efforts to increase the efficiency of their pulse tube refrigerator, they sought to reduce streaming mass flux in the device. Since the pulse tube is an open channel intended to transmit acoustic power without convecting heat from one end to the other, any flow that could convect heat along the tube, e.g. streaming, would reduce the refrigerator’s efficiency [34]. They approached the problem starting with Rott’s theory of streaming which included the necessary thermal effects [21]. Following a suggestion by Lee et al. [35], they rederived Rott’s equations to allow for a tapered geometry and arbitrary phase between pressure and velocity and found the optimal taper angle for which the streaming mass flux density near the wall was reduced to zero. They tested their theory with the pulse tube refrigerator by designing interchangeable pulse tubes with a variety of taper angles. They found that the refrigerator did indeed perform with more cooling power when they installed the pulse tube with the calculated optimal taper angle. This result suggested that the tapered geometry had the effect of suppressing acoustic streaming [1, 2].

Olson and Swift’s report prompted a three-part investigation by Baek et al. and Park et al. beginning in 2000. They devised a two-dimensional model of a tapered pulse tube using linearized conservation equations of mass, momentum, and energy. They included the tube wall in the overall energy model with an adiabatic boundary condition on the outer surface. The oscillation in the tube was driven at both end boundaries with sinusoidal axial velocity and variable relative phase between them. The equations were integrated by successive approximations using a fourth-order Runge Kutta method. Baek et al. investigated the effects of frequency, taper angle, and phase between the velocity sources on the net energy flow and on mean temperature. Among their resulting observations they showed that first-order axial velocity decreased with increasing taper angle [36].

In the second part of their study Baek et al. continued the method of successive approximations on their model to solve for second-order mean axial velocity, second-order mean temperature, and the resulting enthalpy flow loss, which is the
enthalpy flowing from the hot end to the cold end of the pulse tube. This model included temperature dependent viscosity and thermal conductivity and calculated streaming mass flux that was in good agreement with the mass flux predicted by Olson and Swift’s equation [1]. Upon investigation of the effects of varying frequency and taper angle on mean temperature and mean mass flux, they concluded that tapering the pulse tube reduces both the steady mass flux and the steady component of the second-order temperature, which results in a decrease of streaming-driven enthalpy flow loss [3].

The third part of the study used a hybrid analytical/numerical method to focus on the unsteady second-order temperature and axial velocity. With this model Park et al. concluded that the effect of taper angle on enthalpy flow from unsteady mass flux was very small compared to the effect from steady mass flux [37]. The findings from these three reports focused on the change in enthalpy flow in a pulse tube due to changes in its taper angle and driving frequency and related these effects to the performance of a pulse tube refrigerator. While acoustic streaming played an integral role in the results and calculations, the details of its circulation, generation, and time evolution were not addressed.

Further investigations into the effect of tapered geometry on streaming in a pulse tube refrigerator were made by Shiraishi et al. They visualized flow inside the pulse tube by means of a smoke wire and high speed video camera and used the images to compare flow in a straight tube, a convergent tapered tube (converging from cold end to hot end), and a divergent tapered tube. They found that the transverse profile of velocity in the convergent tapered tube was more flattened compared to the profile seen in the straight tube and that the performance of the refrigerator was slightly better with the converging tube. The performance with the diverging tube was greatly reduced, and the image of the streaming profile revealed a higher velocity. The authors did not ascribe the increase in performance with the convergent tapered tube to the decrease in streaming velocity [4].

In a recent study He et al. related the cooling power of a pulse tube refrigerator to the patterns of streaming circulation in a tube with varying taper angle. They modeled the tube numerically as an ideal gas in a two-dimensional, axisymmetric tapered region flanked by straight sections at either end to represent heat
exchangers. They computed the solution to the Navier-Stokes equations using a finite-volume method, assuming an adiabatic boundary condition at the wall, and modeling the heat exchangers at either end using an “ideal heat exchanger model.” They did not include the thickness of the tube wall in the energy calculation. The time-marching scheme reached a periodic stable convergent condition after approximately 50 cycles, after which they calculated the time-averaged mass flow by integrating the instantaneous mass flow at each grid point over one period. Calculations of the refrigerator’s cooling power at different taper angles showed a peak in performance at one particular angle, above which the cooling power dropped precipitously. He et al. analyzed and compared the mass flow distributions within the tubes and concluded that greater cooling power resulted from a nonuniform secondary (mass streaming) flow field because it disrupted the temperature gradient axially along the tube. Below the optimal taper angle the secondary flow field was increasingly uniform which allowed a temperature gradient to degrade performance, and above the optimal angle the flow field was uniformly strong creating the same effect [5]. The authors did not mention any flow separation where the narrow end of the tapered region coupled with the straight section of the hot heat exchanger, nor did they remark on its resemblance to jet-driven streaming discussed above in Section 1.1.4.

1.2.5 Other Contributions

Many more research reports discuss the role of acoustic streaming in thermoacoustic devices other than pulse tube refrigerators as well as non-thermoacoustic applications. George Mozurkewich wrote about acoustic streaming in a cylindrical resonator and its capacity for affecting heat transport. This could be useful in such applications as cooling electronic devices or assisting the power flow of thermoacoustic engines [38]. Lu and Cheng analyzed thermoacoustic energy streaming in a tube with an isothermal outer surface, and they emphasized the connection between mass streaming and energy streaming [39]. Gusev et al. investigated enthalpy transport due to acoustic streaming in an annular thermoacoustic prime-mover and developed a theory applicable to other traveling-wave thermoacoustic devices [40]. Boluriaan and Morris also studied traveling-wave streaming and con-
sidered its suppression by the use of a jet pump. They built a numerical model of a traveling wave in a tube with an abrupt constriction to induce a jet toward one side and reported its effects on the acoustic field [41].

Other examples of numerical modeling of acoustic streaming include a variety of methods and applications. Visualizations of the flow around the parallel plates of a thermoacoustic stack were carried out by Marx, Blanc-Benon, and others [42, 43, 44]. They simulated streaming flow between closely spaced parallel plates, between an array of plates and a heat exchanger, and at the end of an array of parallel plates. Examples of unique numerical methods include two simulations of Rayleigh streaming, one using a lattice Boltzmann model of streaming around a cylinder and between parallel plates by Haydock and Yeomans [45], and another by Sastrapradja that modeled streaming in a cylinder by way of the vorticity transport equation [46, 47]. Lastly, the notable contributions of Takeru Yano reveal through his numerical simulations some strong nonlinear acoustic effects in conjunction with acoustic streaming near a piston [48], in a closed resonant tube with periodic shock waves [49], and in a resonator at high Reynolds number [50]. Numerical modeling of acoustic streaming is an active field of study that has enjoyed many important advances in an increasing variety of areas.

1.3 Motivation and Goals

The motivations for this research are threefold. First is the potential application to thermoacoustic refrigeration and the promise of better-informed design decisions to create more efficient and more reliable devices. Efficiency and reliability are two attributes that make thermoacoustic refrigeration an attractive replacement technology for conventional vapor-compression-cycle refrigerators. There are many fewer moving parts that rely on close tolerances, and as a result thermoacoustic devices are potentially better suited for use in remote, demanding environments such as sea-going vessels or spacecraft [34]. A better understanding of energy flow in these devices and how energy loss can be minimized through control of acoustic streaming can help to make thermoacoustic refrigeration a viable alternative technology.
The second motivating factor is the understanding of streaming itself. As illustrated above in the overview of prior research, there are conflicting and uncertain reports on the mechanism through which streaming is affected by tapered versus straight tubes. The general consensus among those studies cited is that pulse tube refrigerators perform with more cooling power when the pulse tube is tapered, and that the taper angle affects streaming, possibly reducing it. Questions remain regarding the influence of frequency, phase between velocity sources at the ends of the tube, phase between acoustic density and velocity, the relative influence of high-order effects, and more.

The third factor and catalyst for this research is the report by Olson and Swift describing their theoretical analysis of the effect of taper angle on second-order mass flux in a tube, the calculation of an optimal angle to eliminate that flux, and the increased cooling achieved using a pulse tube tapered at that angle [2]. Their findings beg the question, is streaming in a tube reduced by taper?

As noted above in Section 1.2.4, several researchers have investigated tapered pulse tubes following the work of Olson and Swift, mainly with the focus of relating the cooling power and performance of the pulse tube refrigerator to the mass flux, enthalpy flux, and temperature gradients within the tube. This research aims to address the question above by concentrating on the relation of taper angle to the velocity of Rayleigh streaming. The goal is to isolate Rayleigh streaming by developing a standing wave in a straight tube and in tubes of varying taper angles and comparing between them the transverse profiles of streaming velocity at a consistent point along the length of the tube. This is to be accomplished with a time-accurate numerical model that provides a solution to the Navier-Stokes equations. The model equations are separated by perturbation analysis into mathematically ordered terms to allow the high-order effects to be compared.

1.3.1 Challenges

Acoustic streaming is a small effect that demands computational accuracy over a long period of time, especially if the simulation is to be time accurate. The challenge is one of scale. A numerical model is required to represent a large portion of a wavelength in its computational domain while at the same time resolving the
boundary layer within a small fraction of the penetration depth. These lengths are typically very different in scale. Furthermore the numerical stability of the finite difference scheme requires a very small time step commensurate with the finest spatial resolution of the model. Again the scale of the single time step is much much smaller than the hundreds of acoustic cycles necessary for streaming to fully develop. This range of scale at high accuracy results in a large computational expense. This can be made tractable by refining the computational grid only in regions where boundary layers occur and by using multiple processors to share the burden in parallel.

1.4 Thesis Overview

The following chapters present the details of the numerical model, its execution and analysis, the resulting figures showing acoustic streaming, and discussion of what they mean. Chapter 2 presents the geometry and coordinate systems for the straight and tapered tubes, and it details the development of the model equations in cylindrical and spherical coordinates in preparation for implementation in the finite-difference scheme. In Chapter 3 the model equations and computational domain are discretized and mapped onto a stretched grid. The following sections describe the finite-difference method including details of computing second derivatives and source terms. The boundary treatments include discussion of a method to avoid grid points placed directly on an axis. For details on the derivation of the non-reflecting boundary condition, the reader is referred to Appendix B. Chapter 3 concludes with a short description of the multi-processor implementation. Chapter 4 describes various aspects of executing the model. These include the method of exciting oscillation via a line source, computation of volume velocity to normalize the streaming velocity and compare between tubes of various cross-sectional area, and the method of computing the Lagrangian streaming velocity. Chapter 5 shows the verification of the numerical model with analytical predictions. It presents a study of the differences between adiabatic and isothermal temperature boundary conditions at the tube wall and their effect on Rayleigh streaming. It also investigates the importance of high-order terms in the simulation of streaming. Finally Chapter 5 shows a comparison of streaming circulation, axial streaming velocity
profiles, and mass flux in tubes of different taper angles. The final chapter gives a review of the research, listing results and contributions to the study of acoustic streaming, and offers possible future enquiries.
Chapter 2

Numerical Model

The numerical model developed for this study is inspired by the 1997 experiment of Olson and Swift [1]. It uses physical parameters, operating conditions, and calculations of tube geometry based on the description of their pulse tube refrigerator. The model deviates from the experiment in that it considers only Rayleigh streaming driven by pure standing-wave phasing in order to exclude any effects of traveling-wave-driven streaming as described in Section 1.1.3. The tube length is adjusted accordingly to accommodate a sufficient portion of a standing wave. The details of the experimental apparatus are listed in Section 2.1. Section 2.2 shows the choice of coordinate systems and the orientation of the model in those contexts. The rest of the chapter is dedicated to the presentation and development of the model equations in each coordinate system including a discussion of perturbation expansion. Finally the equations are arranged in a non-conservation form for implementation in the finite-difference computation.

2.1 Experimental Basis

The experimental apparatus that serves as the basis for this model is described in detail in Olson and Swift’s paper, “Acoustic streaming in pulse tube refrigerators: tapered pulse tubes” [1]. They show the tapered pulse tube oriented vertically so that the narrow end is at the top connected to the hot heat exchanger and the wide end is at the bottom with the cold heat exchanger.
The experiment uses three interchangeable pulse tubes of stainless steel, each having a length of 4.65 cm and a volume of 11.0 cm$^3$. The first is a right circular cylinder of radius 8.7 mm. The second is a truncated cone with a total included angle of 0.049 rad = 2.8°, and the third is a truncated cone having approximately twice that angle, 0.103 rad. The taper angle of the second tube is calculated by Equation 2.1 to be the optimal taper angle to suppress streaming [1].

\[
\frac{1}{A} \left( \frac{dA}{dx} \right) = -\frac{\omega|p_1|}{\gamma p_m \langle u_1 \rangle} \left[ \left( 1 + \frac{2(\gamma - 1)(1 - b\sigma^2)}{3\sigma(1 + \sigma)} \right) \cos \theta \right.
\]
\[+ \left. \left( 1 + \frac{2(\sigma - 1)(1 - b)\sqrt{\sigma}}{3(1 + \sigma)} \right) \sin \theta \right] - \frac{(1 - b)(1 - \sqrt{\sigma})}{3(1 + \sigma)(1 + \sqrt{\sigma})} \frac{(dT_m/dx)}{T_m}
\]

Here, $\gamma$ is the ratio of specific heats, $\sigma$ is the Prandtl number, $\theta$ is the phase angle by which the oscillating axial velocity $u_1$ leads the oscillating pressure $p_1$, and $\langle u_1 \rangle$ denotes the lateral (radial) spatial average of $u_1$. The values $p_m$ and $T_m$ are the steady-state mean pressure and temperature with no time dependence. The factor $b$ is empirically determined from the calculation of the temperature dependence of viscosity, $\mu(T) = \mu_0(T/T_0)^b$. The taper angle $\phi$ is found from the relation for a conical tube, $(1/A)(dA/dx) = -2\tan(\phi/2)/R$, where $R$ is the radius of the tube [2].

The pulse tube refrigerator was filled with helium gas at 3.1 MPa and driven at a frequency of 100 Hz. The amplitude of the oscillation was adjusted so that the pressure amplitude at the hot end of the pulse tube was kept within 0.5% of 1.95 $\times$ 10$^5$ Pa. Axial velocity in the tube was determined by flow measurements taken in the constriction between the pulse tube and the compliance and inferred elsewhere in the tube by geometry. The temperature of the coolant flowing through the hot heat exchangers was maintained at 15° C, and a heat load was supplied to the cold heat exchanger where the temperature of the gas was monitored with a thermocouple [1].

Olson and Swift reported this temperature, $T_c$, versus the heat load on the cold heat exchanger, $Q_{cold}$, in their results. Temperature increased with heat load for each of the three pulse tubes, but was at least 5° C lower in the case of the
optimally tapered tube than in the other two cases. This indicated that a higher cooling efficiency was obtained with the taper angle calculated by Equation 2.1 than with the other two pulse tube taper angles [1].

2.2 Model Geometry

Two geometries serve as the basis of the numerical model for comparison of streaming in straight and tapered tubes. The straight tube is built on a cylindrical coordinate system, while the tapered tube uses spherical coordinates. The choice of spherical coordinates facilitates computation of the boundary at the tube wall by virtue of the normal incidence of the computational grid to the angled surface. To maintain orthogonality at the ends of the tapered tube, it is assumed that the end caps are spherical rather than planar as they are in the cylindrical tube. The difference in geometry at the tube ends is considered to be negligible for this study due to the shallow taper angle.

The two geometries are shown side by side in Figure 2.1. The included angle of the tapered tube is greatly exaggerated in this diagram to better illustrate the coordinate systems. The orientation of the tapered tube in spherical coordinates is drawn to match that of the pulse tube in Olson and Swift’s experiment. As such, references to the vertical or $z$ axis are made so that the positive direction is downward in both geometries. It is also noted that the numerical model omits the compliance at the top of the pulse tube as well as the flow straightener and everything below it as described by Olson and Swift [1]. Here only the tube is modeled.

To further simplify the calculations, the model is reduced to a two-dimensional representative slice in each geometry. These are indicated by the shaded regions in Figure 2.1. For the purpose of this investigation only axisymmetric modes of streaming are considered. It is assumed that the fluid dynamics modeled in one two-dimensional slice are typical of any such axial slice through the tube.
2.3 Model Equations

With the geometries established, the next step is to develop the model equations that govern the fluid motion and to write them in a form that can be computed by the finite-difference algorithm. The equations are presented as follows in general vector notation, not specific to either geometry:

Continuity Equation
\[
\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0 \tag{2.2}
\]

Momentum Equation
\[
\rho \frac{D\mathbf{v}}{Dt} = -\nabla p - \nabla \times [\mu (\nabla \times \mathbf{v})] + \frac{4}{3} \nabla (\mu \nabla \cdot \mathbf{v}) + \rho \mathbf{g} \tag{2.3}
\]

Energy Equation
\[
\rho c_v \frac{DT}{Dt} = -p \nabla \cdot \mathbf{v} + \frac{\mu}{2} \phi_{ij} \phi_{ij} + \nabla \cdot (\kappa \nabla T) \tag{2.4}
\]

Equation of State
\[
p = \rho RT \tag{2.5}
\]

Sutherland’s Formulae
\[
\mu = \mu_0 \frac{T_0 + C_1}{T + C_1} \left( \frac{T}{T_0} \right)^{3/2} \tag{2.6}
\]
The continuity equation is derived in Chapter 2 of Anderson’s Computational Fluid Dynamics [51]. The variable \( \rho \) represents the total density, and \( \mathbf{v} \) is the velocity vector. The momentum equations, expressed here as a single equation with velocity vector \( \mathbf{v} \) are the Navier-Stokes equations for a compressible fluid [52]. In this equation Stokes’ hypothesis is applied, which says that the bulk viscosity is zero for a monatomic gas. Thus dilatational viscosity \( \lambda \) and shear viscosity \( \mu \) are related by \( \lambda = \frac{2}{3} \mu \) [53]. The viscosity is written with spatial dependence because it is allowed to vary with temperature. The body force due to gravity, written here as density times acceleration due to gravity, \( \rho \mathbf{g} \), provides the means to investigate in a possible future study the effects of gravity and tube orientation on the development of streaming.

The temperature dependence of viscosity, \( \mu \), and thermal conductivity, \( \kappa \), is defined by Sutherland’s formulae, Equations 2.6 and 2.7. These are obtained from the International Critical Tables of Numerical Data where \( C_1 \) is given as 70 K and \( C_2 \) is 33 K for helium [54]. Alternatively, thermal conductivity may be calculated from viscosity by the relation \( \kappa = \mu c_p / Pr \), where \( Pr \) is the Prandtl number. The equation of state, Equation 2.5, assumes a perfect gas and defines total pressure \( p \) by the product of total density, total temperature, and the specific gas constant \( R \).

The energy equation is obtained from a simplified Fourier-Kirchhoff-Neumann equation similar to Equation 10-1.8 in Pierce’s Acoustics [53].

\[
\rho \frac{D e}{D t} = \sigma_{ij} \frac{\partial v_i}{\partial x_j} - \nabla \cdot \mathbf{q}
\]  

(2.8)

This presentation differs from Pierce’s in that the variable \( e \) replaces \( u \) as the internal energy per mass, and the summation symbol is omitted in favor of the Einstein summation convention of indicial notation [55].

The energy equation is written in terms of temperature by first considering the right-hand side of Equation 2.8. The first constitutive equation defines the stress
tensor $\sigma$ as the sum of normal stresses and shear stresses \[56].

$$\sigma_{ij} = \sigma_n \delta_{ij} + \lambda \delta_{ij} \frac{\partial v_k}{\partial x_k} + \mu \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right)$$ \hspace{1cm} (2.9)

Applying Stokes’ hypothesis, $\lambda = -\frac{2}{3} \mu$, yields

$$\sigma_{ij} = \sigma_n \delta_{ij} + \mu \left[ -\frac{2}{3} \delta_{ij} \frac{\partial v_k}{\partial x_k} + \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) \right]$$ \hspace{1cm} (2.10)

where the quantity in square brackets is identified by Pierce as components of the rate-of-shear tensor \[53],

$$\phi_{ij} = -\frac{2}{3} \delta_{ij} \frac{\partial v_k}{\partial x_k} + \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right).$$ \hspace{1cm} (2.11)

Multiplying Equation 2.10 by the velocity-gradient tensor produces the following expression.

$$\sigma_{ij} \frac{\partial v_i}{\partial x_j} = \sigma_n \delta_{ij} \frac{\partial v_i}{\partial x_j} + \mu \phi_{ij} \frac{\partial v_i}{\partial x_j}$$ \hspace{1cm} (2.12)

Since the rate-of-shear tensor is symmetric and has zero trace, its product with the velocity-gradient tensor can be written concisely as one half the sum of the squared components of the rate-of-shear tensor. Equation 2.12 becomes

$$\sigma_{ij} \frac{\partial v_i}{\partial x_j} = \sigma_n \delta_{ij} \frac{\partial v_i}{\partial x_j} + \frac{\mu}{2} \phi_{ij} \phi_{ij}.$$ \hspace{1cm} (2.13)

Substituting Equation 2.12 into Equation 2.8 gives the following form of the energy equation.

$$\rho \frac{\partial v_i}{\partial t} = \sigma_n \delta_{ij} \frac{\partial v_i}{\partial x_j} + \frac{\mu}{2} \phi_{ij} \phi_{ij} - \nabla \cdot q$$ \hspace{1cm} (2.14)

Assuming that the fluid in the model maintains local thermodynamic equilibrium, the average normal component of the stress tensor can be defined as negative pressure, $\sigma_n = -p$ \[53]. By the second constitutive equation, heat flux is related to the temperature gradient near equilibrium by the coefficient of thermal conductivity, $q = -\kappa \nabla T$. For a gas with constant specific heats, internal specific energy can be expressed in terms of temperature by the caloric equation of state, $e = c_v T$ \[51].
Using these relations, Equation 2.14 is written in terms of temperature as presented above in Equation 2.4 and repeated here in Equation 2.15.

\[
\rho c_v \frac{DT}{Dt} = -p \nabla \cdot \mathbf{v} + \frac{\mu}{2} \phi_{ij} \phi_{ij} + \nabla \cdot (\kappa \nabla T) \tag{2.15}
\]

Like viscosity in the momentum equation, thermal conductivity is spatially dependent in this equation since it is also allowed to vary with temperature.

### 2.3.1 Cylindrical Coordinates

![Figure 2.2. Cylindrical coordinates](image)

The transformation of the model equations to cylindrical coordinates is carried out by way of the following definitions where \( \Phi \) and \( \mathbf{A} \) are generic scalar and vector quantities, respectively, and \( \hat{z}, \hat{r}, \) and \( \hat{\theta} \) are the three coordinate vectors whose directions are indicated in the diagram in Figure 2.2.

\[
\begin{align*}
\mathbf{v} &= \hat{z} v_z + \hat{r} v_r + \hat{\theta} v_\theta \\
\frac{D}{Dt} &= \frac{\partial}{\partial t} + v_z \frac{\partial}{\partial z} + v_r \frac{\partial}{\partial r} + \frac{v_\theta}{r} \frac{\partial}{\partial \theta} \\
\nabla \Phi &= \hat{z} \frac{\partial \Phi}{\partial z} + \hat{r} \frac{\partial \Phi}{\partial r} + \frac{1}{r} \hat{\theta} \frac{\partial \Phi}{\partial \theta} \\
\nabla \cdot \mathbf{A} &= \frac{\partial A_z}{\partial z} + \frac{1}{r} \frac{\partial (r A_r)}{\partial r} + \frac{1}{r} \frac{\partial A_\theta}{\partial \theta}
\end{align*}
\tag{2.16-2.19}
\]
\[
\n\nabla \times \mathbf{A} = \hat{z} \left( \frac{1}{r} \left[ \frac{\partial}{\partial r} (rA_\theta) - \frac{\partial A_r}{\partial \theta} \right] \right) + \\
\hat{r} \left( \frac{1}{r} \frac{\partial A_z}{\partial \theta} - \frac{\partial A_\theta}{\partial z} \right) + \\
\hat{\theta} \left( \frac{\partial A_r}{\partial z} - \frac{\partial A_z}{\partial r} \right)
\]

(2.20)

Taking advantage of the simplification afforded by axisymmetry, any resulting term that is associated with variation in the \( \hat{\theta} \)-direction, i.e. \( v_\theta, \frac{\partial}{\partial \theta}, \) or \( \hat{\theta} \), is assumed to be zero. The cylindrical-coordinate version of the Navier-Stokes equations of motion for a compressible fluid with varying viscosity is found in *Basic Equations of Engineering Science* by Hughes and Gaylord [52].

The model equations in cylindrical coordinates with symmetry are summarized below. The components of the rate-of-shear tensor are kept in indicial notation for brevity in the temperature equation. Body force due to gravity is included only in the \( \hat{z} \)-direction.

Continuity Equation

\[
\frac{\partial \rho}{\partial t} + \frac{\partial \rho v_z}{\partial z} + \frac{\partial \rho v_r}{\partial r} = -\frac{\rho v_r}{r}
\]

(2.21)

Momentum Equation, \( \hat{z} \)-direction

\[
\rho \left( \frac{\partial v_z}{\partial t} + v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_z}{\partial r} \right) = -\frac{\partial p}{\partial z} + \frac{\partial}{\partial z} \left[ \frac{2}{3} \mu \left( \frac{\partial v_z}{\partial z} - \frac{v_r}{r} - \frac{\partial v_r}{\partial r} \right) \right] + \frac{1}{r} \frac{\partial}{\partial r} \left[ \mu r \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] + \rho g_z
\]

(2.22)

Momentum Equation, \( \hat{r} \)-direction

\[
\rho \left( \frac{\partial v_r}{\partial t} + v_z \frac{\partial v_r}{\partial z} + v_r \frac{\partial v_r}{\partial r} \right) = -\frac{\partial p}{\partial r} + \frac{\partial}{\partial z} \left[ \mu \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] + \frac{\partial}{\partial r} \left[ \frac{2}{3} \mu \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} - \frac{\partial v_z}{\partial z} \right) \right] + \frac{2}{r} \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} \right)
\]

(2.23)
Temperature Equation

\[
\rho c_v \left( \frac{\partial T}{\partial t} + v_z \frac{\partial T}{\partial z} + v_r \frac{\partial T}{\partial r} \right) = -p \left( \frac{\partial v_r}{\partial r} + \frac{v_r}{r} + \frac{\partial v_z}{\partial z} \right) + \frac{\mu}{2} \phi_{ij} \phi_{ij} \\
+ \frac{\partial}{\partial r} \left( \kappa \frac{\partial T}{\partial r} \right) + \frac{1}{r} \kappa \frac{\partial T}{\partial r} + \frac{\partial}{\partial z} \left( \kappa \frac{\partial T}{\partial z} \right)
\] (2.24)

2.3.1.1 Perturbation Method

Perturbation analysis provides a way to discretize functions in the model equations and control their level of approximation through comparison with small ordering parameters. The functions are approximated as finite series expansions whose truncation renders them valid over the discrete domain of the numerical calculation. The truncation point is chosen by the modeler to control the amount of discretization error and to reduce the number of terms in the model equations. Through the selection of terms from the perturbation expansion the resulting simplification not only controls discretization error, but also may reduce round-off error and computer run time.

The appropriate small ordering parameter for the numerical model of acoustic streaming is shown in Appendix A to be the acoustic Mach number \( \varepsilon \). This nondimensional quantity compares the acoustic particle velocity to the constant speed of sound. Its value is typical of the magnitude of acoustic variation in the model variables. In addition the Stokes number \( \eta \) is shown in Appendix A to be representative of the magnitude of thermal and viscous dissipation elements in the model.

The static part of each model variable is order zero in \( \varepsilon \) and represents the ambient quantity. It is not defined as the mean of the total variable but represents a time- and space-invariant quantity that describes the state of the variable in a quiescent medium. The fluctuating part of each variable is defined as the total variable minus the static part and includes fluctuating quantities of order one and higher in \( \varepsilon \).
\[ \rho' = \rho - \rho_0 = \rho_1 + \rho_2 + O(\varepsilon^3) \]
\[ \mathbf{v} = \mathbf{v} - 0 = \mathbf{v}_1 + \mathbf{v}_2 + O(\varepsilon^3) \]
\[ T' = T - T_0 = T_1 + T_2 + O(\varepsilon^3) \]
\[ p' = p - p_0 = p_1 + p_2 + O(\varepsilon^3) \]
\[ \mu' = \mu - \mu_0 = \mu_1 + O(\eta\varepsilon^2) \]
\[ \kappa' = \kappa - \kappa_0 = \kappa_1 + O(\eta\varepsilon^2) \]

(2.25)

Since there is no steady flow, the fluctuating velocity is the same as the total velocity. The subscripts indicate order in \( \varepsilon \). The fluctuating parts that are \( O(\varepsilon) \) represent those variations that are acoustic in time with \( e^{i\omega t} \) dependence. The mean of each of these first-order variables is zero by definition. The second-order variables and higher represent all other fluctuations including, notably, nonlinear effects which may have nonzero mean.

For the operating parameters of this model the Stokes number is much smaller than the acoustic Mach number. The dissipative terms, which are those associated with the transport coefficients \( \mu \) and \( \kappa \), are of order \( \eta \). Thus viscosity \( \mu_0 \) is \( O(\eta) \), and \( \mu' \) which fluctuates with acoustic temperature is \( O(\eta\varepsilon) \). The same holds for the coefficient of thermal conductivity.

The expansion of the model variables is applied to the model equations in cylindrical coordinates. Any derivatives of zero-order quantities drop out allowing the focus of the numerical model to center on the solution of the fluctuating quantities. The purpose of solving for the fluctuating part rather than the total variable is to reduce round-off error which may occur due to acoustic fluctuations that are very small compared to the total quantities. The products of the fluctuating variables are retained up to \( O(\varepsilon^2) \). Products involving dissipation coefficients are retained up to \( O(\eta\varepsilon^2) \). Since this research investigates the importance of higher order terms in the streaming simulation, those terms are kept distinct so that they may be omitted or included for comparison. Applying the perturbation method to Equations 2.21 to 2.24 yields Equations 2.26 to 2.29.

Continuity Equation

\[
\frac{\partial \rho'}{\partial t} + \frac{\partial}{\partial z}[(\rho_0 + \rho') v_z] + \frac{\partial}{\partial r}[(\rho_0 + \rho') v_r] = - (\rho_0 + \rho') \frac{v_r}{r} \quad (2.26)
\]
Momentum Equation, \( \hat{z} \)-direction

\[
\rho_0 \left( \frac{\partial v_z}{\partial t} + v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_z}{\partial r} \right) + \rho' \frac{\partial v_z}{\partial t} = -\frac{\partial p'}{\partial z} + \left( \frac{2}{3} (\mu_0 + \mu') \left( \frac{\partial v_z}{\partial z} - \frac{v_r}{r} - \frac{\partial v_r}{\partial r} \right) \right) + \frac{1}{r} \frac{\partial}{\partial r} \left( (\mu_0 + \mu') r \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right) + (\rho_0 + \rho') g_z
\] (2.27)

Momentum Equation, \( \hat{r} \)-direction

\[
\rho_0 \left( \frac{\partial v_r}{\partial t} + v_z \frac{\partial v_r}{\partial z} + v_r \frac{\partial v_r}{\partial r} \right) + \rho' \frac{\partial v_r}{\partial t} = -\frac{\partial p'}{\partial r} + \left( \frac{2}{3} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right) + \frac{2}{r} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} \right)
\] (2.28)

Temperature Equation

\[
c_v \rho_0 \left( \frac{\partial T'}{\partial t} + v_z \frac{\partial T'}{\partial z} + v_r \frac{\partial T'}{\partial r} \right) + c_v \rho' \frac{\partial T'}{\partial t} = \frac{\partial}{\partial z} \left[ (\kappa_0 + \kappa') \frac{\partial T'}{\partial z} \right] + \left[ \frac{1}{r} (\kappa_0 + \kappa') \frac{\partial T'}{\partial r} + \frac{\partial}{\partial r} \left( (\kappa_0 + \kappa') \frac{\partial T'}{\partial r} \right) \right] - (p_0 + p') \left( \frac{\partial v_z}{\partial z} + \frac{v_r}{r} \frac{\partial v_r}{\partial r} \right) + \frac{\mu_0}{2} \phi_{ij} \phi_{ij}
\] (2.29)

2.3.1.2 Development for Numerical Implementation

Equations 2.27, 2.28, and 2.29 each contain two time derivatives. This complicates the time-stepping calculation by the numerical scheme, so the extra time derivatives are rewritten by linearizing each of these equations.

\[
\frac{\partial v_z}{\partial t} = -\frac{1}{\rho_0} \frac{\partial p'}{\partial z} + g_z
\] (2.30)

\[
\frac{\partial v_r}{\partial t} = -\frac{1}{\rho_0} \frac{\partial p'}{\partial r}
\] (2.31)
\[
\frac{\partial T'}{\partial t} = -\frac{p_0}{c_v \rho_0} \left( \frac{\partial v_r}{\partial r} + \frac{v_r}{r} + \frac{\partial v_z}{\partial z} \right)
\]  

(2.32)

Substituting Equations 2.30 to 2.32 back into Equations 2.27 to 2.29 yields the model equations with a single time derivative. This substitution is valid because the discretization errors from the linearized acoustic-mode equations are \(O(\varepsilon^2)\). When these are substituted back into the model equations, they are multiplied by \(O(\varepsilon)\) terms making the discretization errors \(O(\varepsilon^3)\) which is consistent with the rest of the equation [57].

\[
\rho_0 \left( \frac{\partial v_z}{\partial t} + v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_z}{\partial r} \right) - \rho' \frac{\partial p'}{\partial z} = -\frac{\partial p'}{\partial z} + \frac{\partial}{\partial z} \left[ \frac{2}{3} (\mu_0 + \mu') \left( 2 \frac{\partial v_z}{\partial z} - \frac{v_r}{r} - \frac{\partial v_r}{\partial r} \right) \right] + \frac{1}{r} \frac{\partial}{\partial r} \left[ (\mu_0 + \mu') r \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] + \rho_0 g_z
\]  

(2.33)

\[
\rho_0 \left( \frac{\partial v_r}{\partial t} + v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_r}{\partial r} \right) - \rho' \frac{\partial p'}{\partial r} = -\frac{\partial p'}{\partial r} + \frac{\partial}{\partial z} \left[ (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] + \frac{\partial}{\partial r} \left[ \frac{2}{3} (\mu_0 + \mu') \left( 2 \frac{\partial v_r}{\partial r} - \frac{v_r}{r} - \frac{\partial v_z}{\partial z} \right) \right] + \frac{2}{r} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} \right)
\]  

(2.34)

\[
c_v \rho_0 \left( \frac{\partial T'}{\partial t} + v_z \frac{\partial T'}{\partial z} + v_r \frac{\partial T'}{\partial r} \right) - \rho' p_0 \left( \frac{\partial v_z}{\partial z} + \frac{v_r}{r} + \frac{\partial v_r}{\partial r} \right) = \frac{\partial}{\partial z} \left[ (\kappa_0 + \kappa') \frac{\partial T'}{\partial z} \right] + \frac{1}{r} (\kappa_0 + \kappa') \frac{\partial T'}{\partial r} + \frac{\mu_0}{2} \phi_{ij} \phi_{ij} + \frac{\partial}{\partial r} \left[ (\kappa_0 + \kappa') \frac{\partial T'}{\partial r} \right] - (p_0 + p') \left( \frac{\partial v_z}{\partial z} + \frac{v_r}{r} + \frac{\partial v_r}{\partial r} \right)
\]  

(2.35)

With this substitution the momentum and temperature equations are re-arranged in preparation for numerical implementation.
Momentum Equation, \(\hat{z}\)-direction

\[
\frac{\partial v_z}{\partial t} + \frac{\partial}{\partial z} \left[ \frac{p'}{\rho_0} - \frac{2}{3\rho_0} (\mu_0 + \mu') \left( 2 \frac{\partial v_z}{\partial z} - \frac{v_r}{r} - \frac{\partial v_r}{\partial r} \right) \right] + \frac{\partial}{\partial r} \left[ -\frac{1}{\rho_0} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] = \\
\frac{\rho' \rho_0'}{\rho_0^2} \frac{\partial p'}{\partial z} - \left( v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_z}{\partial r} \right) + \frac{1}{\rho_0 r} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) + g_z
\] (2.36)

Momentum Equation, \(\hat{r}\)-direction

\[
\frac{\partial v_r}{\partial t} + \frac{\partial}{\partial z} \left[ -\frac{1}{\rho_0} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r} \right) \right] + \frac{\partial}{\partial r} \left[ \frac{p'}{\rho_0} - \frac{2}{3\rho_0} (\mu_0 + \mu') \left( 2 \frac{\partial v_r}{\partial r} - \frac{v_r}{r} - \frac{\partial v_z}{\partial z} \right) \right] = \\
\frac{\rho' \rho_0'}{\rho_0^2} \frac{\partial p'}{\partial r} - \left( v_z \frac{\partial v_r}{\partial z} + v_r \frac{\partial v_r}{\partial r} \right) + \frac{2}{\rho_0 r} (\mu_0 + \mu') \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} \right)
\] (2.37)

Temperature Equation

\[
\frac{\partial T'}{\partial t} + \frac{\partial}{\partial z} \left[ \frac{1}{\rho_0 c_v} \left( p_0 v_z - (\kappa_0 + \kappa') \frac{\partial T'}{\partial z} \right) \right] + \frac{\partial}{\partial r} \left[ \frac{1}{\rho_0 c_v} \left( p_0 v_r - (\kappa_0 + \kappa') \frac{\partial T'}{\partial r} \right) \right] = \\
- v_z \frac{\partial T'}{\partial z} + \frac{1}{\rho_0 c_v r} (\kappa_0 + \kappa') - v_r \frac{\partial T'}{\partial r} + \frac{\mu_0}{2} \phi_{ij} \phi_{ij} + \\
+ \frac{p_0}{\rho_0 c_v} \left[ \left( \frac{\rho'}{\rho_0} - \frac{p'}{p_0} \right) \left( \frac{\partial v_z}{\partial z} + \frac{v_r}{r} + \frac{\partial v_r}{\partial r} \right) - \frac{v_r}{r} \right]
\] (2.38)

The model equations in cylindrical coordinates are summarized here as array components of an equation of difference operators. For brevity the coefficients of viscosity and thermal conductivity of order \(\eta e^2\) are rewritten as \(\mu_0 + \mu' = \mu(T')\) and \(\kappa_0 + \kappa' = \kappa(T')\) to emphasize their dependence on acoustic temperature.

\[
\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}}{\partial z} + \frac{\partial \mathbf{G}}{\partial r} = \mathbf{S} \mathbf{F} + \mathbf{S} \mathbf{G}
\] (2.39)
These equations of motion, Equations 2.39 and 2.40, along with Sutherland’s formulae, Equations 2.6 and 2.7, and the equation of state 2.5 provide a closed-form system representing the numerical model in cylindrical coordinates.
2.3.2 Spherical Coordinates

The axisymmetric spherical coordinates are shown in Figure 2.3. Tube length increases in the downward direction the same as in cylindrical coordinates, but in this geometry length is denoted by the $r$-coordinate. Distance from the tube axis is measured by the angle $\theta$ in radians which is the quantity used in the equations of motion for this coordinate system. Where distance from the axis in arc length is required, it is calculated by $\alpha = r\theta$.

![Figure 2.3. Spherical coordinates](image)

The definitions used to transform the model equations to spherical coordinates are listed in equations 2.41 to 2.45. As in cylindrical coordinates any term that is associated with variation in the $\varphi$-direction as a result of this transformation is assumed to be zero due to axisymmetry. The spherical-coordinate version of the Navier-Stokes equations is found in Hughes and Gaylord [52].

\[
\mathbf{v} = \dot{r}v_r + \dot{\theta}v_\theta + \dot{\varphi}v_\varphi
\]  
\[
\frac{D}{Dt} = \frac{\partial}{\partial t} + v_r \frac{\partial}{\partial r} + \frac{v_\theta}{r} \frac{\partial}{\partial \theta} + \frac{v_\varphi}{r \sin \theta} \frac{\partial}{\partial \varphi}
\]  
\[
\nabla \Phi = \frac{\dot{r}}{r} \frac{\partial \Phi}{\partial r} + \dot{\theta} \frac{1}{r} \frac{\partial \Phi}{\partial \theta} + \dot{\varphi} \frac{1}{r \sin \theta} \frac{\partial \Phi}{\partial \varphi}
\]  

(2.41)  
(2.42)  
(2.43)
\[ \nabla \cdot \mathbf{A} = \frac{1}{r^2} \frac{\partial}{\partial r} (r^2 A_r) + \frac{1}{r \sin \theta} \frac{\partial}{\partial \theta} (\sin \theta A_\theta) + \frac{1}{r \sin \theta} \frac{\partial A_\phi}{\partial \phi} \] (2.44)

\[ \nabla \times \mathbf{A} = \hat{r} \left( \frac{1}{r \sin \theta} \left[ \frac{\partial}{\partial \theta} (\sin \theta A_\phi) - \frac{\partial A_\theta}{\partial \phi} \right] \right) + \hat{\theta} \left( \frac{1}{r \sin \theta} \frac{\partial A_r}{\partial \phi} - \frac{1}{r} \frac{\partial}{\partial r} (r A_\phi) \right) + \hat{\phi} \left( \frac{1}{r} \left[ \frac{\partial}{\partial r} (r A_\theta) - \frac{\partial A_r}{\partial \theta} \right] \right) \] (2.45)

The model equations in spherical coordinates with axisymmetry are listed below.

Continuity Equation

\[ \frac{\partial \rho}{\partial t} + \frac{\partial (\rho v_r)}{\partial r} + \frac{\partial (\rho v_\theta)}{\partial \theta} + \frac{\partial (\rho v_\phi)}{\partial \phi} = -2 \frac{\rho v_r}{r} - \frac{\rho v_\theta}{r} \cot \theta \] (2.46)

Momentum Equation, \( \hat{r} \)-direction

\[ \rho \left( \frac{\partial v_r}{\partial t} + v_r \frac{\partial v_r}{\partial r} + v_\theta \frac{\partial v_r}{\partial \theta} - \frac{v_\theta^2}{r} \right) = -\frac{\partial p}{\partial r} + \frac{\partial}{\partial r} \left[ \frac{2}{3} \mu \left( 2 \frac{\partial v_r}{\partial r} + 2 \frac{v_r}{r} + \frac{\partial (v_\theta / r)}{\partial \theta} + \frac{v_\theta}{r} \cot \theta \right) \right] \]

\[ + \frac{1}{r} \frac{\partial}{\partial \theta} \left[ \mu \left( \frac{\partial (v_\theta / r)}{\partial r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} \right) \right] \]

\[ + \mu \left[ \frac{4}{r} \left( \frac{\partial v_r}{\partial r} - \frac{v_r}{r} \right) - \frac{2}{r} \left( \frac{\partial v_\theta}{\partial \theta} + v_\theta \cot \theta \right) + \left( \frac{r \partial (v_\theta / r)}{\partial r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} \right) \cot \theta \right] + \rho g_r \]
Momentum Equation, $\dot{\theta}$-direction

\[
\rho \left( \frac{\partial v_\theta}{\partial t} + v_r \frac{\partial v_\theta}{\partial r} + \frac{v_\theta}{r} \frac{\partial v_\theta}{\partial \theta} + \frac{v_r v_\theta}{r} \right) = -\frac{\partial p}{\partial \theta} \tag{2.48}
\]

\[
+ \frac{\partial}{\partial r} \left[ \mu \left( r \frac{\partial (v_\theta/r)}{\partial r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} \right) \right] \\
+ \frac{1}{r} \frac{\partial}{\partial \theta} \left[ \frac{2\mu}{3r} \left( 2 \frac{\partial v_\theta}{\partial \theta} + v_r - \frac{\partial v_r}{\partial r} - v_\theta \cot \theta \right) \right] \\
+ \frac{\mu}{r} \left[ 3 \left( r \frac{\partial (v_\theta/r)}{\partial r} + \frac{1}{r} \frac{\partial v_r}{\partial \theta} \right) + \\
2 \left( \frac{1}{r} \frac{\partial v_\theta}{\partial \theta} - \frac{v_\theta}{r} \cot \theta \right) \cot \theta \right] + \rho g_\theta
\]

Temperature Equation

\[
\rho c_v \left( \frac{\partial T}{\partial t} + v_r \frac{\partial T}{\partial r} + \frac{v_\theta}{r} \frac{\partial T}{\partial \theta} \right) = \frac{\partial}{\partial r} \left( \kappa \frac{\partial T}{\partial r} \right) + \frac{2\kappa}{r} \frac{\partial T}{\partial r} \tag{2.49}
\]

\[
+ \frac{1}{r^2} \frac{\partial}{\partial \theta} \left( \kappa \frac{\partial T}{\partial \theta} \right) + \frac{1}{r^2} \cot \theta \kappa \frac{\partial T}{\partial \theta} \\
- p \left( \frac{\partial v_r}{\partial r} + 2 \frac{v_r}{r} + \frac{\partial (v_\theta/r)}{\partial \theta} + \frac{v_\theta}{r} \cot \theta \right) + \frac{\mu}{2} \phi_{ij} \phi_{ij}
\]

Development of the model equations in spherical coordinates follows the same steps as in cylindrical coordinates. The model variables are expanded using perturbation methods. The expansions are truncated according to the required order of approximation. Extra time derivatives are rewritten by way of a linear substitution, and the equations are rearranged to take the form of Equation 2.50. The model equations in spherical coordinates developed for numerical implementation are listed below as array components of the partial differential equation 2.50.

\[
\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}}{\partial r} + \frac{\partial \mathbf{G}}{\partial \theta} = \mathbf{SF} + \mathbf{SG} \tag{2.50}
\]
The combination of these equations of motion, Equations 2.50 and 2.51, the equation of state 2.5, and Sutherland’s formulae, Equations 2.6 and 2.7, gives a closed-form system of the model equations in spherical coordinates.

### 2.4 Summary

This chapter established the elements of a numerical model of nonlinear acoustic streaming in straight and tapered tubes. It cited the details of the experiment that serves as the basis for the geometries and parameters of the model. It de-
fined the coordinate systems and equations of motion, showing the derivation of the energy equation in terms of temperature. The model was made valid over the discrete time and space domain by way of the perturbation expansion of the dependent variables pressure, density, velocity, temperature, viscosity, and thermal conductivity. In so doing, the issue of discretization error was addressed, and round-off error was reduced through the exclusion of the static, zero-order portion of the model variables. The resulting model equations were arranged in a form applicable to their implementation in a finite-difference scheme. The equations of motion were described in cylindrical coordinates by Equations 2.39 and 2.40, and in spherical coordinates by Equations 2.50 and 2.51. Together with the equation of state 2.5 and Sutherland’s formulae, Equations 2.6 and 2.7, these provide a closed-form set of model equations that may be solved by numerical methods. A description of the finite-difference scheme used to implement these equations follows in Chapter 3.
Chapter 3

Numerical Implementation

This chapter describes the methods used to perform a time-marching numerical solution to the model equations. In the first section an algebraic grid-generation technique based on a hyperbolic-tangent function focuses grid refinement on the boundary regions near the outer wall and rigid end or ends of the tube. This section proceeds with a description of conformal mapping of the model equations from physical to computational domains and concludes with the particular grid arrangements for cylindrical- and spherical-coordinate geometries. Section 3.2 details the MacCormack-method finite-difference scheme and the partition of second derivatives and source terms using a split-step method in two-dimensional cylindrical and spherical coordinates. A discussion of boundary treatments follows in Section 3.3, which includes the definition of model variables at the boundaries for isothermal, adiabatic, slip, and no-slip conditions. The final section addresses domain decomposition and implementation of the numerical scheme using multiple processors.

3.1 Grid Refinement and Generation

The grid resolution requirements for modeling boundary-layer-driven streaming in a closed tube are not uniform over the entire computational domain. Near the outer wall and ends of the tube, large gradients in the flow variables due to interaction with solid surfaces require grid spacing smaller than the boundary layer thickness. In the middle of the tube, far from the tube walls and ends, the solution is dominated by first-order acoustic fluctuations and can be resolved by a much
more coarse grid. The larger grid spacing takes advantage of smaller gradients near the middle of the tube to reduce the size of the grid arrays and the number of calculations required per time step. A hyperbolic-tangent curve such as the one shown in Figure 3.1 provides a simple and appropriate mapping of a regular computational grid Ξ to a refined physical grid X with large spacing in the middle and small spacing at the ends.

![Hyperbolic tangent function mapping regular grid Ξ to refined grid X](image)

**Figure 3.1.** Hyperbolic tangent function mapping regular grid Ξ to refined grid X

The variables Ξ and X represent general normalized computational and physical domains in one dimension. The following section describes the mapping between computational and physical domains.

### 3.1.1 Grid-Refinement Method

Grid refinement using a hyperbolic-tangent function is carried out by way of the algebraic procedure described by Marcel Vinokur [58]. This method allows for asymmetric refinement where the physical grid spacing at one end may be different from that at the other. It is possible to define the physical grid spacing at either end to be the maximum so that the grid has fine resolution only at one end. The
ratios of regular grid spacing to refined grid spacing at the ends are defined as the slopes $S_0$ and $S_1$.

$$S_0 = \frac{\Delta \Xi_N}{\Delta X_N(0)} \quad S_1 = \frac{\Delta \Xi_N}{\Delta X_N(1)}$$  \hspace{1cm} (3.1)$$

These ratios are used to define two parameters, $A = \sqrt{S_0/S_1}$ and $B = \sqrt{S_0S_1}$, which allow for asymmetry in the hyperbolic-tangent function.

Equation 3.2 describes the normalized grid $X_N$ in terms of the parameter $A$ and the function $u$. 

$$X_N = \frac{u}{A + (1 - A)u}$$  \hspace{1cm} (3.2)$$

For the present grid requirements it is assumed that $B > 1$, in which case the function $u$ is defined as

$$u = \frac{1}{2} + \frac{\tanh[\Delta y(\Xi_N - \frac{1}{2})]}{2\tanh(\Delta y/2)},$$  \hspace{1cm} (3.3)$$

where $\Delta y$ is found by inverting the function 

$$B = \frac{\sinh(\Delta y)}{\Delta y}.$$  \hspace{1cm} (3.4)$$

This transcendental equation can be solved for $\Delta y$ by a polynomial approximation. For $B < 2.7829681$

$$\Delta y = \sqrt{6\bar{b}(1 - 0.15\bar{b} + 0.057321429\bar{b}^2 - 0.024707295\bar{b}^3 + 0.0077424461\bar{b}^4 - 0.0010794123\bar{b}^5)}$$  \hspace{1cm} (3.5)$$

where $\bar{b} = B - 1$, and for $B > 2.7829681$

$$\Delta y = v + (1 + \frac{1}{2})\ln 2v - 0.02041793 + 0.24902722w + 1.9496443w^2 - 2.6294547w^3 + 8.56795911w^4$$  \hspace{1cm} (3.6)$$

where $v = \ln B$ and $w = \frac{1}{B} - 0.028527431$. Given the regular computational grid $\Xi_N$ and the physical grid spacing at the ends, $\Delta X_N(0)$ and $\Delta X_N(1)$, this method provides a stretched grid $X_N$ normalized from 0 to 1 in a single dimension with
refinement at either end.

### 3.1.2 Grid Generation

A refined grid is generated using the above method by first defining a uniform grid with a regular interval of one nondimensional unit. The \( \xi \)-direction corresponds to the axial dimension of the physical tube model with a total of \( NI \) points. The \( \eta \)-direction corresponds to the radial dimension with \( NJ \) points.

\[
\begin{align*}
\xi &= 1, 2, 3, \ldots, NI \\
\eta &= 1, 2, 3, \ldots, NJ
\end{align*}
\] (3.7) (3.8)

When these variables are scaled from 0 to 1, their normalized counterparts take the place of the generalized uniform grid variable \( \Xi_N \) in Figure 3.1.

\[
\begin{align*}
\xi_N &= \frac{\xi - 1}{NI - 1} \quad (3.9) \\
\eta_N &= \frac{\eta - 1}{NJ - 1} \quad (3.10)
\end{align*}
\]

This gives the normalized uniform grid in the axial and radial dimensions. The normalized regular grid spacing is calculated similarly as the unit grid interval divided by the number of intervals for each dimension.

\[
\begin{align*}
\Delta \xi_N &= \frac{1}{NI - 1} \quad (3.11) \\
\Delta \eta_N &= \frac{1}{NJ - 1} \quad (3.12)
\end{align*}
\]

Each of these quantities for the uniform grid is used to generate the refined grid in a single dimension in either cylindrical or spherical coordinates. The normalized uniform grid variable \( \xi_N \) or \( \eta_N \) is substituted for \( \Xi_N \) in Equation 3.2, and the normalized regular interval \( \Delta \xi_N \) or \( \Delta \eta_N \) is substituted for \( \Delta \Xi_N \) in Equation 3.1 according to the direction. The remaining quantities needed to calculate the slopes \( S_0 \) and \( S_1 \) are the normalized physical grid intervals which are determined according to the requirements for each model.

Special attention is given to the axis of the tube where \( r = 0 \). At this point a
singularity results from the dependence on the inverse of the radius in the model equations in both cylindrical and spherical coordinates. It is therefore advantageous to set the grid points nearest the axis a distance of one half of a grid interval \( \Delta r \) away so as to avoid the point \( r = 0 \). This is shown in Figure 3.2. The strategies for handling this offset boundary are discussed below in Section 3.3.1.

![Figure 3.2. Grid near the axis showing an offset of \( \Delta r/2 \) to avoid a singularity at \( r = 0 \)](image)

### 3.1.2.1 Cylindrical Coordinates

In cylindrical coordinates the slopes \( S_0 \) and \( S_1 \) at the ends of the axial grid are

\[
S_0 = \frac{\Delta \xi_N}{\Delta z_N(0)} \quad \text{and} \quad S_1 = \frac{\Delta \xi_N}{\Delta z_N(1)}
\]

(3.13)

where \( \Delta z_N(0) \) and \( \Delta z_N(1) \) are grid intervals at the ends of the tube axis normalized by the length \( L \). The resulting normalized refined grid \( X_N \) is then multiplied by the tube length \( L \) to produce the refined grid in the \( \hat{z} \)-direction.

\[
z = LX_N
\]

(3.14)

In the radial direction the grid is offset from the axis by a distance of one half of the grid interval nearest the axis. The total length of the radial grid is the tube radius \( R \) minus the offset. The normalized physical grid intervals at the extremes
of the radial dimension are
\[ \Delta r_N(0) = \frac{\Delta r(0)}{R - \frac{1}{2}\Delta r(0)}, \quad \text{and} \quad \Delta r_N(1) = \frac{\Delta r(R)}{R - \frac{1}{2}\Delta r(0)}, \] 
(3.15)
and the slopes at the ends of the radial grid are
\[ S_0 = \frac{\Delta \eta_N}{\Delta r_N(0)}, \quad \text{and} \quad S_1 = \frac{\Delta \eta_N}{\Delta r_N(1)}. \] 
(3.16)

The refined radial grid is the normalized refined grid \( X_N \) from Equation 3.2 multiplied by the total physical length in the radial direction plus the offset from the axis.
\[ r = (R - \frac{1}{2}\Delta r(0))X_N + \frac{1}{2}\Delta r(0) \] 
(3.17)

### 3.1.2.2 Spherical Coordinates

In spherical coordinates the grid is constructed in the same manner as in cylindrical coordinates. Here the axial direction is denoted by the variable \( r \) as in Figure 2.3, so the physical grid in the axial direction is calculated as
\[ r = LX_N. \] 
(3.18)

The refined grid in the radial direction is defined as it is in cylindrical coordinates with the exception that the displacement from the tube axis is the angle \( \theta \) as shown in Figure 2.3. The total angle included in the radial domain extends from the offset at \( \theta = \frac{1}{2}\Delta \theta(0) \) to the tube wall at \( \theta = \Theta \). The normalized grid intervals in the \( \hat{\theta} \)-dimension near the tube axis and at the wall are
\[ \Delta \theta_N(0) = \frac{\Delta \theta(0)}{\Theta - \frac{1}{2}\Delta \theta(0)}, \quad \text{and} \quad \Delta \theta_N(1) = \frac{\Delta \theta(\Theta)}{\Theta - \frac{1}{2}\Delta \theta(0)}. \] 
(3.19)

The slopes from Equation 3.1 are
\[ S_0 = \frac{\Delta \eta_N}{\Delta \theta_N(0)}, \quad \text{and} \quad S_1 = \frac{\Delta \eta_N}{\Delta \theta_N(1)}. \] 
(3.20)
and the refined radial grid in spherical coordinates is given by

\[ \theta = (\Theta - \frac{1}{2} \Delta \theta(0))X_N + \frac{1}{2} \Delta \theta(0). \]  

(3.21)

With the refined grids defined in both the cylindrical and spherical coordinates, the next step is to transform the model equations to operate on the computational domain. This is accomplished through conformal mapping.

### 3.1.3 Conformal Mapping

Through grid refinement the computational cost of the numerical simulation is reduced in areas of small fluctuations of the flow variables and focused on areas of large fluctuations. The finite-difference operation, however, is valid only over a regular grid. To accommodate this restriction, the set of differential equations is conformed via the chain rule to provide a mapping between the regular and refined grids. Chapters 5.3 and 5.4 of John Anderson’s book, *Computational Fluid Dynamics*, describe this transformation and its theory [51]. Using this method Equations 2.39 and 2.50 are mapped to a partial differential equation on a uniform computational grid with grid variables \( \xi \) and \( \eta \). This takes the form

\[ \frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}}{\partial \xi} + \frac{\partial \mathbf{G}}{\partial \eta} = \mathbf{SF} + \mathbf{SG} \]  

(3.22)

where

\[
\begin{align*}
\mathbf{w} & = J \mathbf{w} \\
\mathbf{F} & = J \xi_x \mathbf{F} + J \xi_y \mathbf{G} \\
\mathbf{G} & = J \eta_x \mathbf{F} + J \eta_y \mathbf{G} \\
\mathbf{SF} & = J \mathbf{SF} \\
\mathbf{SG} & = J \mathbf{SG}.
\end{align*}
\]  

(3.23)

Here \( x \) and \( y \) represent generic orthogonal components of the physical grid, standing in for \( z \) and \( r \) in cylindrical coordinates, and \( r \) and \( \theta \) in spherical coordinates.
The Jacobian of the transformation is defined as

\[
J \equiv \begin{vmatrix}
\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\
\frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta}
\end{vmatrix} = \frac{\partial x \, \partial y}{\partial \xi \, \partial \eta} - \frac{\partial x}{\partial \eta} \frac{\partial y}{\partial \xi}.
\] (3.24)

This definition depends on the inverse metrics which may be written in subscript notation as \(x_\xi, y_\eta, x_\eta,\) and \(y_\xi\). The metrics are then related to the inverse metrics by way of the Jacobian.

\[
\begin{align*}
\xi_x &= \frac{\partial \xi}{\partial x} = \frac{\partial y}{\partial \eta} J^{-1} \\
\xi_y &= \frac{\partial \xi}{\partial y} = -\frac{\partial x}{\partial \eta} J^{-1} \\
\eta_x &= \frac{\partial \eta}{\partial x} = \frac{\partial y}{\partial \xi} J^{-1} \\
\eta_y &= \frac{\partial \eta}{\partial y} = \frac{\partial x}{\partial \xi} J^{-1}
\end{align*}
\] (3.25)

The inverse metrics are calculated numerically from the generated grids in cylindrical and spherical coordinates using second-order centered differencing with one-sided finite differencing at the edges. Since the grid refinement algorithm in each dimension is independent of the grid in the orthogonal dimension, two of the inverse metrics are identically zero.

\[
\begin{align*}
\left. \frac{\partial x}{\partial \xi} \right|_i &= \frac{x_{i+1} - x_{i-1}}{2\Delta \xi} \\
\left. \frac{\partial x}{\partial \eta} \right|_j &= 0 \\
\left. \frac{\partial y}{\partial \xi} \right|_i &= 0 \\
\left. \frac{\partial y}{\partial \eta} \right|_j &= \frac{y_{j+1} - y_{j-1}}{2\Delta \eta}
\end{align*}
\] (3.26)

Consequently, the Jacobian simplifies to

\[
J = \frac{\partial x \, \partial y}{\partial \xi \, \partial \eta},
\] (3.27)
and $F$ and $G$ in Equation 3.23 become

$$\bar{F} = J \xi_x F \quad \text{and} \quad \bar{G} = J \eta_y G \, .$$

(3.28)

Applying this transformation to the model equations prepares them for the time-stepped solution by way of finite differencing.

### 3.2 Finite-Difference Scheme

The solution to the model equations is propagated from one time step to the next by a MacCormack-type two-step predictor-corrector method. The version used in this research is a basic scheme that is second-order in time and second-order in space. This is chosen for simplicity and convenience and because it allows a higher ratio of time step to spatial step than the fourth-order-in-space method for similar accuracy [59]. The ratio relating time step $\Delta t$ and spatial step $\Delta x$ is called the Courant number, which is defined as

$$C = \frac{c_0 \Delta t}{\Delta x} \quad (3.29)$$

where $c_0$ is the speed of propagation. By the Courant-Friedrichs-Lewy condition, this number must be less than or equal to 1 in order to ensure stability in hyperbolic equations [51]. The model of a tapered tube must be stable at both the wide end and the narrow end, and the grid spacing in the radial direction at the wide end needs to be small enough to resolve the boundary layer near the tube wall. This results in a minimum radial grid spacing at the narrow end that is smaller than the corresponding interval at the wide end. To meet the CFL condition for stability at the narrow end, the time step is limited by the Courant number. A numerical scheme that allows a higher Courant number helps to alleviate restrictions on the size of the time step due to the smaller spatial step incurred by the tapered tube.

The first stage of the MacCormack method is the predictor step which computes the set $\bar{W}$ of model variables from one dimension of Equation 3.22 as a temporary array denoted by the superscript $\bar{n}$. This step uses a forward difference in space with respect to the direction $d$, described below, to make a prediction about the time derivative. This derivative is multiplied by the time step $\Delta t$ and added to
The corrector step computes a new estimate of the time derivative by way of a backward difference in space with respect to \( d \) using the predicted array. The next time step \( w^{n+1} \) is found by averaging the predicted and corrected estimates of the time derivative, multiplying that average by \( \Delta t \), and adding the result to the original \( w^n \).

\[
\tilde{w}^{n+1}_i = \frac{1}{2} \left( w^n_i + \tilde{w}^{n}_i - d \frac{\Delta t}{\Delta \zeta} (\tilde{F}^{n}_i - \tilde{F}^{n}_{i-d}) + \Delta t SF^n_i \right)
\]  

(3.31)

Equation 3.31 shows this operation as it is implemented in the numerical model. Its theory and derivation are described in detail by Anderson [51].

The direction variable \( d \) in Equations 3.30 and 3.31 provides a way to reverse the order of the forward and backward spatial differences. Letting \( d = 1 \) gives the order as described above, whereas \( d = -1 \) reverses it. Alternating these two versions is necessary to maintain accuracy in nonlinear problems [60].

### 3.2.1 Split-Step Method

The finite-difference procedure described above can be represented concisely by the symbolic operator \( L_\zeta(\Delta t) \). This denotes the solution to the equation

\[
\frac{\partial \bar{w}}{\partial t} + \frac{\partial F}{\partial \zeta} = SF
\]

(3.32)

such that \( L_\zeta(\Delta t) \) advances \( \bar{w} \) one time step in the \( \zeta \)-direction.

\[
\bar{w}(t + \Delta t) = L_\zeta(\Delta t)\bar{w}(t)
\]

(3.33)

Likewise \( L_\eta(\Delta t) \) gives the solution operator in the \( \eta \)-direction to

\[
\frac{\partial \bar{w}}{\partial t} + \frac{\partial G}{\partial \eta} = SG
\]

(3.34)
so that
\[
\mathbf{w}(t + \Delta t) = L_\eta(\Delta t)\mathbf{w}(t) .
\] (3.35)

Together these can operate to advance the solution of the two-dimensional system in Equation 3.22 using a time-splitting technique [61].

When the \( \xi \)-direction and \( \eta \)-direction operators are applied in sequence, the combination provides a solution to the two-dimensional system advanced by one time step. Alternating the sequence as in the formula
\[
\mathbf{w}(t + 2\Delta t) = L_\xi(\Delta t)L_\eta(\Delta t)L_\eta(\Delta t)L_\xi(\Delta t)\mathbf{w}(t)
\] (3.36)
steps the solution through two time steps and assures second-order accuracy in time [61, 59]. To implement this balanced arrangement with the alternating versions of the MacCormack two-step method described above, the value of \( d \) is set to 1 for the first pair of operators, \( L_\xi(\Delta t)L_\eta(\Delta t) \), and subsequently changed to \(-1\) for the second pair, \( L_\eta(\Delta t)L_\xi(\Delta t) \). This order is reversed for the next iteration of Equation 3.36 with \( d \) set to \(-1\) for the first pair of operators and reset to 1 for the second pair. The resulting solution at each step is accurate to second order in both time and space for the nonlinear model.

### 3.2.2 Second Derivatives and Source Terms

Calculating dissipation in the Navier-Stokes equations requires additional measures to maintain second-order accuracy. The second derivatives associated with coefficients of viscosity and thermal conductivity appear in the model equations 2.51 and 2.40 as gradients within the \( F \) and \( G \) arrays. These arrays are the arguments to spatial derivatives in the axial and radial directions respectively. Those gradients along the dimension orthogonal to these directions are computed using a second-order centered-difference method. For the \( F \) array, the gradients along the radial dimension are center differenced, and for the \( G \) array, the gradients along the axial dimension are center differenced.

The remaining gradients along the same dimension as the main spatial derivatives are computed using forward or backward finite-difference calculations, but the direction is chosen to be the reverse of the main derivative. For instance, during
the predictor step when $d = 1$, the $F$ array is subject to a forward-difference operation in the $\xi$-direction. The gradients in that same direction must be computed as backward differences to maintain second-order accuracy [51].

Gradients in the arrays of source terms, $\mathbf{SF}$ and $\mathbf{SG}$, are treated in the same manner. Those that are in the direction orthogonal to the main finite-difference operator, $L_\xi$ or $L_\eta$, are calculated using a second-order centered-difference scheme. Those in the same direction are computed with forward or backward differencing such that it is the reverse of the direction $d$ in use at that time by the operator $L$.

### 3.3 Boundary Treatment

Three types of boundaries surround the computational domain of the tube model. Along the axis is a symmetric boundary which represents the mirror image of the fluid dynamics occurring within the domain. Opposite the axis is the rigid tube wall. This boundary presents a reflecting surface for acoustic pressure and normal velocity, but could allow slip or no-slip conditions for tangential velocity and adiabatic or isothermal conditions for acoustic fluctuations in temperature. These options for the rigid boundary are useful for testing the model as well as for exploring the dynamics of streaming. The remaining boundaries lie at the ends of the tube. They can represent either a rigid end cap with the same properties as the rigid tube wall or a non-reflecting boundary where acoustic disturbances disappear with minimal trace of reflection as if passing through into an adjacent computational domain. This section presents a selection of boundary treatment methods. The specific boundaries used for each simulation are listed with the results in Chapter 5.

At the rigid-wall boundary no fluid motion occurs normal to the surface. Velocity on the wall in the normal direction is defined as zero. Velocity tangent to the surface is also defined as zero if there is a no-slip condition imposed. If fluid is allowed to slide unimpeded along the surface, i.e. in an inviscid model, the tangential velocity is calculated using a one-sided second-order Neumann boundary treatment as described by Anderson [51].

$$
\left. \left( \frac{\partial u}{\partial y} \right) \right|_1 = \frac{-3u_1 + 4u_2 - u_3}{2\Delta y} + O(\Delta y)^2
$$

(3.37)
In Equation 3.37 the gradient in the $y$-direction at $u_1$ on the boundary may be defined as zero to represent the absence of shear. Then with information from $u_2$ and $u_3$, the value of $u_1$ is easily determined. The one-sided Neumann algorithm is also used to determine fluctuating pressure on the rigid boundary since the normal pressure gradient is known to be zero at that point.

If the thermal boundary condition is considered to be isothermal at the surface, then the temperature is defined to be that of the surface. In this model the isothermal wall is assumed to be the same temperature as the ambient gas, and so fluctuating temperature on the boundary is defined as zero. On the other hand, if the surface is considered as an adiabatic boundary, the temperature of the wall is assumed to fluctuate along with the temperature of the adjacent gas. The same one-sided Neumann method is used to hold the normal gradient of temperature to zero at that point.

The non-reflecting boundary uses a Method-of-Characteristics (MOC) technique to calculate the values of field variables on the boundary points so that they match outgoing fluctuations without introducing fluctuations into the domain. Appendix B details the derivation of the MOC boundary using the perturbed primitive variables for density, velocity, and temperature. This technique is valid for linear and nonlinear propagating disturbances but does not take into account dissipation [62]. Care must be taken near this boundary so that any reflected fluctuations due to dissipative effects do not interfere with the solution in question. The present model makes use of a buffer zone near the MOC boundary and is arranged so that outgoing fluctuations are small in that area as compared with the rest of the model. This configuration is similar to the work of Boluriaan [30].

### 3.3.1 One-Sided Axisymmetric Boundary

At the axis of the tube in both cylindrical and spherical coordinate systems, the radial coordinate is zero, resulting in a singularity at the axis due to $r^{-1}$ or $(\sin \theta)^{-1}$ radial dependence in the model equations. Here this problem is addressed by positioning the grid points nearest the axis a distance of one-half grid space away from the axis and by using a one-sided finite-difference calculation to determine the values of the flow variables at those points. The following derivation of this
method for points offset from a boundary is based on the description of one-sided finite differencing shown by Anderson [51].

In this method a model variable \( u \) is approximated by the second-order polynomial expression,
\[
u = a + b\eta + c\eta^2,
\]
where \( \eta \) is the distance from the axis. Figure 3.3 shows the spacing of grid points near the axis. The point labeled \( u_0 \) lying directly on the axis is used only for the derivation of the offset one-sided finite-difference operation and is not part of the computational domain which begins at \( u_1 \). Assigning values for \( \eta \) as the distance from the axis to each grid point yields a set of three equations with three unknowns.

\[
\begin{align*}
u_0 &= a + b(0) + c(0)^2 \\
u_2 &= a + b\left(\frac{3}{2}\Delta\eta\right) + c\left(\frac{3}{2}\Delta\eta\right)^2 \\
u_3 &= a + b\left(\frac{5}{2}\Delta\eta\right) + c\left(\frac{5}{2}\Delta\eta\right)^2
\end{align*}
\]

Considering the model flow quantities at the axis, namely pressure, density, temperature, axial velocity, and radial velocity, each is assumed to be exactly the same from one side of the axis to the other with the exception of radial velocity,
which is exactly opposite across the axis. For this exception the Dirichlet boundary treatment imposes a zero radial velocity at the axis. For the other quantities the Neumann boundary treatment maintains a zero gradient across the axis. The following sections derive these two operations for the boundary point offset from the axis resulting in a value for \( u_1 \).

### 3.3.1.1 Dirichlet Condition for an Offset Boundary

By definition the Dirichlet boundary condition specifies a value \( \alpha \) at the boundary so that the first of Equations 3.39 becomes

\[
  u_0 = a = \alpha. \tag{3.40}
\]

This definition allows the value to vary spatially along the boundary, but here only a constant \( \alpha \) is considered [63]. Given \( u_2 \) and \( u_3 \) from the solution for the inner domain, the remaining equations yield expressions for the coefficients \( b \) and \( c \).

\[
b = \frac{25u_2 - 9u_3 - 16\alpha}{15\Delta\eta} \tag{3.41}
\]

\[
c = \frac{-10u_2 + 6u_3 + 4\alpha}{15\Delta\eta^2} \tag{3.42}
\]

The offset boundary point \( u_1 \) at \( \eta = \Delta\eta/2 \) is then defined by \( u_2 \), \( u_3 \), and \( \alpha \) through the expression

\[
u_1 = \frac{2}{3} u_2 - \frac{1}{5} u_3 + \frac{8}{15} \alpha, \tag{3.43}
\]

and for the case where \( \alpha = 0 \),

\[
u_1 = \frac{2}{3} u_2 - \frac{1}{5} u_3. \tag{3.44}
\]

### 3.3.1.2 Neumann Condition for an Offset Boundary

The definition of the Neumann boundary condition specifies that the gradient of the solution is some determined value at the boundary. Taking the derivative of equation 3.38 gives the gradient

\[
  \frac{\partial u}{\partial \eta} = b + 2c\eta. \tag{3.45}
\]
Considering its value at the point on the axis \( \eta = 0 \) further simplifies the expression to

\[
\frac{\partial u}{\partial \eta} \bigg|_{0} = b = \frac{\alpha}{\Delta \eta},
\]  

where \( b \) is defined for convenience as the constant \( \alpha/\Delta \eta \). Solving the rest of the system of equations gives expressions for the coefficients \( a \) and \( c \).

\[
a = \frac{25u_2 - 9u_3 - 5\alpha}{16},
\]  

\[
c = \frac{-u_2 + u_3 - \alpha}{4\Delta \eta^2}
\]  

Using these equations allows the offset boundary point \( u_1 \) at \( \eta = \Delta \eta/2 \) to be written as

\[
u_1 = \frac{3}{2}u_2 - \frac{1}{2}u_3 + \frac{1}{8}\alpha,
\]  

and for a zero gradient at the boundary,

\[
u_1 = \frac{3}{2}u_2 - \frac{1}{2}u_3.
\]

This method for computing the Dirichlet and Neumann conditions for an offset boundary at the axis not only avoids the singular point directly on the axis but also allows coarse grid resolution at that boundary of the computational domain. Previous attempts at an axisymmetric boundary using other methods to avoid a singularity required fine grid resolution near the axis to ensure numerical stability. The arrangement described here allows fewer grid points with no stability issues near the axis.

### 3.4 Multi-Processor Computation

Parallel implementation shortens the computation time for each step of the simulation by distributing the solution arrays among multiple processors. In a rectangular domain with a large length-to-width ratio, a one-dimensional domain decomposition is a simple and effective choice. Here the computational domain is distributed to multiple processors only along the axial dimension. The total number of grid points in that direction, \( NI \), is divided evenly among \( n \) processors with any re-
mainining grid points being assigned to the last node. The master node, labeled 0

\[ \begin{array}{cccccc}
0 & 1 & \ldots & n-2 & n-1 & \text{NI} \\
\end{array} \]

\textbf{Tube axis}

\textbf{Figure 3.4.} Decomposition of computational domain into \( n \) nodes, labeled 0 to \( n - 1 \) in Figure 3.4, handles the file input and output for the entire domain, distributing data from the input files to all of the processors and gathering data from them to write to the solution files.

Communication between the subdomains is carried out by way of the Message Passing Interface (MPI), a standardized library of routines for parallel computers [64]. The procedure for sharing information between adjacent subdomains is carried out at the same time as the other boundary treatment operations. Data from the first interior array of points is copied to the edge array of the neighboring subdomain as shown in Figure 3.5. Since the template for the second-order finite-difference method is three grid points wide, the overlap between domains requires only two grid points in common in the \( \xi \)-direction. Once the shared edges of the subdomains are defined with data supplied by the neighboring subdomains, the simulation carries on to the next time step.

\[ \begin{array}{c|c|c}
A & \quad & B \\
\xi: & i & i+1 & i & i+1 \\
\end{array} \]

\textbf{Figure 3.5.} Message passing between adjacent subdomains A and B
3.5 Summary

This chapter presented the details of executing the model equations in a time-stepping finite-difference algorithm. It addressed the computational challenges of small time steps and wide ranges of spatial and temporal scales by refining the physical grid to high resolution only where it is required along solid boundaries and by distributing the computational domain among multiple processors to reduce the time of computation. Second-order accuracy was maintained in the two-dimensional finite-difference algorithm through the use of a balanced split-step method and through careful arrangement of second derivatives and source terms. The use of an offset one-sided boundary treatment near the tube axis further relaxed the grid resolution requirements while avoiding problems with numerical instabilities. The following chapter describes the methods used to simulate acoustic sources, calculate acoustic streaming velocities, and compare those velocities between models of different geometries.
Chapter 4

Simulation Method

The model equations are presented above in Chapter 2, and the details of their implementation in a finite-difference algorithm are described in Chapter 3. The remaining steps to realize the numerical model lie in the method of generating standing waves in the model domain and calculating streaming velocities. This chapter provides the final steps to simulate streaming in straight and tapered tubes. The method of exciting acoustic oscillation is shown in Section 4.1 with two arrangements that produce standing waves. The specific geometric parameters for each model under consideration are presented along with the methods used to choose them. Section 4.3 describes the calculation of the Lagrangian streaming velocity, and the final section offers a normalization scheme to facilitate comparison of the axial streaming velocities between tubes of various taper angles.

4.1 Excitation of Standing Waves

To model boundary-layer-driven streaming for comparison between straight and tapered tubes an efficient and reliable method of generating standing waves is needed. An efficient method brings the acoustic field into a steady state of oscillation with the fewest possible acoustic cycles and hence minimal computational effort. A reliable method generates a standing wave with a predictable amplitude, well-defined phasing, and minimal steepening due to nonlinearity at high amplitude. A pure sinusoidal standing wave isolates boundary-layer interactions as the driving mechanism of streaming and excludes the influence of a traveling-wave component as described above in Section 1.1.3. An excitation method that meets
these requirements is the two-line source used by Boluriaan and Morris in their simulations of acoustic streaming [30]. This method is described below along with a variation using a single source and a reflective surface to generate a standing wave.

4.1.1 Line Source

A fluctuating pressure source $P_0 \sin \omega t$ is added to the model at a specific position $z_s$ along the axial coordinate. Its amplitude is localized at that position by a Gaussian envelope in the axial direction, while in the radial direction the amplitude is constant. The expression for the time derivative of the pressure source with a Gaussian envelope is

$$S_p = P_0 \exp \left[ -\frac{1}{2} \left( \frac{z - z_s}{\alpha} \right)^2 \right] \omega \sin \omega t,$$

where the width $\alpha$ of the Gaussian function is chosen to be six times the maximum grid spacing in the axial direction. This allows adequate spatial resolution for a compact, yet smooth transition from the localized source to the surrounding field. This pressure source function is added to the model equations as a vector with components in the equations of mass conservation and energy conservation. The pressure-source function is scaled by linear acoustic approximations to provide fluctuating density and fluctuating temperature components, both of which are necessary to ensure only acoustic-mode excitation. The source vector $\mathbf{S}$, defined by

$$\mathbf{S} = \begin{bmatrix} S_p/c^2 \\ 0 \\ 0 \\ S_p/(\rho_0 c_p) \end{bmatrix},$$

is added to the source term in the axial direction of the partial differential equation. In the split-step implementation this is Equation 3.32, which becomes

$$\frac{\partial \mathbf{W}}{\partial t} + \frac{\partial \mathbf{F}}{\partial \xi} = \mathbf{SF} + \mathbf{S},$$

(4.3)
where $\mathbf{S} = J \mathbf{S}$. A standing wave may be achieved through the judicious choice of boundary conditions and line-source locations along the tube axis as described below.

### 4.1.2 Two-Source Method

The method described by Boluriaan and Morris establishes a standing wave between two in-phase line sources located one and a half wavelengths apart in a straight tube [30]. The total length of the tube is two wavelengths, and the boundary conditions present non-reflecting boundaries at the tube ends using the Method-of-Characteristics boundary condition of Thompson [62]. The tube wall is modeled as a rigid boundary. The MOC boundary condition is mentioned above in Section 3.3, and its implementation is described in detail in Appendix B.

To introduce two line sources numerically along the $z$-axis of a straight tube, the source pressure function $S_p$ in Equation 4.1 is redefined as the sum of two Gaussian envelopes centered at positions $z_{s1}$ and $z_{s2}$ as in Equation 4.4.

$$S_p' = S_p(z_{s1}) + S_p(z_{s2}) \quad (4.4)$$

This new source pressure is then used in place of $S_p$ to define the source vector $\mathbf{S}$, which is added to the partial differential equation, Equation 3.32, as before.

The resulting standing wave is shown in Figure 4.1. This figure is generated from a linear, inviscid, one-dimensional simulation. Between the sources, the forward-going wave from the source at $z = 0.25\lambda$ interferes constructively with the reverse-going wave from the source at $z = 1.75\lambda$. At the ends of the tube, the initial disturbances from the sources meet the non-reflective boundaries and do not return. Once the waves from the two sources combine outside of the region between them, they cancel due to their wavelength relative to the distance between the sources.

This arrangement has a distinct advantage for nonlinear oscillations at high-amplitude. Since the disturbances from each source propagate a distance less than two wavelengths before meeting a non-reflecting end, there is little effect from wave steepening if the shock distance is large compared to the tube length. Such a source method is ideal for studying Rayleigh streaming at high Reynolds number [30].
4.1.3 Reflected-Source Method

A variation on the two-source method cuts this arrangement in half by placing a rigid boundary at one end of the tube and locating a single line source a certain distance away depending on the wavelength of the acoustic oscillation and the taper angle of the tube. The method to calculate this distance is described below in Section 4.2. Between the source and the rigid end the incident wave from the source interferes constructively with the reflected wave from the rigid end and results in a standing wave. Beyond the source in the other direction, the reflected wave from the rigid end cancels the disturbance emanating from the source. Opposite the rigid end is a non-reflecting end from which neither the direct nor the reflected
wave returns.

This arrangement supports a standing wave in a shorter length of tube than the two-source method. A shorter tube presents a potentially smaller computational domain not only in the axial direction but also in the radial direction when the tube is tapered. For a given taper angle the narrow end of the tube is limited by how much it can converge relative to the viscous penetration depth. If the tapered tube is sufficiently long, the radius at the wide end may be much larger than the radius in the region of interest where streaming occurs, and this extra volume may incur a heavy cost in computational grid resolution. A reduction in the axial dimension of a tapered tube also reduces its extent radially and eases the computational burden.

Figure 4.2. Normalized velocity and pressure of a standing wave generated by a single source at $z/\lambda = 0.25$ in a straight tube with one rigid end and one non-reflecting end. The curves are shown at times of maximum excursion indicated by the phase angle $\phi$. 
Figure 4.2 shows an implementation of the reflected-source method in a straight tube to generate a quarter-wavelength standing wave in a tube $3/8\lambda$ long. This linear, one-dimensional, inviscid model illustrates the constructive interaction of the incident and reflected waves between the line source at $z = \lambda/4$ and the rigid end at $z = 0$. On the right side of the figure between the source and the non-reflecting boundary, the destructive interference can be seen as cancellation in both pressure and velocity.

![Graph of velocity at $\phi = \pi/2$ (solid) and $3\pi/2$ (dashed)](image1)

![Graph of pressure at $\phi = 0$ (solid) and $\pi$ (dashed)](image2)

**Figure 4.3.** Normalized velocity and pressure of a standing wave generated by a single source at $r/\lambda = 0.56$ in a tapered tube with one rigid end and one non-reflecting end. The curves are shown at times of maximum excursion indicated by the phase angle $\phi$.

A similar resonance can be achieved in a tapered tube. Figure 4.3 shows a one-dimensional model of a tube with spherical spreading to simulate a slight taper with the rigid end of the tube a distance $0.2\lambda$ from the origin of the spherical coordinate system. Here the source is $0.36\lambda$ from the rigid end, which is longer
than the corresponding distance in the straight tube. The method of determining the dimensions for the model tubes is given in detail in the following section.

4.2 Model Tube Dimensions

To isolate the effect of taper angle on acoustic streaming, careful consideration is given to the parameters of the tube design. First, the axial location and radial length of transverse profiles of streaming velocity are chosen to be comparable between tubes of varying taper angles. The axial streaming velocity is considered to have its maximum magnitude near the midpoint of a streaming vortex, which is typically situated between the velocity node and antinode of a standing wave as shown in Figure 1.1. In the standing waves depicted in Figures 4.2 and 4.3 this location is assumed to be one eighth of a wavelength from the rigid reflective end of the model tubes. A survey of numerically generated streaming patterns in variously tapered tubes supports this assumption. The first design criterion is to make the length of these transverse profiles equal between the tapered and straight tubes.

This profile length is determined by the constraint on the width of the narrow end of the most highly tapered tube under consideration. In keeping with the boundary-layer assumption held by Olson and Swift, the width of the narrow end of the 0.8° tapered tube, being the largest taper angle under consideration in this study, is chosen to be no smaller than the narrow end of the most highly tapered pulse tube used by Olson and Swift [1]. This width along with the taper angle determines the width of the tube $\lambda/8$ away at the location of the streaming profile. The radius of the tube at that point is calculated to be 3.75 cm. This is the radius used for the straight cylindrical tube as well.

Holding the radius of the streaming profile constant among the variously tapered tubes at a distance of $\lambda/8$ from the rigid narrow end allows the calculation of the distance from the narrow end to the origin of the spherical coordinate system in each of the models. This distance is indicated by $R_1$ in the diagram shown in Figure 4.4 and is listed for each tapered tube in Table 4.1.

Another design requirement places a short region between the line source and the non-reflecting end to act as a buffer between the boundary and areas where
dissipative effects are large. Since the MOC boundary accounts for linear and non-linear incident waves but neglects dissipation, there may be incoming disturbances from the dynamics of the thermal and viscous boundary layer adjacent to the tube wall at the non-reflecting end. To minimize this possibility the reflected wave from the rigid end and the positive-going wave from the line source are allowed to cancel for a distance of $\lambda/8$ before encountering the non-reflecting end. This reduces the incident amplitude considerably and provides a space for any aberrant reflections from the wide end of the tube to decay before interacting with the standing wave. The length of the tube is defined by the position of the line source minus the position of the rigid end plus a one-eighth-wavelength buffer, $L = R_S - R_1 + \lambda/8$.

The remaining step is to find the location of the line source $R_S$ that produces a standing wave in a tube with one rigid end and one non-reflecting end. In a straight tube a source position one quarter wavelength from the rigid end establishes a standing wave between the source and rigid end and nearly complete cancellation between the source and non-reflecting end as shown in Figure 4.2. In a tapered tube this distance is found by way of spherical Bessel functions. These functions form a solution to the wave equation in spherical coordinates. For waves propagating only in the radial direction between two boundaries the wave equation takes the form of Bessel’s equation of order $1/2$. The solution of Bessel’s equation for the acoustic pressure can be expressed as a linear combination of spherical Bessel and
Neumann functions of order 0 such that

\[ R(r) = A j_0(kr) + B n_0(kr), \]  

where \( R(r) \) is the radial portion of the separable solution to the wave equation [65], \( k \) is the wavenumber, \( r \) is the radial coordinate, \( j_0 \) and \( n_0 \) are the spherical Bessel and Neumann functions of order 0, and \( A \) and \( B \) are coefficients.

Equation 4.5 may be solved in terms of acoustic pressure, and so the boundary values are applied according to what is known about the acoustic pressure at \( r = R_1 \) and \( r = R_S \). At \( R_1 \) the gradient of pressure vanishes due to interaction with the rigid surface. Taking the derivative with respect to \( r \), writing the solution \( R(r) \) as pressure \( p \), and evaluating at \( r = R_1 \) yields

\[ \left. \frac{\partial p}{\partial r} \right|_{r=R_1} = Ak j'_0(kR_1) + Bkn'_0(kR_1) = 0 \]  

for the rigid boundary condition. Using a recursion relation for the derivatives of the spherical Bessel and Neumann functions allows them to be expressed as functions of first order.

\[ j'_0(x) = -j_1(x) \]  
\[ n'_0(x) = -n_1(x) \]  

Substituting these into Equation 4.6 completes the expression for the solution at \( r = R_1 \).

\[ A j_1(kR_1) + B n_1(kR_1) = 0 \]  

At the position of the source \( r = R_S \), a node occurs in the pressure standing wave. If dissipation is neglected, the acoustic pressure disappears at that point so that \( p = 0 \). The solution at \( R_S \) is expressed from Equation 4.5 as

\[ A j_0(kR_S) + B n_0(kR_S) = 0. \]
Table 4.1. Parameters of Tube Geometries

<table>
<thead>
<tr>
<th>Total included angle [deg]</th>
<th>Taper [deg]</th>
<th>$R_1$ [m]</th>
<th>$R_{ST}$ [m]</th>
<th>$R_S$ [m]</th>
<th>Tube Length [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>N/A</td>
<td>1.040</td>
<td>2.080</td>
<td>3.120</td>
</tr>
<tr>
<td>0.4</td>
<td>0.2</td>
<td>9.707</td>
<td>10.747</td>
<td>11.967</td>
<td>3.300</td>
</tr>
<tr>
<td>0.8</td>
<td>0.4</td>
<td>4.334</td>
<td>5.374</td>
<td>6.807</td>
<td>3.513</td>
</tr>
<tr>
<td>1.2</td>
<td>0.6</td>
<td>2.542</td>
<td>3.582</td>
<td>5.259</td>
<td>3.756</td>
</tr>
<tr>
<td>1.6</td>
<td>0.8</td>
<td>1.647</td>
<td>2.687</td>
<td>4.624</td>
<td>4.017</td>
</tr>
</tbody>
</table>

Equations 4.8 and 4.9 form a system of equations

\[
\begin{bmatrix}
  j_1(kR_1) & n_1(kR_1) \\
  j_0(kR_S) & n_0(kR_S)
\end{bmatrix}
\begin{bmatrix}
  A \\
  B
\end{bmatrix} =
\begin{bmatrix}
  0 \\
  0
\end{bmatrix}
\]  
(4.10)

which can be solved for $R_S$ in terms of $R_1$ by setting the determinant equal to 0. This is expressed as

\[
\left| j_1(kR_1) n_1(kR_1) \\
  j_0(kR_S) n_0(kR_S)
\right| = 0,
\]  
(4.11)

or equivalently

\[
j_1(kR_1)n_0(kR_S) - n_1(kR_1)j_0(kR_S) = 0.
\]  
(4.12)

Given the distance $R_1$, the position of the source $R_S$ may be found numerically with the help of mathematical software such as Mathematica [66]. The resulting parameters for each of the model tubes are shown in Table 4.1, and the simulations of acoustic streaming using these model tubes are presented in Chapter 5.

4.3 Calculation of Streaming Velocity

Once a steady standing wave is established in each of the tube models, a time-averaged streaming velocity is calculated over many periods of oscillation. Yano shows that the second-order Lagrangian velocity representing streaming is calculated as the time average of mass flux divided by the ambient density [48]. The
axial streaming velocity $u_{st}$ and radial streaming velocity $v_{st}$ are shown in the following equations:

$$u_{st} = \frac{\langle \rho u \rangle}{\rho_0} \quad (4.13)$$

$$v_{st} = \frac{\langle \rho v \rangle}{\rho_0}, \quad (4.14)$$

where the bracket notation $\langle \rangle$ denotes the time average over an integer number of periods. If the density is expanded into its ambient and perturbed quantities, the Lagrangian streaming velocity can be written as the sum of Eulerian streaming velocity and a second-order mean mass flux divided by density.

$$v_{st} = \frac{\langle (\rho_0 + \rho')v \rangle}{\rho_0} \quad (4.15)$$

$$v_{st} = \langle v \rangle + \frac{\langle \rho'v \rangle}{\rho_0} \quad (4.16)$$

Since first-order velocity represents acoustic fluctuation, its time average is zero leaving only the second-order Eulerian streaming velocity as the first term. Neglecting quantities smaller than second order allows the Lagrangian streaming velocity to be written

$$v_{st} = \langle v_2 \rangle + \frac{\langle \rho_1v_1 \rangle}{\rho_0}. \quad (4.17)$$

Nyborg notes that this second term is often referred to as the velocity transform $v_T = \rho_0^{-1}\langle \rho_1v_1 \rangle$ [7].

In a pure standing wave where the phase between acoustic density and acoustic velocity is $\pi/2$ everywhere, the Lagrangian and Eulerian descriptions of acoustic streaming are equivalent because the second-order mass flux $\rho_1v_1$ averages to zero over time. It is only when the acoustic field deviates from a standing wave that the velocity transform comes into play, and the Eulerian and Lagrangian streaming velocities are distinct. For this reason the velocity transform term $v_T$ is important in the calculation of traveling-wave streaming, and Eulerian streaming velocity $\langle v_2 \rangle$ is important in dissipation-driven streaming [41].

The Lagrangian streaming velocity is computed in the present numerical model by an average of mass flux $\rho v$ over the number of time steps for each period of oscillation. The mean mass flux for each period is written to a file as an array
representing the spatial domain of the model. A separate program reads ten of these files, averages the arrays over the number of files and divides by the ambient density. This program, meanACS.fpp, is listed in Section D.5.3.

The result is a set of streaming velocities $u_{st}$ and $v_{st}$ over ten periods of oscillation covering the extent of the model tube. The streaming velocity may be examined at any spatial point, but the transverse profiles at $r = R_{ST}$ in the tapered tube model or $z = Z_{ST}$ in the straight tube model are of equal length for the purpose of direct comparison.

4.4 Normalization

Following the method of Olson and Swift [1] described in Section 2.1, the source amplitude $P_0$ shown in Equation 4.1 is adjusted for each open-closed model so that the pressure amplitude at the closed end of the tube is maintained near $1.95 \times 10^5$ Pa. With the geometrical parameters listed above, however, this method does not necessarily produce a direct comparison of streaming between straight and tapered tubes at the axial positions $Z_{ST}$ and $R_{ST}$. Normalizing with respect to acoustic velocity amplitude where the axial streaming velocity profiles are observed ensures that any variations due to geometry, source amplitude, or discretization do not influence the comparison of streaming between tubes of varying cross-sectional area.

Examination of the theoretical prediction for axial streaming velocity by Schuster and Matz, shown in Equation 5.3, reveals a dependence on the square of acoustic velocity amplitude $U_1$. Dividing this equation by $|U_1|^2/c$ removes that dependence and renders the expression dimensionless. For the purposes of this study to compare streaming in straight and tapered tubes, a representative axial acoustic velocity amplitude is needed that is particular to the cross section under consideration and is independent of variations in the radial direction.

Such an amplitude is found by first calculating the spatial average of mass flux along the cross-sectional area of the tube at the axial position of the streaming profile and then dividing by the ambient density. For a straight tube the instantaneous
mass flux is found by
\[ \dot{m} = \int_0^R 2\pi r u(Z_{ST}, r, t) (\rho_0 + \rho'(Z_{ST}, r, t)) \, dr, \] 
(4.18)
and its spatial average divided by ambient density yields the time-dependent representative acoustic velocity,
\[ U_{Rep}(t) = \frac{2}{\rho_0 R^2} \int_0^R u(Z_{ST}, r, t) (\rho_0 + \rho'(Z_{ST}, r, t)) r \, dr. \] 
(4.19)
For a tapered tube defined in spherical orthogonal coordinates, a cross-sectional slice takes the shape of a spherical cap whose area is \( 2\pi r^2 (1 - \cos \Theta) \). The instantaneous mass flux across this slice is calculated as
\[ \dot{m} = \int_\Theta^\Phi 2\pi r^2 u(R_{ST}, \theta, t) (\rho_0 + \rho'(R_{ST}, \theta, t)) \sin \theta \, d\theta, \] 
(4.20)
and the representative acoustic velocity is
\[ U_{Rep}(t) = \frac{1}{\rho_0 (1 - \cos \Theta)} \int_\Theta^\Phi u(R_{ST}, \theta, t) (\rho_0 + \rho'(R_{ST}, \theta, t)) \sin \theta \, d\theta. \] 
(4.21)
The magnitude of this reference acoustic velocity is its maximum absolute value over one period of oscillation, that period being the last one used for the time-averaged streaming calculation in Equation 4.13. The square of this velocity magnitude divided by the speed of sound serves as the normalizing factor for the axial streaming velocity.
\[ U_N = \frac{|U_{Rep}|^2}{c_0} \] 
(4.22)
Normalization by this factor allows a direct comparison of streaming velocity between models with varying cross-sectional areas, and accounts for variations in acoustic velocity amplitude due to geometry, source strength, and spatial discretization.
4.5 Summary

This chapter showed methods of generating standing waves in a straight tube using two line sources spaced 1.5 wavelengths apart and in straight and tapered tubes using a single line source and a reflective tube end. The second method shortens the axial dimension and reduces the computational cost of the simulation. The parameters of the tapered tube geometry were presented as well as the method of calculating the distance between the line source and the rigid end. These dimensions were tabulated for each tube model to be simulated. The chapter concluded with a description of the streaming velocity computation and a method of normalization between tubes of different taper angles at a specific axial location. Chapter 5 shows the results of the simulations of acoustic streaming in straight and tapered tubes, and compares the streaming velocities between tubes of different taper angles.
Chapter 5

Results

This chapter presents results of the simulations of nonlinear acoustic streaming in straight and tapered tubes. The first section compares the numerical simulation of streaming velocity in a straight, two-wavelength-long tube to the theoretical prediction of Schuster and Matz [17]. In Section 5.2 the transverse profile of axial streaming velocity near an adiabatic wall is compared to the streaming velocity profile near an isothermal wall. The components that make up the Lagrangian velocity are shown to provide some insight into the differences in streaming near adiabatic and isothermal walls. The next section reveals the effects of including high-order terms in the calculation of streaming velocity. Section 5.4 presents the streaming circulation patterns in short tubes of varying taper angles. The transverse profiles of streaming velocity at a common axial position compare the various taper angles for both adiabatic and isothermal walls. Finally, the positive portion of mass flux density across the tube is shown for the different tapered tubes and compared to Olson and Swift’s prediction of the optimal taper angle for streaming suppression.

5.1 Model Verification

To ensure that the simulations of acoustic streaming compare favorably with theoretical predictions, a preliminary model of a cylindrical tube is created following the method of Boluriaan and Morris [30]. The tube length spans two wavelengths at 100 Hz in helium, which has a speed of sound of 832.12 m/s at a temperature of 200 K and a pressure of 3.1 MPa. The radius $R$ of the tube is 8.7 mm which
corresponds to the radius of the cylindrical pulse tube in Olson and Swift’s experiment [1]. A standing wave with a pressure amplitude of 0.195 MPa and a velocity amplitude of 31.5 m/s is excited in the middle of the tube by two line sources positioned 1.5 wavelengths apart and one quarter wavelength from each end. The ends are rendered non-reflective through the use of the MOC boundary condition described in Section 3.3 and in Appendix B. At the tube wall where \( r = R \), the boundary condition for a rigid adiabatic wall is defined as follows:

\[
\begin{align*}
    u &= 0 \\
    v &= 0 \\
    \frac{\partial p'}{\partial r} &= 0 \\
    \frac{\partial T'}{\partial r} &= 0
\end{align*}
\]  

Near the tube axis, \( r = 0 \), the axisymmetric boundary condition is defined as

\[
\begin{align*}
    \frac{\partial u}{\partial r} &= 0 \\
    v &= 0 \\
    \frac{\partial p'}{\partial r} &= 0 \\
    \frac{\partial T'}{\partial r} &= 0
\end{align*}
\]  

where the Dirichlet and Neumann conditions are implemented for an offset boundary as described in Section 3.3.1. At all boundaries acoustic density is determined from pressure and temperature by way of the equation of state, shown in Equation 2.5.

The numerical grid for this model is uniform along the axial dimension with 200 grid points, corresponding to 100 grid points per wavelength, and is clustered near the wall in the radial direction with a total of 60 grid points. The minimum radial grid spacing at the wall is \( \Delta r_{\text{min}} = 9.1 \mu m \), and near the axis it is \( \Delta r_{\text{max}} = 1.35 \text{mm} \). This radial clustering allows for six grid points within one viscous boundary layer thickness of the tube wall (\( \delta_\nu = 80.3 \mu m \)) and seven grid points within one thermal boundary layer thickness (\( \delta_\kappa = 96.7 \mu m \)). After testing the model at this resolution a Courant number of 0.4 is chosen, and the time step
resulting from $\Delta r_{\text{min}}$ and Equation 3.29 is 4.37 ns. The model equations include terms up to $O(\varepsilon^2)$ and $O(\eta \varepsilon)$. This accounts for nonlinearity and dissipation, but neglects temperature dependence of viscosity and thermal conductivity. An analysis of the grid refinement for this model is presented in Appendix C.

The model is allowed to run for 550 periods of oscillation, and streaming is calculated based on the time-averaged velocity over the last 10 periods. At the acoustic velocity amplitude listed above and with a viscous boundary layer thickness of $8.0346 \times 10^{-5}$ m, the nonlinear Reynolds number is computed by Equation 1.1 to be $Re_{nl} = 8.4$. This measure indicates that streaming developed in this model is in the fast, or nonlinear, range. The streaming circulation is shown by the trace lines in Figure 5.1. The line sources are represented by solid vertical lines, and the dashed line indicates the position of a cross-sectional profile of axial streaming velocity, which is shown below in Figure 5.2. The depiction of the tube wall is omitted from the top of Figure 5.1 so that the counter-rotating inner-streaming cells may be more easily discerned between the wall at $r = 0.0087$ m and the outer-streaming cells that occupy most of the tube radius.

![Figure 5.1](image)

**Figure 5.1.** Streaming circulation in a two-wavelength-long tube with rigid adiabatic wall, averaged over 10 periods from cycles 540 to 550.

The transverse profile of axial streaming velocity is shown in Figure 5.2. Three numerical solutions averaged over different time spans reveal the development and
convergence of streaming velocity. The latter two, from periods 440 to 450 and from periods 540 to 550, are nearly the same, which indicates that the numerical solution is well developed after 550 cycles. These solutions are compared to the theoretical prediction of Schuster and Matz for streaming velocity resulting from a standing wave of similar amplitude in a cylindrical tube [17].

![Image of streaming velocity profile](image)

**Figure 5.2.** Profile of axial streaming velocity comparing the development of the numerical simulation to the theoretical prediction from Schuster and Matz.

The streaming velocity in the axial direction predicted by Schuster and Matz is given by Equation 5.3 [30].

\[
  u_{st} = -\frac{3|U_1|^2}{8c} \sin(2kx) \left[ 1 - 2 \left( \frac{r}{R} \right)^2 \right] 
\]  

(5.3)

This theoretical prediction does not take into account the temperature dependence of viscosity or thermal conductivity, nor does it address the behavior of the fluid near a solid wall. The numerical and theoretical solutions show good agreement outside of the boundary-layer region near the wall, but diverge near the tube axis. The results in Appendix C show that increasing the number of grid points in the radial dimension near the tube axis brings the numerical solution slightly closer to the theoretical prediction, but it does little to decrease the discrepancy shown here.
5.2 Adiabatic vs. Isothermal Boundary Condition at the Tube Wall

The next comparison shows the difference in axial streaming velocity between adiabatic and isothermal temperature boundary conditions at the tube wall. In experimental measurements and observations of Rayleigh streaming, attempts are often made to control temperature gradients at the tube wall by way of a water bath or some other means of thermal isolation. Done carefully, this would result in a nearly isothermal boundary with the wall held at a constant temperature. In a numerical simulation an isothermal wall is easily accomplished by imposing a constant temperature \( T_0 \) at that boundary. This is equivalent to setting the fluctuating temperature at the wall to zero such that

\[
T'(R) = 0 \quad (5.4)
\]

A relatively simple alternative in a numerical model is to implement an adiabatic temperature boundary condition where the temperature of the wall is allowed to fluctuate with the temperature of the fluid so that the temperature gradient at the wall is zero. This is expressed as

\[
\frac{\partial T'}{\partial r} \bigg|_{r=R} = 0 \quad (5.5)
\]

Due to the thermal properties of most materials capable of containing a standing wave in a pressurized vessel, an adiabatic boundary is very difficult to realize physically. Conversely a physically realistic thermal boundary is difficult to model numerically without coupling a thermodynamic model of the tube wall to the acoustic streaming model. Here the effects of two simple boundary conditions are compared, and it is assumed that the effect due to a physically realistic thermal boundary lies somewhere in between.

Figure 5.3 shows the transverse profile of axial streaming velocity for the two thermal wall boundary conditions across the entire radius of the tube (upper graph) and near the wall (lower graph). Of note is the change in velocity magnitude between the two cases when all other model parameters are kept the same. In
the isothermal case not only is the outer streaming velocity stronger, but also the streaming velocity nearest the wall does not show the change in direction associated with an inner-streaming vortex.

To investigate these effects further, the Eulerian streaming velocity \( \langle u_2 \rangle \) and the velocity transform \( u_T \), described above in Section 4.3, are shown in Figure 5.4 for the same transverse profile. Here, too, the magnitudes of the mean velocities are greater in the case of an isothermal boundary at the wall. Also, it can be seen that the non-standing-wave phasing near the wall increases the velocity transform component in that region, which contributes to the formation of inner streaming near the adiabatic wall. The Eulerian mean velocity does not feature a reversal in direction that would indicate an inner-streaming vortex near the wall.

The region nearest the tube wall is more easily seen in Figure 5.5. Here the com-

---

**Figure 5.3.** Transverse profile of axial Lagrangian streaming velocity. The viscous boundary layer thickness is indicated by \( \delta_v \). The thermal boundary layer thickness (not shown) is 21% larger.
ponents of the mean Lagrangian velocity are shown within a few boundary-layer thicknesses of the tube wall. Near an adiabatic wall the sum of these components results in the small region of positive velocity shown in Figure 5.3 indicating an inner-streaming vortex. In the case of an isothermal wall, however, it is not the absence of the velocity transform component near the wall that discourages the formation of inner streaming, but the balance between the two velocity components that cancels flow reversal near the wall. While this effect may not be consistent in all cases of Rayleigh streaming near an isothermal rigid boundary, its analysis provides some insight into the current model of helium oscillating in a cylindrical tube.

**Figure 5.4.** Eulerian and velocity transport components of axial streaming velocity.
5.3 Effects of High-Order Terms on Streaming Development

Thus far the numerical model used for comparison with the theory of Schuster and Matz and in the comparison between adiabatic and isothermal tube walls has included only the lowest order terms necessary to produce circulation from nonlinear acoustic streaming. These may be referred to as “nonlinear dissipative” terms of order $\varepsilon^2$ and $\eta\varepsilon$ where the temperature-dependent portions of the coefficients of viscosity and thermal conductivity are neglected, as is the term in the temperature equation involving components of the rate-of-shear tensor, $\frac{1}{2}\mu_0\phi_{ij}\phi_{ij}$. In this section, those higher order terms are included in stages to explore the effect on the development of Rayleigh streaming.
The first stage is the nonlinear dissipative case described above. The streaming velocity resulting from this version of the model equations is denoted ND in Figure 5.6, showing the region near an adiabatic wall, and Figure 5.7, showing the same region near an isothermal wall. The second stage results from a version of the model equations that includes temperature-dependent viscosity and thermal conductivity. It is denoted NDT in Figures 5.6 and 5.7. The terms involving these coefficients, $\mu(T')$ and $\kappa(T')$, are of $O(\eta \varepsilon^2)$, but terms involving components of the rate-of-shear tensor, also $O(\eta \varepsilon^2)$, are omitted at this stage. The following stage, marked ND2, includes all terms up to $O(\varepsilon^2)$ and $O(\eta \varepsilon^2)$ including $\mu(T')$ and $\kappa(T')$ as well as rate-of-shear components. The final stage includes temperature dependence in the viscosity coefficient associated with the rate-of-shear-tensor components, $\frac{1}{2} \mu(T') \phi_{ij} \phi_{ij}$, which makes that term $O(\eta \varepsilon^3)$. The streaming velocity profile resulting from this last version is marked ND2T.

In Figure 5.6 the profiles of streaming velocity near an adiabatic wall resulting from the ND and NDT versions of the model equations show the effect of adding temperature dependence to viscosity and thermal conductivity. Adjacent to the wall, the axial velocity decreases, and the point where it changes direction moves toward the wall. This indicates a decrease in the radial dimension of an inner-streaming vortex. Well outside of the boundary-layer region the axial streaming velocity increases slightly. With the addition of the components of the rate-of-shear tensor in version ND2, the increase in axial velocity far from the wall is more pronounced. The addition of temperature-dependent viscosity to the rate-of-shear-tensor term produces no discernible difference between the profiles marked ND2 and ND2T. With only 50 periods elapsed in this simulation, the acoustic streaming circulation is still in development, and the differences between versions of the model equations can be seen most clearly in the upper 10% of the tube radius. With longer simulation time, more differences may become apparent throughout the transverse profile of the tube.

Near an isothermal wall, Figure 5.7 shows that the greatest effect in the developing axial velocity profile occurs with the addition of temperature dependence to viscosity and thermal conductivity. Version ND, which neglects this dependence, stands alone in this figure. The rest of the versions of the model equations result in very similar axial streaming velocity profiles. It is also worthwhile to note the
Figure 5.6. Axial Lagrangian streaming profiles of varying order near an adiabatic wall. ND: terms up to $O(\varepsilon^2)$ and $O(\eta\varepsilon)$ with $\mu = \mu_0$ and $\kappa = \kappa_0$. NDT: $O(\varepsilon^2)$ and $O(\eta\varepsilon)$ terms with $\mu = \mu(T')$ and $\kappa = \kappa(T')$. ND2: $O(\varepsilon^2)$ and $O(\eta\varepsilon^2)$ terms. ND2T: $O(\varepsilon^2)$ and $O(\eta\varepsilon^2)$ terms plus $\frac{1}{2}\mu(T')\phi_{ij}\phi_{ij}$. The viscous boundary layer thickness is indicated by $\delta_\nu$. The thermal boundary layer thickness (not shown) is 21% larger.

absence of reversal in Lagrangian velocity in the boundary-layer region adjacent to the isothermal wall. Even with the addition of high-order terms in the model equations, there is no indication of the development of an inner-streaming vortex in this simulation.

5.4 Comparison of Straight and Tapered Tubes

The next set of models explores the effect of varying taper angle on the development of acoustic streaming. The geometric parameters for these models are listed in Table 4.1. In each model Rayleigh streaming is simulated using the version of
Figure 5.7. Axial Lagrangian streaming profiles of varying order near an isothermal wall. ND: terms up to $O(\varepsilon^2)$ and $O(\eta \varepsilon)$ with $\mu = \mu_0$ and $\kappa = \kappa_0$. NDT: $O(\varepsilon^2)$ and $O(\eta \varepsilon)$ terms with $\mu = \mu(T')$ and $\kappa = \kappa(T')$. ND2: $O(\varepsilon^2)$ and $O(\eta \varepsilon^2)$ terms. ND2T: $O(\varepsilon^2)$ and $O(\eta \varepsilon^2)$ terms plus $\frac{1}{2} \mu(T') \phi_{ij} \phi_{ij}$. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.

The model equations that includes terms accurate to $O(\varepsilon^2)$ and $O(\eta \varepsilon^2)$, referred to as ND2 in the previous section. The standing wave that generates the streaming circulation is excited by a single line source in combination with one rigid end and one non-reflecting end as described in Section 4.1.3. The amplitude of the source is adjusted so that the acoustic pressure amplitude at the rigid end is approximately 0.195 MPa. For each of the models the nonlinear Reynolds number lies in the range between 60 and 140 as calculated using Equation 1.1. The variation in those numbers is due to the variation in velocity amplitude of the standing waves in the different tapered tubes.

Figures 5.8 to 5.12 show the circulation patterns due to streaming velocity
averaged over 10 periods of oscillation. The taper angle in the title of each figure refers to the model taper angle which is one half of the total included angle of the tapered tube. The solid line in each figure indicates the position of the line source, and the dashed line shows the position of the transverse profile where the radius of each tube is of a consistent length equal to 0.0375 m. The boundaries drawn around the stream traces show the extent of the computational domain. These boundaries coincide with the tube wall at the top of the diagram and the tube ends at either side. The boundary near the axis is offset from \( r = 0 \) due to the grid arrangement presented in Chapter 3. In these figures the temperature boundary condition at the tube wall and the rigid end is adiabatic. Simulations involving isothermal rigid surfaces are discussed below, but the circulation patterns are very similar to those shown in Figures 5.8 to 5.12 for adiabatic surfaces and are not presented here.

**Figure 5.8.** Streaming circulation in a cylindrical tube with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.
Figure 5.9. Streaming circulation in a 0.4° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 30 to 40.

Figure 5.10. Streaming circulation in a 0.8° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 30 to 40.
Figure 5.11. Streaming circulation in a 1.2° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.

Figure 5.12. Streaming circulation in a 1.6° tapered tube (total included angle) with rigid adiabatic wall and left end, and non-reflecting right end, averaged over 10 periods from cycles 20 to 30.
These plots of streaming circulation cells show that as the taper angle increases, the streaming flow takes on a spiral shape that tends outward from the counterclockwise vortex in the region between the source and the rigid left end. The vortex near the source also spirals, but in an inward, clockwise direction. This apparent flow from one area in the domain to another suggests that a transfer of mass is occurring. It also suggests that the simulation has not yet reached an equilibrium state, and after an elapsed time of only 30 periods of oscillation, that may well be the case. Considering that Figure 5.1 shows a streaming solution converging after 550 periods, the current set of results represent streaming flow that has only just begun to circulate. These figures can be considered a preliminary indication of the effect of taper angle on streaming velocity even though that velocity is quite small in such an early stage of development.

**Figure 5.13.** Axial Lagrangian streaming profile near adiabatic wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.
Axial streaming velocity within boundary layer of adiabatic wall

Figure 5.14. Profile of axial Lagrangian streaming velocity adjacent to an adiabatic wall. The viscous boundary layer thickness is indicated by $\delta _{\nu}$. The thermal boundary layer thickness (not shown) is 21% larger.

Figure 5.13 shows the transverse profile of normalized axial streaming velocity in tubes of various taper angles with adiabatic walls. As the taper angle increases, the velocity near the tube wall decreases, and the radial position of the minimum velocity shifts toward the wall. Looking at a region closer to the tube wall in Figure 5.14 reveals that the zero crossing from negative to positive velocity also approaches the wall until the positive streaming velocity adjacent to the wall disappears. This zero crossing signifies a flow reversal near the tube wall that is indicative of an inner-streaming vortex. Its shift toward the wall shows that the inner-streaming region becomes narrower with increasing taper angle and eventually disappears altogether for angles of 0.6° and higher.

For the case of an isothermal temperature boundary condition at the tube wall, axial streaming velocity also decreases with increasing taper angle in the
region nearest the wall, as shown in Figure 5.15. The radial position of minimum velocity shifts toward the wall as well, but since no flow reversal is apparent near the isothermal wall, the increasing taper angle has no effect on the absent inner-streaming vortex. This absence was noted above in Sections 5.2 and 5.3.

![Figure 5.15. Profile of axial Lagrangian streaming velocity adjacent to an isothermal wall. The viscous boundary layer thickness is indicated by $\delta_v$. The thermal boundary layer thickness (not shown) is 21% larger.](image)

5.5 Change in Average Streaming Velocity with Taper Angle

The overall strength of streaming circulation can be compared between tubes of various taper angles by considering the spatial average of axial streaming velocity in the positive direction for each tube. Figure 5.16 illustrates the positive portion
of the axial streaming velocity used for the spatial average. This velocity, $u_{st}^+$, is averaged over the area of the tube cross section only where streaming flow is in the positive axial direction. The area across which the streaming flow is negative is not considered in the calculation. This mean axial streaming velocity is taken to represent the strength of the circulation due to Rayleigh streaming.

![Graph](image)

**Figure 5.16.** Positive axial streaming velocity contribution (solid line) to the spatially averaged velocity $\langle u_{st}^+ \rangle$. Only the upper 10% of the radial profile is shown.

Figure 5.17 compares the normalized mean velocities between tubes of various taper angles for each temperature boundary condition at the wall. For the adiabatic case the strength of streaming circulation remains constant for taper angles up to $0.4^\circ$ and increases slightly with taper for the $0.6^\circ$ and $0.8^\circ$ angles. With an isothermal wall the strength of streaming circulation is greater overall and grows with increasing taper angle over the entire range. It must be noted that these simulations ran only to 30 periods of vibration which is very short compared to the 550 periods used for model verification in Section 5.1. Streaming circulation has only just begun in these models. More computation time is needed to show trends in streaming circulation strength with taper angle for fully developed flow.
Figure 5.17. Comparison of mean positive streaming velocity in the axial direction as an indicator of relative circulation strength between variously tapered tubes.

5.6 Summary

This chapter presented results of acoustic streaming simulations in tubes of varying taper angle. Streaming velocity in a straight tube was shown to converge to a steady solution and to compare favorably with an early theory of Rayleigh streaming in a tube by Schuster and Matz. The difference between isothermal and adiabatic temperature boundary conditions and their effects on streaming were investigated through the dissection of the Lagrangian streaming velocity into its components of Eulerian velocity and velocity transform. The latter was shown to come into play near the rigid tube wall where the phase between density and velocity deviated from standing-wave phasing. This effect was related to the development of inner streaming near an adiabatic wall and its absence near an isothermal wall.
A study of high-order terms in the simulation of Rayleigh streaming showed that those terms of order $\varepsilon^2$ and $\eta\varepsilon^2$, which include temperature dependence of viscosity and thermal conductivity, contribute to the formation of streaming velocity. Terms of order $\eta\varepsilon^3$ were shown to make no discernible contribution.

The effect of taper angle was considered next, and the change in streaming circulation pattern was shown in a series of trace-line plots. The influence of taper angle was also shown in plots of the transverse profile of streaming velocity. These revealed that for both adiabatic and isothermal walls, the streaming velocity near the wall decreased, and the radial position of minimum velocity shifted toward the wall. For the adiabatic wall, the small region of positive velocity adjacent to the wall decreased in width with increasing taper angle which indicated a narrowing of the inner streaming vortex until it ultimately disappeared. Finally a comparison of the spatial average of the axial streaming velocity in the positive axial direction showed an increase in the strength of circulation with taper angle for both adiabatic- and isothermal-walled tubes. It was noted that more computation time would be needed to verify this trend with fully developed streaming circulation.
Chapter 6

Conclusions

This chapter offers a review of the previous chapters in this study with a summary of the main components of the research into the effects of taper angle on Rayleigh streaming in a tube. It highlights contributions made during the course of this project to the technique of numerical modeling and to the knowledge of acoustic streaming. Factors limiting the performance of the current numerical model are explained, and suggestions for possible improvements follow in the next section. The chapter concludes with suggestions to expand these ideas into future research.

6.1 Review

The story of boundary-layer-driven streaming in tapered tubes began with Lord Rayleigh and the observations that led him to develop a theory of the circulation generated by standing waves between parallel plates [16]. Chapter 1 traced the history of this theory through Schuster and Matz’s solution for a cylindrical tube [17], Rott’s addition of thermal effects [21], Lighthill’s emphasis on the effects of fluid inertia [10], and Olson and Swift’s adaptation for varying taper angle and arbitrary phasing [1]. This latter contribution combined theory with an experiment that suggested it was possible to suppress streaming mass flux in the pulse tube of a pulse-tube refrigerator by tapering the tube slightly. The optimal taper angle to achieve this suppression was calculated using Olson and Swift’s theory. Two research groups made numerical simulations of pulse-tube refrigerators to investigate this claim. The first group of Baek, et al. modeled the steady solution of time-averaged mass flux in a tapered tube and found that it decreased with increasing
taper angle [3]. The second group of He, et al. modeled a time-marching solution of streaming mass flow in a tapered tube connected at either end to straight tubes representing heat exchangers. They concluded that a non-uniform pattern of mass flow, which occurred at a certain taper angle, improved the performance of the refrigerator by disrupting the temperature gradient along the tube. Neither of these groups examined streaming generated by pure standing-wave phasing.

Chapter 1 concluded with the goals of the current research, which are to develop a time-accurate numerical model of nonlinear acoustic streaming driven by boundary-layer effects in straight and tapered tubes, to compare the effect of high-order terms of the governing equations on streaming, and to investigate the effect of variously tapered tubes on streaming velocity. To this end, Chapter 2 began the description of the numerical model with the governing equations. These were developed using perturbation methods to reduce round-off error and to allow investigation of the high-order terms. The development was carried out both in cylindrical coordinates for straight tubes and in spherical coordinates for tapered tubes.

Chapter 3 described the numerical method used to perform a time-accurate solution to the model equations. It gave the details of the algebraic grid refinement technique that clustered grid points near rigid surfaces in the model in order to resolve boundary-layer regions. Conformal mapping transformed the model equations from a physical to a computational domain, which made the solution space valid for implementation in a finite-difference scheme over a refined grid. The MacCormack finite-difference method was chosen to be second order in time and space, and was described with special emphasis on the management of second derivatives and source terms. A selection of boundary treatments used in the model was described, including the method of characteristics for a non-reflecting boundary and a method to compute a one-sided axisymmetric boundary at a position offset from the tube axis. The chapter concluded with a description of domain decomposition to distribute the computational task among multiple processors.

Chapter 4 continued the description of the numerical model with details of its use in simulating streaming. A two-line source was used to generate a standing wave in a straight tube with two non-reflecting ends by following the method developed by Boluriaan and Morris [30]. This allowed the rapid establishment of
a standing wave at high amplitude without wave steepening. This method was adapted for shorter tubes by using a single line source in coordination with one rigid end and one non-reflecting end. Quarter-wavelength standing waves were demonstrated in both straight and tapered tubes. The calculation for determining the tapered tube dimensions was presented, as were the methods of calculating Lagrangian streaming velocity and the normalization factor. The latter was used to compare streaming velocities between variously tapered tubes.

Chapter 5 presented the results of the numerical simulations. The first three of these, as listed below, were obtained using a model of a straight tube with a length of two wavelengths and a radius of 8.7 mm, which was equal to the radius of the straight pulse tube used by Olson and Swift in their experiment [1]. The remaining results used the dimensions for straight and tapered tubes listed in Table 4.1. The results of this study are as follows:

1. A numerical simulation of acoustic streaming velocity was shown to converge and to compare favorably with the theoretical prediction of Schuster and Matz [17].

2. The current model showed that streaming velocity near an isothermal wall has less tendency to form an inner-streaming cell than in proximity to an adiabatic wall. An analysis of the components of the Lagrangian streaming velocity revealed the influence of non-standing-wave phasing on the formation of inner streaming.

3. Comparison of the effects of high-order terms in the model equations showed the importance of temperature dependence of viscosity and thermal conductivity, as well as components of the rate-of-shear tensor on Rayleigh streaming.

4. Examination of trace-line patterns of streaming in straight and tapered tubes showed a spiral tendency in the circulation with increasing taper angle.

5. Comparing transverse profiles of axial streaming velocity in straight and tapered tubes showed a decreasing trend in the region near the tube wall and a shift toward the wall of the position of minimum velocity with increasing taper angle for both adiabatic and isothermal temperature boundary
conditions. Near an adiabatic wall the width of the inner-streaming vortex narrowed with increasing taper angle until that region vanished completely.

6. A graph of the spatially averaged positive axial streaming velocity for each tapered tube suggested an increase in Rayleigh streaming circulation with increasing taper angle.

This last result contrasts with the findings of Baek, et al. who reported that time-averaged mass flux decreased with increasing taper angle [3]. One reason for this difference lies in the choice of normalization factor used in the current study. The streaming velocity profiles shown at particular axial positions along the tube are normalized in this study by an acoustic velocity derived from the mass flux at that same axial position. Baek, et al. calculated non-dimensional second-order time-averaged mass flux, which they showed to be independent of axial position, rather than streaming velocity and thus did not use an axially dependent normalization factor. More importantly their model converged upon a fully developed second-order flow whereas the current time-accurate model attained only the beginning of streaming development. More computation time and analysis would be necessary to compare the results from the two methods fully.

6.2 Contributions

Several aspects of this research may be considered contributions to the state of knowledge not only of acoustic streaming in tapered tubes, but also of techniques of numerical simulation. Those numerical techniques developed for this study that may be novel or potentially helpful are listed first.

1. The Method-of-Characteristics boundary condition described by Thompson [62] has been re-derived for the perturbed primitive variables of acoustic density, velocity, and temperature. This is shown in Appendix B.

2. A one-sided axisymmetric boundary condition for points offset from the axis has been developed in Section 3.3.1 for both the Neumann and Dirichlet conditions.
3. A variation on the two-line-source method of Boluriaan and Morris [30] has been implemented using a single line source between a rigid boundary and a non-reflecting boundary to simulate a reflected standing wave.

Contributions to the knowledge of acoustic streaming and its study through numerical modeling are listed as follows:

1. A numerical model of acoustic streaming in tapered tubes has been demonstrated using a time-marching finite-difference technique.

2. The importance of temperature dependence of viscosity and thermal conductivity as well as that of the rate-of-shear-tensor components to the accurate calculation of Rayleigh streaming has been demonstrated through direct numeric simulation.

3. A difference in the development of Rayleigh streaming has been shown numerically between isothermal and adiabatic wall temperature boundary conditions.

4. A difference in streaming circulation patterns has been shown between tubes of various taper angles.

5. Preliminary numerical results have been obtained that suggest that circulation due to Rayleigh streaming increases with increasing taper angle.

### 6.3 Limitations of the Current Model

It may be noted in examining the results of this study that the duration of the simulations of acoustic streaming in tapered tubes spans only 30 cycles of oscillation, whereas the simulation that verifies the model against streaming theory converges after 550 cycles. This difference can be attributed mainly to the restrictions imposed by the tapered tube in spherical coordinates on the minimum grid spacing and on the time step. As the taper angle increases, the radius at the wide end also increases, but the boundary layer thickness at the tube wall remains the same. The boundary layer needs the same number of radial grid points to resolve it at the wide end of a tapered tube as it does throughout a straight tube. As the
radius of the tapered tube converges toward the narrow end, so does the minimum radial grid spacing. This occurs as a result of the spherical coordinate system. The converged grid spacing is not required by the boundary layer thickness at the narrow end of the tube. Therefore, the minimum radial grid spacing at the narrow end is smaller than needed. The time step, however, must be defined by this minimum grid spacing to ensure numerical stability, and thus it relates inversely to the taper angle. An increasing taper angle results in a decreasing time step and an increased number of computations required to model a single acoustic cycle.

A possible solution to this problem is to implement a multiblock grid for the tapered tube that has fewer radial grid points at the narrow end than at the wide end. This would be done in multiple steps along the length of the tube with an interpolation scheme to handle the junction between each stage. This type of grid arrangement would allow greater taper angles without the unnecessary computational burden of the smaller and more numerous time steps.

6.4 Future Work

Additional improvements to the numerical model include using a higher order finite difference calculation. A MacCormack two-step scheme that is second order in time and fourth order in space could be more computationally efficient than the current method that is second order in both time and space. Considering the range of $\Delta t/\Delta x$ ratios in the refined grid, spatial accuracy would dominate over most of the domain since the time step is determined by a small area at the narrow end of the tapered tube and near the rigid walls in the boundary layer [60].

Acceleration due to gravity has been retained in the development of the model equations, but it was not used in any of the simulations. Since the tapered pulse tube in Olson and Swift’s refrigerator was oriented vertically, there is a possibility that gravity could play a role in the boundary-layer interactions that drive streaming in a tapered tube.

It was noted earlier that neither adibatic nor isothermal wall boundary conditions completely represent a thermally realistic tube wall. This presents another opportunity for improvement of the model. Coupling a thermodynamic model of the tube wall to the streaming model inside the tube would allow a more accu-
rate simulation and more possibilities for the investigation of temperature effects. Finally, choosing a different source method and boundary condition at the tube ends would allow phasing between pressure and velocity to match that observed in the empirical measurements and would permit a direct comparison between experiments and numerical analyses of streaming in straight and tapered tubes.
Appendix

A

Small Ordering Parameters

This section defines the small ordering parameters $\varepsilon$ and $\eta$ through the process of nondimensionalizing the model equations, and compares their relative magnitudes. It is preceded by a brief introduction to perturbation theory. The end of the article compares the formal perturbation method to a similar method that expands quantities into ambient and fluctuating parts. This second method is referred to as the “acoustic disturbance method.”

A.1 Perturbation Theory

The model presented in Equations 2.21 to 2.24 consists of the independent variables of time and space, the functions that depend on those variables, i.e. the model variables of pressure, density, velocity, temperature, viscosity, thermal conductivity, and their products, and the partial differentials that operate on those functions. The model equations derived either through finite control volume or infinitesimal fluid element approach are based on the assumption that time and space are continuous [51]. In a numerical implementation, that assumption is no longer correct. A time-stepping finite-difference scheme discretizes time through the calculation of functions and operators over a discrete spatial domain. The validity of the functions and operators over this domain depends on their treatment through perturbation methods. The discrete finite difference operators are discussed in Chapter 3. Here the validity of the model functions over a discrete domain is addressed.

Perturbation expansion approximates a function by replacing it with a finite
sequence of numbers. The difference between the true function $f(x)$ and its approximation $\bar{f}(x)$ is called the discretization error,

$$E_N(x) = f(x) - \bar{f}_N(x),$$

(A.1)

where $N$ is the number of elements in the sequence [67]. If the approximate function of $x$ is defined as a sequence with respect to a small parameter $\epsilon$ as

$$\bar{f}_N(x; \epsilon) = \sum_{n=0}^{N-1} a_n(x)g_n(\epsilon),$$

(A.2)

then the true function may be written

$$f(x; \epsilon) = \sum_{n=0}^{N-1} a_n(x)g_n(\epsilon) + E_N(x; \epsilon)$$

(A.3)

with the discretization error represented as the $N^{th}$ element of the sequence. The accuracy of this approximation depends on the behavior of the discretization error as the parameter $\epsilon$ approaches zero. The function is accurately represented by the sequence

$$f(x; \epsilon) \sim \sum_{n=0}^{\infty} a_n(x)g_n(\epsilon) \quad \text{as} \quad \epsilon \to 0$$

(A.4)

if $E_N(x; \epsilon)$ from Equation A.3 behaves according to

$$E_N(x; \epsilon) = O[g_N(\epsilon)].$$

(A.5)

The Landau symbol $O$ describes the behavior of the function $E(x; \epsilon)$ as compared with a gauge function $g(\epsilon)$. The relation

$$E(x; \epsilon) = O[g(\epsilon)] \quad \text{as} \quad \epsilon \to 0$$

(A.6)

holds true if a positive function $M(x)$ independent of $\epsilon$ exists along with an $\epsilon_0 > 0$ such that

$$|E(x; \epsilon)| \leq M(x)|g(\epsilon)| \quad \text{for all} \quad \epsilon \leq \epsilon_0.$$
This can also be written as follows [68, 69].

$$\lim_{\epsilon \to 0} \left| \frac{E(x; \epsilon)}{g(\epsilon)} \right| < \infty \quad (A.8)$$

When the gauge function is of the form $g_N(\epsilon) = \epsilon^N$, then the approximation is said to have an order of accuracy $N$ [67].

### A.2 Nondimensionalized Equations of Motion

The procedure to nondimensionalize the equations of motion closely follows the work of Cotaras [70]. First the model equations are presented with the assumption that the medium is a perfect monatomic gas. The fluid dynamic variables are normalized by their characteristic values, each of which is determined in turn by setting the coefficients equal to unity where possible. Those that are different from unity are used to show the relative sizes of their associated terms. Here the model equations are presented as in Chapter 2, omitting the body force due to gravity.

**Continuity Equation**

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0 \quad (A.9)$$

**Momentum Equation**

$$\rho \frac{D\mathbf{v}}{Dt} = -\nabla p - \nabla \times [\mu (\nabla \times \mathbf{v})] + \frac{4}{3} \nabla (\mu \nabla \cdot \mathbf{v}) \quad (A.10)$$

**Energy Equation**

$$\rho c_v \frac{D\mathbf{T}}{Dt} = -p \nabla \cdot \mathbf{v} + \frac{\mu}{2} \phi_{ij} \phi_{ij} + \nabla \cdot (\kappa \nabla T) \quad (A.11)$$

**Equation of State**

$$p = \rho RT \quad (A.12)$$

### A.2.1 Characteristic time and space

Since the model solution to linear order is a standing wave, and the radius of the tube is much greater than the boundary layer thickness, it’s reasonable to relate...
the characteristic time and space by way of the wave equation, setting time and
space derivatives equal.

\[ \nabla^2 \Phi - \frac{1}{c_0^2} \frac{\partial^2 \Phi}{\partial t'^2} = 0 \]  

(A.13)

Here \( \Phi \) is the velocity potential and could be replaced by any of the acoustic
variables that satisfy the linear wave equation.

The time variable is nondimensionalized by choosing an appropriate character-
stic quantity such that \( \hat{t} = t/t_{ch} \), where the nondimensional variable is signified
by the circumflex on top, i.e. \( \hat{t} \). Choose a quantity for time that is characteristic
of the signal. For a periodic wave, \( t_{ch} \equiv 1/\omega \) is appropriate. The spatial variable
is nondimensionalized similarly by \( \hat{x} = x/x_{ch} \), and the characteristic distance is
defined by the relation of space to time in the nondimensional wave equation.

Using the following relations between the dimensional and nondimensional vari-
ables and their characteristic values yields the nondimensional wave equation be-
low.

\[ t = t_{ch} \hat{t} \]
\[ x = x_{ch} \hat{x} \]
\[ \Phi = \Phi_{ch} \hat{\Phi} \]
\[ \nabla^2 = \frac{1}{x_{ch}^2} \hat{\nabla}^2 \]

\[ \frac{\Phi_{ch}}{x_{ch}^2} \hat{\nabla}^2 \hat{\Phi} - \frac{\Phi_{ch}}{c_0^2 x_{ch}^2} \frac{\partial^2 \hat{\Phi}}{\partial \hat{t}^2} = 0 \]  

(A.14)

Dividing by the leading coefficient and setting the remaining coefficient equal to
unity reveals the required characteristic distance.

\[ x_{ch} \equiv c_0 t_{ch} = \frac{c_0}{\omega} \]  

(A.15)

The characteristic velocity potential is left undefined as it drops out and plays no
further role in the discussion.
A.2.2 Nondimensional Continuity Equation

The continuity equation is nondimensionalized by defining nondimensional particle velocity and density variables. Characteristic quantities are derived that allow the coefficients of the nondimensional equation to be unity. Here the nondimensional equation is written in terms of excess density rather than total density so that the characteristic quantity may be defined as shown below. The excess density, also referred to as condensation, is denoted by $\rho'$. It is found by subtracting the static density from the total density. The static part of the variable is chosen to be the same as the ambient quantity that exists in the quiescent medium.

$$\rho' = \rho - \rho_0$$  \hspace{1cm} (A.16)

The nondimensional excess density is found by dividing excess density by the characteristic density.

$$\hat{\rho}' = \frac{\rho - \rho_0}{\rho_{ch}}$$  \hspace{1cm} (A.17)

Replacing density in the continuity equation by $\rho = \rho_0 + \rho'$ and separating the material derivative gives the following form of the equation.

$$\frac{\partial \hat{\rho}'}{\partial \hat{t}} + \hat{v} \cdot \nabla \hat{\rho}' + \rho_0 \nabla \cdot \hat{v} + \rho' \nabla \cdot \hat{v} = 0$$  \hspace{1cm} (A.18)

The dimensional variables are substituted by the product of their characteristic and nondimensional quantities,

$$\rho' = \rho_{ch} \hat{\rho}'$$
$$\mathbf{v} = v_{ch} \hat{\mathbf{v}}$$
$$t = t_{ch} \hat{t}$$
$$\nabla = \frac{1}{x_{ch}} \hat{\nabla}$$

and the resulting equation is normalized by the leading coefficient. This reveals the relationship of the characteristic quantities in the remaining coefficients.

$$\frac{\partial \hat{\rho}'}{\partial \hat{t}} + \frac{v_{ch} t_{ch}}{x_{ch}} \hat{\mathbf{v}} \cdot \hat{\nabla} \hat{\rho}' + \frac{v_{ch} t_{ch}}{x_{ch}} \rho_0 \hat{\nabla} \cdot \hat{\mathbf{v}} + \frac{v_{ch} t_{ch}}{x_{ch}} \rho' \hat{\nabla} \cdot \hat{\mathbf{v}} = 0$$  \hspace{1cm} (A.19)
The characteristic velocity is determined by making \( v_{ch}t_{ch}/x_{ch} = 1 \). Since \( x_{ch} = c_0t_{ch} \) the characteristic velocity is defined as the small-signal speed of sound.

\[
v_{ch} \equiv c_0
\] (A.20)

The coefficient \( \rho_0/\rho_{ch} \) is also set to unity which shows that the characteristic density is the same as the ambient density.

\[
\rho_{ch} \equiv \rho_0
\] (A.21)

With these defined characteristics the nondimensional continuity equation can be written

\[
\frac{\partial \hat{\rho}'}{\partial \hat{t}} + \hat{\nabla} \cdot \hat{v} + \hat{\rho}'\hat{\nabla} \cdot \hat{v} + \hat{v} \cdot \hat{\nabla} \hat{\rho}' = 0
\] (A.22)

which is further simplified by using the product rule of the divergence and noting that \( \hat{\rho} = 1 + \hat{\rho}' \).

\[
\frac{\partial \hat{\rho}'}{\partial \hat{t}} + \hat{\nabla} \cdot (\hat{\rho}\hat{v}) = 0
\] (A.23)

This analysis brings out an important nondimensional factor with the definition of nondimensional particle velocity, \( \hat{v} = v/c_0 \). The magnitude of this scaled velocity is defined as the acoustic Mach number,

\[
\varepsilon = |\hat{v}| = \frac{U}{c_0}
\] (A.24)

where \( U \) is the magnitude of the dimensional particle velocity. This number is representative of the magnitude of the scaled condensation, \( \rho'/\rho_0 \), as well as the scaled excess pressure and temperature to be shown. The Mach number \( \varepsilon \) is used here as one of the small ordering parameters.

### A.2.3 Nondimensional Momentum Equation

By separating the pressure and viscosity into static and excess quantities the momentum equation is nondimensionalized in a fashion similar to the continuity equation. The static part of pressure is considered to be the same as the ambient pressure which is the uniform pressure experienced by the quiescent medium. The
static viscosity is that part of the viscosity which is unaffected by variations in temperature, the temperature dependence of viscosity being given by Equation 2.6.

Substituting the expanded variables, \( p = p_0 + p' \) and \( \mu = \mu_0 + \mu' \), into Equation A.10 yields the following form after some judicious use of vector identities.

\[
\rho \frac{\partial \mathbf{v}}{\partial t} + \rho (\mathbf{v} \cdot \nabla) \mathbf{v} = -\nabla p' + (\mu_0 + \mu') \left[ \nabla^2 \mathbf{v} + \frac{1}{3} \nabla(\nabla \cdot \mathbf{v}) \right] \\
+ \frac{4}{3} (\nabla \cdot \mathbf{v}) \nabla \mu' - \nabla \times (\nabla \times \mathbf{v}) 
\]  

(A.25)

Here density is not separated into static and excess density, but is left as the total quantity.

The nondimensional form of the equation is produced by replacing each of the variables with the product of its nondimensional variable and its characteristic quantity. For excess pressure \( p' = p_{ch}\hat{p}' \), and for excess viscosity \( \mu' = \mu_{ch}\hat{\mu}' \). Normalizing the equation by the leading coefficient results in the following form.

\[
\hat{\rho} \frac{\partial \hat{\mathbf{V}}}{\partial \hat{t}} + \hat{\rho}(\hat{\mathbf{V}} \cdot \hat{\nabla})\hat{\mathbf{V}} = -\hat{\nabla} p_{ch} + \frac{\mu_0}{\rho_0 c_0^2} \left( \frac{\mu_{ch}}{\mu_0} \hat{\mu}' \right) \left[ \hat{\nabla}^2 \hat{\mathbf{V}} + \frac{1}{3} \hat{\nabla}(\hat{\nabla} \cdot \hat{\mathbf{V}}) \right] \\
+ \frac{4}{3} (\hat{\nabla} \cdot \hat{\mathbf{V}}) \hat{\nabla} \hat{\mu}' - \hat{\nabla} \times (\hat{\nabla} \times \hat{\mathbf{V}}) 
\]  

(A.26)

Setting the first coefficient on the right-hand side equal to unity provides the definition for characteristic pressure.

\[
p_{ch} \equiv \rho_0 c_0^2 
\]  

(A.27)

The characteristic viscosity is then defined by setting \( \mu_{ch}/\mu_0 = 1 \) inside the bracketed expression.

\[
\mu_{ch} \equiv \mu_0 
\]  

(A.28)

The numerical parts of the coefficients, 1/3 and 4/3, are not included in this operation.

The coefficient leading the viscous terms in the bracketed expression at the end of Equation A.26 is a nondimensional number that indicates the relative importance of viscosity. It is identified as the Stokes number by Cotaras [70] who references section 3 of the paper by Truesdell [71] for this name. Recalling that
the characteristic time is defined above as \(1/\omega\), this coefficient becomes

\[
\eta = \frac{\mu_0 \omega}{\rho_0 c_0^2}
\]  

(A.29)

which is identified here as the small ordering parameter \(\eta\). Hamilton and Morfey use this same small ordering parameter in their analysis to indicate the relative importance of viscous stresses to pressure fluctuations in a plane wave, although they decline to identify it except as \(\eta\) [57]. It is adopted here as a parameter to indicate the order of dissipative terms in the model equations. These terms include viscous as well as thermal conduction terms as is shown below in the nondimensionalization of the energy equation.

With these coefficients defined, the nondimensional momentum equation may be written as follows.

\[
\hat{\rho} \frac{\partial \hat{\mathbf{v}}}{\partial t} + \hat{\rho} (\hat{\mathbf{v}} \cdot \hat{\nabla}) \hat{\mathbf{v}} = -\hat{\nabla} \hat{p}' + \eta \left\{ (1 + \hat{\mu}') \left[ \hat{\nabla}^2 \hat{\mathbf{v}} + \frac{1}{3} \hat{\nabla} (\hat{\nabla} \cdot \hat{\mathbf{v}}) \right] + \frac{4}{3} (\hat{\nabla} \cdot \hat{\mathbf{v}}) \hat{\nabla} \hat{\mu}' - \hat{\nabla} \hat{\mu}' \times (\hat{\nabla} \times \hat{\mathbf{v}}) \right\}
\]

(A.30)

A.2.4 Nondimensional State Equation

To write the equation of state in nondimensional form it is sufficient to leave the density, pressure, and temperature as total quantities without separating them into static and excess variables. These nondimensional quantities are then defined as \(\hat{\rho} = \rho/\rho_{ch}\), \(\hat{p} = p/p_{ch}\), and \(\hat{T} = T/T_{ch}\). Substituting these relations into Equation A.12 and normalizing by the leading coefficient produces a nondimensional form of the equation of state.

\[
\hat{p} = \frac{\rho_{ch} T_{ch} R}{\hat{\rho} \hat{T}}
\]

(A.31)

Since the characteristic density and pressure are defined in Equations A.21 and A.27, the characteristic temperature may be defined by setting the coefficient on the right-hand side of Equation A.31 equal to one.

\[
\frac{T_{ch} R}{c_0^2} = 1
\]

(A.32)
For an ideal gas the specific gas constant $R = c_v(\gamma - 1)$, and the characteristic temperature is defined by

$$T_{ch} \equiv \frac{c_0^2}{c_v(\gamma - 1)}$$  \hspace{1cm} (A.33)

where $\gamma = c_p/c_v$, the ratio of specific heats. The nondimensional equation of state is then written in its final form.

$$\hat{p} = \hat{\rho} \hat{T}$$  \hspace{1cm} (A.34)

### A.2.5 Nondimensional Energy Equation

The remaining characteristic quantity, that associated with thermal conductivity, is defined through nondimensionalization of the energy equation. As in the case of the momentum equation, only two of the variables are separated into static and excess parts. Temperature is replaced by $T = T_0 + T'$, and thermal conductivity becomes $\kappa = \kappa_0 + \kappa'$. Pressure, density, and viscosity are expressed as total variables since separating them provides no additional information about characteristic quantities.

$$\rho c_v \left[ \frac{\partial T'}{\partial t} + (\mathbf{v} \cdot \nabla) T' \right] = -p \nabla \cdot \mathbf{v} + \frac{1}{2} \mu \phi_{ij} \phi_{ij} + (\kappa_0 + \kappa') \nabla^2 T' + \nabla \kappa' \cdot \nabla T'$$  \hspace{1cm} (A.35)

The components of the rate-of-shear tensor are nondimensionalized by dividing each of them by the characteristic value as shown.

$$\dot{\phi}_{ij} = \frac{\phi_{ij}}{\phi_{ch}} = \frac{x_{ch}}{v_{ch}} \left\{ \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3} \frac{\partial v_k}{\partial x_i} \delta_{ij} \right\}$$  \hspace{1cm} (A.36)

Multiplying by $\phi_{ch}$ and setting the coefficient on the right-hand side to unity defines the characteristic rate of shear as $\phi_{ch} \equiv 1/t_{ch}$.

The following substitutions are made to Equation A.35 using the defined characteristic quantities.

$$\rho \quad = \quad \rho_0 \hat{\rho}$$

$$\hat{p} \quad = \quad \rho_0 c_0^2 \hat{\rho}$$

$$\mu \quad = \quad \mu_0 \hat{\mu}$$
\[ \mathbf{v} = c_0 \hat{\mathbf{v}} \]
\[ \nabla = \frac{\omega}{c_0} \hat{\nabla} \]
\[ \nabla^2 = \frac{\omega^2}{c_0^2} \hat{\nabla}^2 \]
\[ \phi_{ij} \phi_{ij} = \omega^2 \hat{\phi}_{ij} \hat{\phi}_{ij} \]
\[ t = \hat{t}/\omega \]
\[ T = T_{ch} \hat{T} \]
\[ \kappa' = \kappa_{ch} \hat{k}' \]

The characteristic temperature is notated as \( T_{ch} \) at this point for simplicity. These substitutions produce the energy equation in the following nondimensional form.

\[
\frac{\rho_0 c_v c_0}{\mu_0 \omega^2} \rho \left[ \frac{\partial \hat{T}'}{\partial \hat{t}} + (\mathbf{\hat{v}} \cdot \hat{\nabla}) \hat{T}' \right] = -\frac{\rho_0 c_v^2}{\mu_0 \omega^2} \hat{p} \hat{\nabla} \cdot \hat{\mathbf{v}} + \frac{1}{2} \frac{\mu_0 \omega^2}{\mu_0} \hat{\phi}_{ij} \hat{\phi}_{ij} + \frac{\kappa_{0} \omega^2 T_{ch}}{c_v^2} \left[ \hat{\nabla}^2 \hat{T}' + \frac{\kappa_{ch}}{\kappa_0} \hat{k}' \hat{\nabla}^2 \hat{T}' + \frac{\kappa_{ch}}{\kappa_0} \hat{\nabla} \hat{k}' \cdot \hat{\nabla} \hat{T}' \right] \tag{A.37}
\]

Normalizing by the leading coefficient and applying the definition of characteristic temperature from Equation A.33 shows the magnitudes of the right-hand-side terms relative to those on the left-hand side.

\[
\hat{\rho} \left[ \frac{\partial \hat{T}'}{\partial \hat{t}} + (\mathbf{\hat{v}} \cdot \hat{\nabla}) \hat{T}' \right] = -(\gamma - 1) \hat{p} \hat{\nabla} \cdot \hat{\mathbf{v}} + (\gamma - 1) \frac{\mu_0 \omega^2}{\rho_0 c_v^2} \hat{\phi}_{ij} \hat{\phi}_{ij} + \frac{\kappa_{0} \omega}{\rho_0 c_v^2} \left[ \hat{\nabla}^2 \hat{T}' + \frac{\kappa_{ch}}{\kappa_0} \hat{k}' \hat{\nabla}^2 \hat{T}' + \frac{\kappa_{ch}}{\kappa_0} \hat{\nabla} \hat{k}' \cdot \hat{\nabla} \hat{T}' \right] \tag{A.38}
\]

The last two coefficients inside the bracketed expression on the right-hand side can be set equal to unity so that the characteristic thermal conductivity is defined as \( \kappa_{ch} \equiv \kappa_0 \). Using the relations for an ideal gas, \( \sigma = \mu c_p / \kappa \) and \( \gamma = c_p / c_v \), the coefficient for the thermal conduction terms is rewritten

\[
\frac{\kappa_{0} \omega}{\rho_0 c_v^2} = \left( \frac{\mu_0 \omega}{\rho_0 c_v^2} \right) \frac{\gamma}{\sigma} \tag{A.39}
\]

where \( \sigma \) is the Prandtl number and \( \gamma \) is the ratio of specific heats. The small ordering parameter \( \eta \), defined in Equation A.29 is used to show the relative importance
of the dissipative terms, those involving viscosity and thermal conductivity.

\[
\hat{\rho} \left[ \frac{\partial \hat{T}'}{\partial t} + (\hat{\mathbf{v}} \cdot \hat{\nabla}) \hat{T}' \right] = -(\gamma - 1)\hat{\rho} \hat{\nabla} \cdot \hat{\mathbf{v}} + \eta(\gamma - 1)\frac{\mu}{2} \hat{\phi}_{ij} \hat{\phi}_{ij} + (A.40)
\]

\[
\eta \frac{\gamma}{\sigma} \left[ (1 + \hat{\kappa}') \hat{\nabla}^2 \hat{T}' + \hat{\nabla} \hat{\kappa}' \cdot \hat{\nabla} \hat{T}' \right]
\]

The Prandtl number in front of the thermal conduction terms indicates their importance relative to the viscous terms in this equation and in the momentum equation. The factor \((\gamma - 1)\) is equal to the parameter \(B/A\) for a perfect gas, which indicates the amount by which the small-signal sound speed varies for a finite amplitude disturbance [72]. For helium \(\gamma = 1.67\), so this factor has little effect on the choice of small ordering parameters \(\varepsilon\) and \(\eta\) whose magnitudes are discussed below.

### A.3 Relative Magnitudes

The previous section used small ordering parameters to compare the relative orders of magnitude of the various terms in the nondimensionalized equations of motion. This section further investigates the relative sizes of terms and variables by assigning numerical values from the simulation under consideration [1]. Specifically the ratio of fluctuating to ambient quantities of the flow variables as well as those of the fluctuating material properties, viscosity and thermal conductivity, are compared to the Mach number. These comparisons will help later in defining ordered Taylor series expansions of the variables.

It is important to recognize the different meanings of the phrase “order of magnitude.” In this analysis, order of magnitude is represented by the notation \(O(\epsilon)\) where the ordering parameter \(\epsilon\) is either the Mach number \(\varepsilon\) or the Stokes number \(\eta\). What this says is that given some function \(f(x; \epsilon)\), it is of the order of \(g(x; \epsilon)\) as \(\epsilon\) approaches \(\epsilon_0\) if there exists another function \(M(x)\) such that \(|f| \leq M|g|\). That is \(f(x; \epsilon) = O[g(x; \epsilon)]\) if \(f(x; \epsilon)/g(x; \epsilon)\) is bounded as \(\epsilon \to \epsilon_0\). That’s a fairly broad definition of “order of magnitude.” In fact, \(M\) could be \(10^3\), and it would still be correct to say \(f = O(g)\) rather than the more common assertion that \(f\) is “three orders of magnitude” greater than \(g\).

To say that \(f\) is similar in value to \(g\) it is more appropriate to use the notation
\( f(x; \epsilon) \sim g(x; \epsilon) \). This statement says that the two functions are similar in value if the ratio \( f(x; \epsilon)/g(x; \epsilon) \rightarrow 1 \) as \( \epsilon \rightarrow \epsilon_0 \). While this notation may appear to be the better choice since it is more precise than the big O notation, it should be noted that this gain in precision comes at the expense of information about the shapes of the functions. [68]

To proceed in comparing relative magnitudes to ordering parameters, consider that the pressure amplitude reported by Olson and Swift in their experiment reached approximately \( 1.95 \times 10^5 \) Pa at the heat exchanger at the hot end of the pulse tube [1]. This corresponds to a sound pressure level of 197 dB which will serve as the basis for the calculation of magnitudes of the rest of the variables.

The working gas in Olson and Swift’s experiment was helium which at a mean pressure of \( 3.1 \times 10^6 \) Pa and mean temperature of 200 K has an ambient density of \( 7.462 \text{ kg/m}^3 \) and speed of sound 832.12 m/s. To linear order this translates to a particle velocity magnitude of \( u_o = |p|/\rho_0 c_0 = 31.4 \text{ m/s} \), a density magnitude of \( |\rho| = |p|/\rho_0^2 = 0.282 \text{ kg/m}^3 \), and a temperature magnitude of \( |T| = |p|T_0 \beta/\rho_0 c_p = 5.033 \text{ K} \) where \( \beta = 0.5 \times 10^{-2} \text{ K}^{-1} \) and \( c_p = 5192 \text{ J/kg} \cdot \text{K} \). The material properties of Helium are taken from DeltaE software for thermoacoustic design [73].

The magnitudes of the remaining variables, viscosity \( \mu \) and thermal conductivity \( \kappa \), may be computed using appropriate forms of Sutherland’s formula. The temperature dependent viscosity of helium is expressed as

\[
\mu = \mu_0 \frac{T_0 + C}{T + C} \left( \frac{T}{T_0} \right)^{3/2}
\]

where Sutherland’s constant for helium \( C = 70 \text{ K} \).[54] Taking the value reported by DeltaE for viscosity, \( \mu_0 = 1.5133 \times 10^{-5} \text{ kg/s} \cdot \text{m} \), reveals that the viscosity magnitude \( |\mu| = \mu - \mu_0 = 2.8736 \times 10^{-7} \text{ kg/s} \cdot \text{m} \). Similarly the magnitude of variations in thermal conductivity is found to be \( |\kappa| = \kappa - \kappa_0 = 0.00227 \text{ W/m} \cdot \text{K} \) by way of Sutherland’s formula

\[
\kappa = \kappa_{273} \frac{273 + C}{273 + C} \left( \frac{T}{273} \right)^{3/2}
\]

where \( \kappa_{273} = 0.139 \text{ W/m} \cdot \text{K} \), \( C = 33 \text{ K} \),[54] and \( \kappa_0 = 0.11402 \text{ W/m} \cdot \text{K} \) [73].

Now that the magnitudes of the fluctuating variables are calculated, it is pos-
sible to normalize them and compare with the values of the small ordering parameters. These parameters may be calculated as follows:

\[ \varepsilon = \frac{u_0}{c_0} = 3.773 \times 10^{-2} \]

\[ \eta = \frac{\mu f}{\rho_0 c_0^2} = 5.858 \times 10^{-10} \]

The normalized magnitudes of the variables are listed here and compared.

\[ \frac{|p|}{p_0} = 6.290 \times 10^{-2} \sim \varepsilon \]
\[ \frac{|\rho|}{\rho_0} = 3.779 \times 10^{-2} \sim \varepsilon \]
\[ \frac{|T|}{T_0} = 2.517 \times 10^{-2} \sim \varepsilon \]
\[ \frac{|\mu|}{\mu_0} = 1.899 \times 10^{-2} \sim \varepsilon \]
\[ \frac{|\kappa|}{\kappa_0} = 1.991 \times 10^{-2} \sim \varepsilon \]

This analysis reveals that the fluctuating quantities of the flow variables as well as those of the viscosity and thermal conductivity are all of the order of the Mach number. Only the terms associated with viscosity \( \mu \) and thermal conductivity \( \kappa \) are of the order \( \eta \). This result will help in determining how to implement these ordering parameters in a Taylor series solution for each variable in the perturbation method.

### A.3.1 Ordered Equations

It would be straightforward to define the equations of motion to second order if there were only one small ordering parameter. In this analysis, however, there are two, the Mach number and the Stokes number. Hamilton and Morfey address this problem by following the example of Lighthill in his 1956 article *Viscosity effects in sound waves of finite amplitude* [74]. They assert that \( \varepsilon \) and \( \eta \) are of "comparable smallness" and introduce "a generic small parameter \( \tilde{\varepsilon} \) that characterizes the smallness of both." With this reasoning they decide to keep terms of order \( \tilde{\varepsilon} \) and \( \tilde{\varepsilon}^2 \), and discard terms of order \( \tilde{\varepsilon}^3 \) and higher. This includes \( \varepsilon, \eta \varepsilon, \) and \( \varepsilon^2 \), and excludes \( \eta \varepsilon^2, \varepsilon^3 \), and higher order terms.

It may be useful in general to say that \( \varepsilon \) and \( \eta \) are of similar smallness, but
in light of the discussion above on relative magnitudes, it could be argued that in this particular simulation the two small ordering parameters are not similar in magnitude. Furthermore, including terms of order $\eta$ while omitting those that are higher order in $\varepsilon$ may exclude certain terms that are, in fact, more significant than the largest $\eta$-ordered viscous terms. This can be shown by noting that the value of $\eta$ is smaller than that of $\varepsilon^6$ using the numbers calculated above. If one includes terms of order $\eta$, one may as well include terms of order $\varepsilon^6$ and larger. For the sake of demonstrating asymptotic methods and comparing the formal perturbation expansion to the acoustic disturbance method, these extra terms in higher orders of $\varepsilon$ will not be included, but it will suffice to note that they are not necessarily insignificant contributions compared to viscous terms.

A.3.2 Discussion

This perturbation analysis of the equations of motion results in a set of equations at three different levels of approximation in $\bar{\varepsilon}$: zero through second order. At the zero order there is only the hydrostatic equation, $\nabla p_0 = \rho_0 g_0$. This single equation could provide a solution for either $p_0$ or $\rho_0$ if the other is known, but it leaves zero-order temperature, $T_0$, to be determined by other means. This could be declared constant from the start, or calculated from a windowed time average of the total variables as the solution progresses. The solution at higher orders is more straightforward since there are five equations and five unknowns of order $\bar{\varepsilon}$ and $\bar{\varepsilon}^2$.

Interestingly, if the zero-order variables are allowed to vary in time as well as space, the resulting zero-order equations of continuity and energy balance dictate that the time dependence of both zero-order density and zero-order temperature is zero. As a consequence the higher order equations are the same as in the above analysis. If, on the other hand, the zero-order variables are denied any variation, then there are no zero-order equations, leaving those quantities to be declared constant from the start.

Perturbation methods of solving systems of equations are complicated by the many choices available for the definition of the zero-order quantities. Herein lies the art of setting up the computation to provide meaningful answers.
A.4 Acoustic Disturbance Method

The acoustic disturbance method differs from the perturbation expansion primarily in that the equations of motion are not separated into ordered equations. The higher order terms in the acoustic disturbance method are treated as “source” terms and placed on the right-hand side of each equation. The zero-order variables are not allowed any time or space dependence, but are held constant as ambient values. Any deviation in the total quantity from this ambient value is represented by the fluctuating part of the variable. Thus the fluctuating part may represent acoustic variations as well as a non-zero mean value over any spatial range or period in time. The fluctuating variables represent anything that is not ambient.

The following is a list of the flow variables and material properties expanded into ambient and fluctuating values. Note that since the ambient velocity is zero, the zero-order components of the rate-of-shear tensor $\phi_{ij}$ disappear.

\[
\begin{align*}
\rho &= \rho_0 + \rho' \\
u &= u' \\
p &= p_0 + p' \\
T &= T_0 + T' \\
\phi_{ij} &= \phi'_{ij} \\
\mu &= \mu_0 + \mu' \\
\kappa &= \kappa_0 + \kappa'
\end{align*}
\]

These expanded solutions are substituted into the equations of motion to arrive at the acoustic disturbance equations.

A.5 Conclusions

This article attempts to show that two methods of perturbation analysis, a formal perturbation method of ordered equations and the acoustic disturbance method, are consistent with each other. While the final comparison between the methods may lack mathematical rigor, the process of expanding the five equations of motion brings out some subtle and important points.
First it is important to understand what is meant by the phrase “order of magnitude” and the notations that represent it. The tilde notation, $f \sim g$, indicates that the quantities are actually similar in value, while the big O notation, $f = O(g)$, means only that their ratio is bounded and not necessarily close to 1. This analysis tends to use the former definition, but more research will be needed to determine which notation is appropriate at particular times.

The second subtlety involves the definition of zero-order quantities. They may be considered as constant values or as the mean value of the total variable in time and/or space over some range. This choice may affect not only the zero-order equations but also some first-order equations. Keeping track of the mean zero-order value of a variable allows the higher order values to have a zero mean, but it introduces many more computing steps. The computation of this analysis is based on constant zero-order quantities which are easier to compute and leave the representation of any developing mean value to the fluctuating part of the variables.

Finally, a study of the relative magnitudes of variables and small ordering parameters is helpful in relating the mathematics of the perturbation expansion to the physics of the model. Comparing the magnitudes of the two ordering parameters in this analysis reveals that $\eta < \varepsilon^6$. This suggests that $\eta$ and $\varepsilon$ are not necessarily of similar smallness, as reported by Lighthill[74] via Hamilton and Morfey [75], and that it could be more important to include terms of order $\varepsilon^6$ before terms of order $\eta$ come into play. While this suggestion might be impractical to follow in computing a system of separate ordered equations, it is not inconsistent with the acoustic disturbance method which represents all higher orders of a variable in the fluctuating part. The question then remains whether to include terms of order $\eta \varepsilon^2$ which are of order $\bar{\varepsilon}^3$ by one analysis but not that much smaller than $\eta \varepsilon$ by another. These order $\eta \varepsilon^2$ terms would include fluctuating viscosity and thermal conductivity, and it is one of the goals of the current research to investigate the effects of their inclusion or exclusion in the computational model.
The steps described by Thompson to derive the Method-of-Characteristics boundary condition for non-reflecting boundaries are employed here using equations that solve for the primitive quantities of density, velocity, and temperature in two dimensions [62, 76]. The development process is shown for both cylindrical and spherical coordinates, and the resulting boundary equations are perturbed to provide linear and nonlinear solutions. Non-reflecting boundaries are established in only one of the two dimensions in each coordinate system.

B.1 Governing Equations

The primitive equations of conservation of mass, momentum, and energy are presented below in vector notation. The energy equation is written in terms of temperature via the caloric equation of state, \( e = c_v T \), which assumes a gas of constant specific heats. The momentum equation includes acceleration due to gravity. Dissipative terms are not included since the MOC boundary condition does not account for them.

\[
\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0 \tag{B.1}
\]

\[
\rho \frac{D\mathbf{v}}{Dt} + \nabla p = \rho \mathbf{g} \tag{B.2}
\]

\[
\rho c_v \frac{DT}{Dt} + p \nabla \cdot \mathbf{v} = 0 \tag{B.3}
\]
B.2 Cylindrical Coordinates

The governing equations are cast in cylindrical coordinates with the help of the relations listed in Section 2.3.1. Assuming axisymmetry where variation in the \( \hat{\theta} \)-direction is neglected, the equations are expressed in terms of the remaining coordinates, \( z \) and \( r \).

Continuity:

\[
\frac{\partial \rho}{\partial t} + \rho \frac{\partial v_z}{\partial z} + v_z \frac{\partial \rho}{\partial z} + v_r \frac{\partial \rho}{\partial r} + \frac{\rho v_r}{r} = 0 \tag{B.4}
\]

Momentum-\( \hat{z} \):

\[
\frac{\partial v_z}{\partial t} + v_z \frac{\partial v_z}{\partial z} + v_r \frac{\partial v_z}{\partial r} + \frac{p}{\rho^2} \frac{\partial \rho}{\partial z} + \frac{p}{\rho T} \frac{\partial T}{\partial z} = g_z \tag{B.5}
\]

Momentum-\( \hat{r} \):

\[
\frac{\partial v_r}{\partial t} + v_z \frac{\partial v_r}{\partial z} + v_r \frac{\partial v_r}{\partial r} + \frac{p}{\rho^2} \frac{\partial p}{\partial r} + \frac{p}{\rho T} \frac{\partial T}{\partial r} = 0 \tag{B.6}
\]

Energy:

\[
\frac{\partial T}{\partial t} + v_z \frac{\partial T}{\partial z} + v_r \frac{\partial T}{\partial r} + \frac{p}{\rho c_v} \frac{\partial v_z}{\partial z} + \frac{p}{\rho c_v} \frac{\partial v_r}{\partial r} = 0 \tag{B.7}
\]

These are written in the form of a single partial differential equation of the form

\[
\frac{\partial \mathbf{w}}{\partial t} + \mathbf{A} \frac{\partial \mathbf{w}}{\partial z} + \mathbf{B} \frac{\partial \mathbf{w}}{\partial r} + \mathbf{C}_z + \mathbf{C}_r = 0 \tag{B.8}
\]

where the matrices are defined as follows:

\[
\mathbf{w} = \begin{bmatrix} \rho \\ v_z \\ v_r \\ T \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} v_z & \rho & 0 & 0 \\ \frac{p}{\rho^2} & v_z & 0 & \frac{p}{\rho T} \\ 0 & 0 & v_z & 0 \\ 0 & \frac{p}{\rho c_v} & 0 & v_z \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} v_r & 0 & \rho & 0 \\ 0 & v_r & 0 & 0 \\ \frac{p}{\rho^2} & 0 & v_r & \frac{p}{\rho T} \\ 0 & 0 & \frac{p}{\rho c_v} & v_r \end{bmatrix}
\]

\[
\mathbf{C}_z = \begin{bmatrix} 0 \\ -g_z \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{C}_r = \begin{bmatrix} \frac{\rho v_z}{\rho T} \\ 0 \\ -g_r \\ \frac{\rho v}{\rho c_v r} \end{bmatrix} \tag{B.9}
\]

The eigenvalues of the matrix \( \mathbf{A} \) are found to be the following:

\[
\lambda_1 = v_z - c \quad \lambda_2 = v_z \quad \lambda_3 = v_z \quad \lambda_4 = v_z + c \tag{B.10}
\]
The left eigenvectors $l_i$ are calculated such that $l_i A = \lambda_i l_i$. The eigenvectors corresponding to the eigenvalues are expressed here as rows of a single matrix.

$$
\begin{bmatrix}
    l_1 \\
    l_2 \\
    l_3 \\
    l_4
\end{bmatrix} =
\begin{bmatrix}
    T & -\frac{\rho^2 c T}{p} & 0 & \rho \\
    -p & 0 & 0 & \rho^2 c_v \\
    0 & 0 & 1 & 0 \\
    T & \frac{\rho^2 c T}{p} & 0 & \rho
\end{bmatrix}
$$

(B.11)

The characteristic equations in the \( \hat{z} \)-direction are computed from the equation

$$
l_i \frac{\partial w}{\partial t_z} + \lambda_i l_i \frac{\partial w}{\partial z} + l_i C_z = 0.
$$

(B.12)

This results in the following system of equations:

$$
\begin{align*}
T \frac{\partial \rho}{\partial t_z} & - \frac{\rho^2 c T}{p} \frac{\partial v_z}{\partial t_z} + \rho \frac{\partial T}{\partial t_z} \\
& + (v_z - c) \left( T \frac{\partial \rho}{\partial z} - \frac{\rho^2 c T}{p} \frac{\partial v_z}{\partial z} + \frac{\rho \partial T}{\partial z} \right) + \frac{\rho^2 c T}{p} g_z = 0 \\
-p \frac{\partial \rho}{\partial t_z} & + \rho^2 c_v \frac{\partial T}{\partial t_z} + v_z \left( -p \frac{\partial \rho}{\partial z} + \frac{\rho^2 c_v \partial T}{\partial z} \right) = 0 \\
\frac{\partial v_r}{\partial t_z} & + v_z \frac{\partial v_r}{\partial z} = 0 \\
T \frac{\partial \rho}{\partial t_z} & + \frac{\rho^2 c T}{p} \frac{\partial v_z}{\partial t_z} + \rho \frac{\partial T}{\partial t_z} \\
& + (v_z + c) \left( T \frac{\partial \rho}{\partial z} + \frac{\rho^2 c T}{p} \frac{\partial v_z}{\partial z} + \frac{\rho \partial T}{\partial z} \right) - \frac{\rho^2 c T}{p} g_z = 0
\end{align*}
$$

(B.13)

Next the boundary conditions, \( \mathcal{L}_i \), are defined at the tube ends where \( z = 0 \) and \( z = L \).

$$
\mathcal{L}_i = \begin{cases} 
\lambda_i l_i \frac{\partial w}{\partial z} & \text{outgoing} \\
0 & \text{incoming}
\end{cases}
$$

(B.14)

The \( \hat{z} \)-direction time derivatives are computed by the equation

$$
l_i \frac{\partial w}{\partial t_z} + \mathcal{L}_i + l_i C_z = 0.
$$

(B.15)
This results in the following set of boundary equations:

\[
\frac{\partial \rho}{\partial t_z} = \frac{p}{\rho T} \left[ \frac{1}{\rho c_p} \mathcal{L}_2 - \frac{1}{2c^2} \left( \mathcal{L}_1 + \mathcal{L}_4 \right) \right]
\]

\[
\frac{\partial v_z}{\partial t_z} = \frac{p}{2 \rho^2 c T} \left( \mathcal{L}_1 - \mathcal{L}_4 \right) + g_z \tag{B.16}
\]

\[
\frac{\partial v_r}{\partial t_z} = -\mathcal{L}_3
\]

\[
\frac{\partial T}{\partial t_z} = -\frac{1}{\rho^2 c_p} \left[ \frac{p}{2T} \left( \mathcal{L}_1 + \mathcal{L}_4 \right) + \mathcal{L}_2 \right]
\]

At the \( z = 0 \) end with no mean flow, \( \lambda_1 = -c \), which is directed outward, \( \lambda_2 = \lambda_3 = 0 \), and \( \lambda_4 = c \), which is directed inward. This leaves only \( \mathcal{L}_1 \) with a nonzero value.

\[
\mathcal{L}_1 = -c T \frac{\partial \rho}{\partial z} + \rho^2 c^2 T \frac{\partial v_z}{\partial z} - \rho \frac{\partial T}{\partial z} \tag{B.17}
\]

At \( z = L \) the incoming and outgoing fluctuations are reversed, and only \( \mathcal{L}_4 \) is nonzero.

\[
\mathcal{L}_4 = c T \frac{\partial \rho}{\partial z} + \rho^2 c^2 T \frac{\partial v_z}{\partial z} + \rho \frac{\partial T}{\partial z} \tag{B.18}
\]

A perturbation expansion of variables gives a linear version and a nonlinear version of the boundary conditions. At \( z = 0 \) the nonzero linear boundary condition is given by

\[
\mathcal{L}_1' = -c T_0 \frac{\partial \rho'}{\partial z} + \rho_0 \gamma T_0 \frac{\partial v_z}{\partial z} - \rho_0 c \frac{\partial T'}{\partial z}, \tag{B.19}
\]

and the nonlinear version is

\[
\mathcal{L}_1'' = -c (T_0 + T') \frac{\partial \rho'}{\partial z} + \gamma (\rho_0 T_0 + \rho_0 T' + \rho' T_0) \frac{\partial v_z}{\partial z} - c (\rho_0 + \rho') \frac{\partial T'}{\partial z}. \tag{B.20}
\]

At \( z = L \) the nonzero linear boundary condition is written as

\[
\mathcal{L}_4' = c T_0 \frac{\partial \rho'}{\partial z} + \rho_0 \gamma T_0 \frac{\partial v_z}{\partial z} + \rho_0 c \frac{\partial T'}{\partial z}, \tag{B.21}
\]

and the nonlinear version is

\[
\mathcal{L}_4'' = c (T_0 + T') \frac{\partial \rho'}{\partial z} + \gamma (\rho_0 T_0 + \rho_0 T' + \rho' T_0) \frac{\partial v_z}{\partial z} + c (\rho_0 + \rho') \frac{\partial T'}{\partial z}. \tag{B.22}
\]
Perturbation expansion produces a linear and a nonlinear version of the \( \hat{z} \)-direction boundary equations as well. The linearized equations are listed here in Equations B.23.

\[
\begin{align*}
\frac{\partial \rho'}{\partial t_z} &= \frac{1}{\rho_0 T_0 c_p} \mathcal{L}'_2 - \frac{P_0}{2 \rho_0 c^2 T_0} (\mathcal{L}'_1 + \mathcal{L}'_4) \\
\frac{\partial v_z}{\partial t_z} &= \frac{P_0}{2 \rho_0^2 T_0 c} (\mathcal{L}'_1 - \mathcal{L}'_4) + \left( 1 + \frac{2 \rho'}{\rho_0} + \frac{T'}{T_0} \right) g_z \\
\frac{\partial v_r}{\partial t_z} &= -\mathcal{L}'_3 \\
\frac{\partial T'}{\partial t_z} &= -\frac{1}{\rho_0^2 T_0 c_p} \left[ \frac{1}{2} P_0 (\mathcal{L}'_1 + \mathcal{L}'_4) + T_0 \mathcal{L}'_2 \right]
\end{align*}
\]

The nonlinear boundary equations are written as follows:

\[
\begin{align*}
\frac{\partial \rho'}{\partial t_z} &= \frac{1}{\rho_0 T_0 c_p} \mathcal{L}''_2 - \frac{1}{2 \rho_0 c^2 T_0} (P_0 + p') (\mathcal{L}''_1 + \mathcal{L}''_4) - \left( \frac{T'}{T_0} + \frac{\rho'}{\rho_0} \right) \left[ \frac{1}{\rho_0 T_0 c_p} \mathcal{L}'_2 - \frac{P_0}{2 \rho_0 c^2 T_0} (\mathcal{L}'_1 + \mathcal{L}'_4) \right] \\
\frac{\partial v_z}{\partial t_z} &= \frac{P_0}{2 \rho_0^2 T_0 c} \left[ (\mathcal{L}''_1 - \mathcal{L}''_4) - \left( \frac{T'}{T_0} + 2 \frac{\rho'}{\rho_0} - \frac{p'}{P_0} \right) (\mathcal{L}'_1 - \mathcal{L}'_4) \right] \\
&\quad + \left[ 1 - 3 \left( \frac{\rho'}{\rho_0} \right)^2 - \left( \frac{T'}{T_0} \right)^2 - \frac{\rho'T'}{\rho_0 T_0} \right] g_z \\
\frac{\partial v_r}{\partial t_z} &= -\mathcal{L}''_3 \\
\frac{\partial T'}{\partial t_z} &= -\frac{1}{\rho_0^2 c_p} \left[ \frac{P_0}{2} \left( \frac{T'}{T_0} + 2 \frac{\rho'}{\rho_0} - \frac{p'}{P_0} \right) (\mathcal{L}'_1 + \mathcal{L}'_4) \right. \\
&\quad \left. + 2 \frac{\rho'T_0}{\rho_0} \mathcal{L}'_2 - \frac{P_0}{2} (\mathcal{L}''_1 + \mathcal{L}''_4) + T_0 \mathcal{L}'_2 \right]
\end{align*}
\]

These boundary conditions and equations are used at the non-reflective ends of models of standing waves in cylindrical tubes. They are instrumental in the two-source method of standing-wave generation, which is described by Boluriaan and Morris [30] and illustrated in Section 4.1.2.
B.3 Spherical Coordinates

The spherical-coordinate equations are derived from the governing equations listed in Section B.1 through the use of the relations shown in Equations 2.41 to 2.45. Here, as in the cylindrical-coordinate equations, the assumption that the fluid motion is axisymmetric allows variation in the $\hat{\phi}$-direction to be neglected.

Continuity:

$$\frac{\partial \rho}{\partial t} + v_r \frac{\partial \rho}{\partial r} + \rho \frac{\partial v_r}{\partial r} + \frac{\partial \rho}{\partial \theta} + \frac{\partial \rho}{\partial \theta} + \frac{2}{r} \rho v_r + \frac{1}{r} \rho \cot \theta v_\theta = 0 \quad (B.25)$$

Momentum-$\hat{r}$:

$$\frac{\partial v_r}{\partial t} + p \frac{\partial v_r}{\partial \rho} + v_r \frac{\partial v_r}{\partial r} + p \frac{\partial T}{\partial r} + v_\theta \frac{\partial v_r}{\partial \theta} = g_r \quad (B.26)$$

Momentum-$\hat{\theta}$:

$$\frac{\partial v_\theta}{\partial t} + v_r \frac{\partial v_\theta}{\partial r} + p \frac{\partial v_\theta}{\partial \rho} + v_\theta \frac{\partial v_\theta}{\partial \theta} + p \frac{\partial T}{\partial \theta} = g_\theta \quad (B.27)$$

Energy:

$$\frac{\partial T}{\partial t} + \frac{p}{\rho c_v} \frac{\partial v_r}{\partial r} + v_r \frac{\partial T}{\partial r} + \frac{p}{r \rho c_v} \frac{\partial v_\theta}{\partial \theta} + \frac{v_\theta}{r} \frac{\partial T}{\partial \theta} + 2 \frac{p v_r}{r \rho c_v} + \frac{p \cot \theta v_\theta}{r \rho c_v} = 0 \quad (B.28)$$

These equations may be written succinctly in the form

$$\frac{\partial \mathbf{w}}{\partial t} + \mathbf{A} \frac{\partial \mathbf{w}}{\partial r} + \mathbf{B} \frac{\partial \mathbf{w}}{\partial \theta} + \mathbf{C}_r + \mathbf{C}_\theta = 0 \quad (B.29)$$

where the matrices are defined as follows:

$$\mathbf{w} = \begin{bmatrix} \rho \\ v_r \\ v_\theta \\ T \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} v_r & \rho & 0 & 0 \\ \frac{p}{\rho r^2} & v_r & 0 & \frac{p}{\rho c_v} \\ 0 & 0 & v_r & 0 \\ 0 & \frac{p}{\rho c_v} & 0 & v_r \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \frac{\rho v_\theta}{r} & 0 & \frac{\rho}{r} & 0 \\ 0 & \frac{v_\theta}{r} & 0 & 0 \\ \frac{p}{r \rho c_v} & 0 & \frac{v_\theta}{r} & \frac{p}{r \rho c_v} \\ 0 & 0 & \frac{p}{r \rho c_v} & \frac{v_\theta}{r} \end{bmatrix}$$
\[
C_r = \begin{bmatrix}
\frac{2\rho v_r}{r} \\
-g_r \\
0 \\
\frac{2pv_r}{rpc_v}
\end{bmatrix} \quad C_\theta = \begin{bmatrix}
\frac{2}{r} \cot \theta v_\theta \\
0 \\
-g_\theta \\
\frac{p}{rpc_v} \cot \theta v_\theta
\end{bmatrix} \quad \text{(B.30)}
\]

The eigenvalues of matrix \(A\) are shown in Equations B.31.

\[
\lambda_1 = v_r - c, \quad \lambda_2 = v_r, \quad \lambda_3 = v_r, \quad \lambda_4 = v_r + c \quad \text{(B.31)}
\]

The relation \(l_i A = \lambda_i l_i\) produces a set of left eigenvectors \(l_i\) that may be expressed as rows of a single matrix.

\[
\begin{bmatrix}
l_1 \\
l_2 \\
l_3 \\
l_4
\end{bmatrix} = \begin{bmatrix}
T & -\frac{\rho^2 c T}{p} & 0 & \rho \\
-p & 0 & 0 & \rho^2 c_v \\
0 & 0 & 1 & 0 \\
T & \frac{\rho^2 c T}{p} & 0 & \rho
\end{bmatrix} \quad \text{(B.32)}
\]

These eigenvectors are then used to compute the characteristic equations in the \(\hat{r}\)-direction by the equation

\[
l_i \frac{\partial w}{\partial t_r} + \lambda_i l_i \frac{\partial w}{\partial r} + l_i C_r = 0. \quad \text{(B.33)}
\]

Expanding this matrix equation produces the following system of characteristic equations:

\[
T \frac{\partial \rho}{\partial t_r} - \frac{\rho^2 c T}{p} \frac{\partial v_r}{\partial t_r} + \rho \frac{\partial T}{\partial t_r} + (v_r - c) \left( T \frac{\partial \rho}{\partial r} - \frac{\rho c^2 T}{p} \frac{\partial v_r}{\partial r} + \rho \frac{\partial T}{\partial r} \right) + \frac{2\rho T v_r}{r} + \frac{\rho^2 c T}{p} g_r + \frac{2pv_r}{rc_v} = 0
\]

\[
-p \frac{\partial \rho}{\partial t_r} + \rho c_v \frac{\partial T}{\partial t_r} + v_r \left( -p \frac{\partial \rho}{\partial r} + \rho c_v \frac{\partial T}{\partial r} \right) = 0
\]

\[
\frac{\partial v_r}{\partial t_r} + v_r \frac{\partial v_\theta}{\partial r} = 0
\]

\[
T \frac{\partial \rho}{\partial t_r} + \frac{\rho^2 c T}{p} \frac{\partial v_r}{\partial t_r} + \rho \frac{\partial T}{\partial t_r} + (v_r + c) \left( T \frac{\partial \rho}{\partial r} - \frac{\rho c^2 T}{p} \frac{\partial v_r}{\partial r} + \rho \frac{\partial T}{\partial r} \right) + \frac{2\rho T v_r}{r} - \frac{\rho^2 c T}{p} g_r + \frac{2pv_r}{rc_v} = 0
\]
At the ends of the tapered tube, \( r = 0 \) and \( r = L \), the boundary conditions, \( \mathcal{L}_i \), are defined for incoming and outgoing disturbances.

\[
\mathcal{L}_i = \begin{cases} 
\lambda_i l_i \frac{\partial w}{\partial r} & \text{outgoing} \\
0 & \text{incoming}
\end{cases} 
\]  
(B.35)

Equation B.36 uses the eigenvectors and boundary conditions to compute the \( \hat{r} \)-direction time derivatives as

\[
l_i \frac{\partial w}{\partial t} + \mathcal{L}_i + l_i C_r = 0, 
\]  
(B.36)

which gives the set of boundary equations listed in Equations B.37.

\[
\begin{align*}
\frac{\partial \rho}{\partial t} &= \frac{p}{\rho T} \left[ \frac{1}{\rho c_p} \mathcal{L}_2 - \frac{1}{2c^2} (\mathcal{L}_1 + \mathcal{L}_4) \right] - \frac{2\rho v_r}{r} \\
\frac{\partial v_r}{\partial t} &= \frac{p}{2\rho^2 c T} (\mathcal{L}_1 - \mathcal{L}_4) + g_r \\
\frac{\partial v_\theta}{\partial t} &= -\mathcal{L}_3 \\
\frac{\partial T}{\partial t} &= -\frac{1}{\rho^2 c_p} \left[ \frac{p}{2T} (\mathcal{L}_1 + \mathcal{L}_4) + \mathcal{L}_2 \right] - \frac{2pv_r}{r \rho c_v}
\end{align*}
\]  
(B.37)

In the case of no mean flow, the eigenvalues at end \( r = 0 \) are \( \lambda_1 = -c \), which is directed outward, \( \lambda_2 = \lambda_3 = 0 \), and \( \lambda_4 = c \), which is directed inward. As in the cylindrical coordinate system, only \( \mathcal{L}_1 \) has a nonzero value.

\[
\mathcal{L}_1 = -cT \frac{\partial \rho}{\partial r} + \rho^2 c^2 T \frac{\partial v_r}{\partial r} - \rho c \frac{\partial T}{\partial r} 
\]  
(B.38)

At \( r = L \) the incoming and outgoing fluctuations are reversed, and only remaining nonzero boundary condition is

\[
\mathcal{L}_4 = cT \frac{\partial \rho}{\partial r} + \frac{\rho^2 c^2 T}{p} \frac{\partial v_r}{\partial r} + \rho c \frac{\partial T}{\partial r}.
\]  
(B.39)

The boundary conditions may be expressed to linear or nonlinear order by writing the variables as the sum of their zero-order and fluctuating quantities. At
the boundary where \( r = 0 \) the nonzero linear boundary condition is expressed as

\[
\mathcal{L}_1' = -cT_0 \frac{\partial \rho'}{\partial r} + \rho_0 \gamma T_0 \frac{\partial v_r}{\partial r} - \rho_0 c \frac{\partial T'}{\partial r},
\]  

(B.40)

and the nonlinear boundary condition is written

\[
\mathcal{L}_1'' = -c(T_0 + T') \frac{\partial \rho'}{\partial r} + \gamma(\rho_0 T_0 + \rho_0 T' + \rho'T_0) \frac{\partial v_r}{\partial r} - c(\rho_0 + \rho') \frac{\partial T'}{\partial r}.
\]  

(B.41)

At \( r = L \) the nonzero linear boundary condition is

\[
\mathcal{L}_4' = cT_0 \frac{\partial \rho'}{\partial r} + \rho_0 \gamma T_0 \frac{\partial v_r}{\partial r} + \rho_0 c \frac{\partial T'}{\partial r},
\]  

(B.42)

and to higher order it is written as

\[
\mathcal{L}_4'' = c(T_0 + T') \frac{\partial \rho'}{\partial r} + \gamma(\rho_0 T_0 + \rho_0 T' + \rho'T_0) \frac{\partial v_r}{\partial r} + c(\rho_0 + \rho') \frac{\partial T'}{\partial r}.
\]  

(B.43)

The linearized \( \hat{z} \)-direction boundary equations in spherical coordinates are expressed through the use of perturbation expansion as follows:

\[
\frac{\partial \rho'}{\partial t_r} = \frac{1}{\rho_0 T_0 c_p} \mathcal{L}_2' - \frac{P_0}{2 \rho_0 c^2 T_0} (\mathcal{L}_1' + \mathcal{L}_4') - \frac{2 \rho_0 v_r}{r},
\]

\[
\frac{\partial v_r}{\partial t_r} = \frac{P_0}{2 \rho_0^2 T_0 c} (\mathcal{L}_1' - \mathcal{L}_4') + \left( 1 + \frac{2 \rho'}{\rho_0} + \frac{T'}{T_0} \right) g_r,
\]  

(B.44)

\[
\frac{\partial T'}{\partial t_r} = -\mathcal{L}_3'
\]

\[
\frac{\partial v_r}{\partial t_r} = -\frac{1}{\rho_0 T_0 c_p} \left[ \frac{1}{2} P_0 (\mathcal{L}_1' + \mathcal{L}_4') + T_0 \mathcal{L}_2' \right] - \frac{2 P_0 v_r}{r \rho_0 c_v}.
\]

The nonlinear boundary equations are then provided in Equations B.45.
\[
+ \left[ 1 - 3 \left( \frac{\rho'}{\rho_0} \right)^2 - \left( \frac{T'}{T_0} \right)^2 - \frac{\rho'T'}{\rho_0 T_0} \right] g_r
\] (B.45)

\[
\frac{\partial v_r}{\partial t_r} = -L'_3
\]

\[
\frac{\partial T'}{\partial t_r} = -\frac{1}{\rho_0 c_p} \left[ \frac{P_0}{2} \left( \frac{T'}{T_0} + 2 \frac{\rho'}{\rho_0} - \frac{p'}{P_0} \right) \left( L'_1 + L'_4 \right) \right.
\]

\[
+ 2 \frac{\rho'T_0}{\rho_0} L'_2 - \frac{P_0}{2} (L''_1 + L''_4) + T_0 L'_2 \bigg] - \frac{2P_0}{r \rho_0 c_v} \left( 1 + \frac{\rho'}{\rho_0} + \frac{T'}{T_0} + \frac{p'}{P_0} \right) v_r
\]

### B.4 Summary

The Method-of-Characteristics boundary condition was developed for the non-reflecting ends of straight and tapered model tubes. The governing equations were written in cylindrical and spherical coordinate systems with the assumption of axisymmetric fluid motion. The derivations followed the steps provided by Thompson [62] and resulted in boundary conditions and boundary equations for the case of zero steady flow in both coordinate systems. Perturbation techniques were used on the primitive variables to provide linear and nonlinear versions of the boundary conditions and equations. The resulting expressions were used in the implementation of the MOC boundary condition described in Section 3.3.
To determine the validity of the streaming solutions using the criteria for grid resolution outlined in Section 5.1, grids of three different resolutions are compared. An asymptotic solution for grid intervals approaching zero is calculated using Richardson Extrapolation, and a region of convergence around this solution is computed by the grid-convergence-index method proposed by Roache [77]. The grid resolution study described here shows that the criteria used for the clustered grid in the radial direction of a straight tube produces a solution for streaming that is just outside the lower range of convergence as determined by Roache’s method.

C.1 Method

This grid refinement analysis is based on the solutions of the acoustic streaming simulation calculated with three different spatial discretizations. The model used for this study is the straight tube of length equal to two wavelengths. The ends of the model tube use a non-reflective boundary, and the wall is rigid with no-slip velocity boundary condition and adiabatic temperature boundary condition. The model equations include terms up to $O(\varepsilon^2)$ and $O(\eta \varepsilon)$ consistent with the results labeled “ND” in Section 5.3. Complete details of the model are given in Section 5.1.

The first grid is discretized using the coarsest resolution with 60 grid points in the radial direction and a minimum radial grid spacing of $\Delta r_{\text{min}} = 9.1 \mu m$ adjacent to the tube wall. In the axial direction there are 200 grid points, or 100 points per wavelength. This is the grid resolution that is typical of the streaming simulations in straight tubes and follows the same guidelines for resolving acoustic fluctuations.
as well as variations within the viscous boundary layer for tapered tubes, i.e. at least six grid points within a boundary layer adjacent to a rigid surface. The second grid is refined with 72 points in the radial direction and a minimum radial grid spacing of $\Delta r_{\text{min}} = 7.6 \mu m$ near the wall. The third grid is the most refined with 86 points in the radial direction and a minimum radial grid spacing of $\Delta r_{\text{min}} = 6.3 \mu m$ at the wall. The axial grid resolution remains unchanged for this refinement study in keeping with Roache’s suggestion that one spatial dimension is sufficient for grid resolution analysis when the other spatial dimensions are well resolved and further refinement is computationally expensive [77].

Computational cost also plays a role in the choice of refinement ratio between the three grids. Here the ratio of the number of grid points in the radial direction between grids 1 and 2 and grids 2 and 3 is approximately 1.2. This relation $\sigma$ between grids holds for the minimum radial grid spacing as well. Typically a grid refinement ratio of $\sigma = 2$ is chosen for a straightforward doubling of grid points. In cases where the cost of the computation is high, however, a non-integer ratio may be used as long as $\sigma \geq 1.1$ [77]. To achieve a consistent streaming solution with each of the grids, the Courant number is maintained at 0.4. This causes each of the three discretizations of the model to run with a different time step $\Delta t$ according to Equation 3.29. Not only do the finer resolution models use larger computational domains due to the increased number of grid points, but they also use a smaller time step and must compute more of them to reach the same modeled time. In each of the models the axial streaming velocity $u_{st}$ is computed as in Equation 4.13 over ten cycles from periods 540 to 550. A grid refinement ratio close to the minimum allows for this long computation at finer resolutions.

Using a non-integer grid refinement ratio results in few if any colocated grid points between the three spatial discretizations. This necessitates interpolation to compare the solution at common points along the tube radius. One readily available interpolation method is the cubic spline. This is applied to each of the three streaming solutions, taking into account the symmetric boundary at the tube axis where $r = 0$ and where the gradient of axial streaming velocity with respect to the radius is assumed to be zero as well. This choice of cubic spline interpolation is used for the first analysis in the grid refinement study. For the analysis, two points are selected along the radius at position $A$ where $r = 0.100R$ and at position $B$
where \( r = 0.997R \). Since the order of the cubic spline interpolation exceeds that of the spatial finite difference method, the validity of this analysis may be called into question. A second analysis using linear interpolation is also presented for comparison.

### C.2 Analysis

Following the approach described by Roache \[77\], the estimated asymptotic solution as grid spacing approaches zero is computed by Richardson Extrapolation using the solutions from the two finer grids, \( u_{st2} \) and \( u_{st3} \). This is accomplished with the equation

\[
 u_{st}\big|_{\Delta r \to 0} = u_{st3} + \frac{(u_{st3} - u_{st2})}{(\sigma^p - 1)}
\]

where \( \sigma \) is the refinement ratio and \( p \) is the order of convergence. The order of convergence is calculated using the streaming solutions from the three different grid resolutions.

\[
 p = \ln \left[ \frac{u_{st1} - u_{st2}}{u_{st2} - u_{st3}} \right] \frac{1}{\ln(\sigma)}
\]

The relative error of the solution from the coarse grid (grid 1) with respect to that from the medium refined grid (grid 2) is given by

\[
 \varepsilon_{12} = \frac{u_{st1} - u_{st2}}{u_{st2}}.
\]

The error of the solution from the medium refined grid with respect to that from the finest grid solution (grid 3) is

\[
 \varepsilon_{23} = \frac{u_{st2} - u_{st3}}{u_{st3}}.
\]

These relative errors are used to defined the grid refinement index (GCI) which provides an alternate error estimate taking into account the grid refinement ratio and order of convergence. For coarse resolution the GCI is computed using grids 1 and 2 as

\[
 \text{GCI}_{12} = \frac{F_s|\varepsilon_{12}|}{\sigma^p - 1}.
\]
and for fine resolution grids 2 and 3 are used such that
\[
GCI_{23} = \frac{F_s|\varepsilon_{23}|}{\sigma^p - 1}.
\] (C.6)

The “factor of safety” \( F_s \) is set to the recommended value of 1.25 for a three-grid study [77].

C.3 Results

The solutions for axial streaming velocity computed using three different radial grid resolutions are shown in Figure C.1 along with the radial positions \( A \) and \( B \). The streaming values interpolated at these points using cubic spline interpolation are listed in Table C.1. Equation C.1 gives the asymptotic solution at each point. At position \( A \), \( u_{st}(A) = 0.3716 \text{ m/s} \), and at position \( B \), \( u_{st}(B) = 0.0456 \text{ m/s} \). The observed order of convergence given by Equation C.2 is 2.1 at position \( A \) and 1.4 at position \( B \). This compares reasonably well to the theoretical order of convergence \( p = 2 \), which is assumed from the second-order Taylor series approximation used.
Table C.1. Streaming velocities calculated over three grids for positions A and B found through cubic spline interpolation

<table>
<thead>
<tr>
<th>Grid</th>
<th>Number of points</th>
<th>$u_{st}(A)$ [m/s]</th>
<th>$u_{st}(B)$ [m/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>86</td>
<td>0.3611</td>
<td>0.0440</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>0.3645</td>
<td>0.0444</td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>0.3667</td>
<td>0.0447</td>
</tr>
</tbody>
</table>

Table C.2. Relative error magnitudes and Grid Convergence Indices for points A and B found through cubic spline interpolation

<table>
<thead>
<tr>
<th>Grids</th>
<th>Position A</th>
<th></th>
<th>Position B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\varepsilon$</td>
<td>GCI $\times 100%$</td>
<td>$\varepsilon$</td>
<td>GCI $\times 100%$</td>
</tr>
<tr>
<td>2,3</td>
<td>0.0062</td>
<td>1.65</td>
<td>0.0062</td>
<td>2.58</td>
</tr>
<tr>
<td>1,2</td>
<td>0.0092</td>
<td>2.45</td>
<td>0.0082</td>
<td>3.38</td>
</tr>
</tbody>
</table>

in the Richardson Extrapolation. Much of the difference between the theoretical and computed order of convergence is likely due to grid stretching [77].

Table C.2 lists the results of the grid refinement analysis using the cubic-spline interpolated values. Both the relative error magnitudes and the Grid Convergence Indices at the coarse and fine grid levels are listed. It is assumed in the calculation of Richardson Extrapolation and the Grid Convergence Index that the Taylor series expansion on which they depend converges asymptotically. It is also important that the two solutions used in the GCI calculation are within this asymptotic range. This can be verified by comparing the GCI values at fine and coarse grid levels.

$$GCI_{12} = \sigma^pGCI_{23}$$  \hspace{1cm} (C.7)

At both positions A and B the ratio $GCI_{12}/(\sigma^pGCI_{23}) = 1.0$, so the range-of-convergence assumption is satisfied in this case.

Figure C.2 shows the three solutions at each position and their cubic spline interpolation. The extrapolated solution appears as a single point at each position with error bars surrounding it to show the fine Grid Convergence Index, $GCI_{23}$. It can be seen that the coarse grid solution, which was used in the streaming analysis, falls just outside of the lower error bar at each position.
Figure C.2. Streaming velocity profiles calculated over three grids. The extrapolated solution is shown for positions A and B.

Table C.3. Streaming velocities calculated over three grids for positions A and B found through linear interpolation

<table>
<thead>
<tr>
<th>Grid</th>
<th>Number of points</th>
<th>$u_{st}(A)$ [m/s]</th>
<th>$u_{st}(B)$ [m/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>86</td>
<td>0.3652</td>
<td>0.0443</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>0.3626</td>
<td>0.0437</td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>0.3594</td>
<td>0.0427</td>
</tr>
</tbody>
</table>

As mentioned above, the order of the cubic spline interpolation exceeds that of the second-order-in-space finite difference method. For comparison a similar grid analysis is presented using linear interpolation. Table C.3 lists these interpolated values for the axial streaming velocity at positions A and B. The asymptotic solutions are computed at position A as $u_{st}(A) = 0.3762 \text{ m/s}$, and at position B as $u_{st}(B) = 0.0461 \text{ m/s}$. The observed order of convergence is also computed at each point giving $p(A) = 1.2$ and $p(B) = 1.8$. These again differ from the theoretical order of convergence, $p = 2$, and the effect of the lower order interpolation is evident when these values are compared to those computed above using cubic spline interpolation.
Table C.4. Relative error magnitudes and Grid Convergence Indices for points A and B found through linear interpolation

<table>
<thead>
<tr>
<th>Grids</th>
<th>$\varepsilon$</th>
<th>GCI $\times 100%$</th>
<th>$\varepsilon$</th>
<th>GCI $\times 100%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,3</td>
<td>0.0071</td>
<td>3.78</td>
<td>0.0156</td>
<td>4.90</td>
</tr>
<tr>
<td>1,2</td>
<td>0.0088</td>
<td>4.70</td>
<td>0.0222</td>
<td>6.96</td>
</tr>
</tbody>
</table>

The Grid Convergence Indices and relative error magnitudes calculated from the linearly interpolated solutions are shown in Table C.4. The range of convergence can be verified as above by comparing the GCI values at fine and coarse grid levels. The ratio $\text{GCI}_{12}/(\sigma^p\text{GCI}_{23}) = 1.0$ at each position using the linearly interpolated solutions.

Figure C.3 shows the three solutions at each position along with their linear interpolation and extrapolated solutions. The error bars indicating fine Grid Convergence Index, $\text{GCI}_{23}$, are wider for the analysis using linear interpolation, but the coarse-grid solution again falls outside of the lower error bar at each position.

This grid resolution analysis shows that increasing the number of grid points...
in the radial direction of the straight-tube model causes relatively small changes in the solution for axial streaming velocity. That the coarse-grid solution falls just outside of the range of the Grid Convergence Index for the fine grid level indicates a nearly optimal choice for coarse-grid resolution in the radial direction. The coarse grid offers a less computationally expensive task, and the resulting solution is not significantly improved by an increase in resolution.
Appendix

D Model of Streaming Simulation

The details of the numerical model of acoustic streaming are presented here for reference only. For a working version, please contact the author.

D.1 Compiling

To compile the model for parallel implementation, supply correct environment variables for your system and type `make` using the following makefile.

D.1.1 Makefile

# Makefile

# Define compiler.
# FC = /usr/local/bin/mpiabsf90

# Compiler options
# FFLAGS = -O2

# Libraries
# CDFlib = -L/usr/local/netCDF/lib -lnetcdf
LIBS = $(CDFlib)

# Linker options
# MPIincl = -I/usr/local/mpich-gm-abs/include
LDFLAGS = $(MPIincl)

# Suffix-rules: Begin by throwing away all old suffix-
# rules, and then create new ones for compiling
# *.f90-files.
.SUFFIXES:
.SUFFIXES: .f90 .o
.f90.o:
$(FC) -c $(FFLAGS) $(LDFLAGS) $(LIBS) $<

# Dependencies
FOBJ=main.o mod_boundary.o mod_FDiff_vars.o mod_global.o mod_input_vars.o
mod_grid.o mod_IC.o mod_analyt.o mod_init.o mod_IOtools.o mod_acstypes.o
mod_maccormack_cyl.o mod_maccormack_sph.o mod_source.o mod_plot3d.o
mod_mpifun.o

stream: $(FOBJ)
$(FC) -o $@ $(FFLAGS) $(LDFLAGS) $(FOBJ) $(LIBS)

main.o : main.f90 mod_maccormack_cyl.o mod_maccormack_sph.o mod_analyt.o
       mod_IOtools.o mod_init.o mod_global.o

mod_boundary.o : mod_boundary.f90 mod_mpifun.o mod_source.o mod_grid.o
     mod_FDiff_vars.o mod_input_vars.o mod_global.o

mod_FDiff_vars.o : mod_FDiff_vars.f90 mod_global.o

mod_global.o : mod_global.f90

mod_input_vars.o : mod_input_vars.f90 mod_global.o

mod_grid.o : mod_grid.f90 mod_mpifun.o mod_input_vars.o mod_global.o

mod_IC.o : mod_IC.f90 mod_mpifun.o mod_source.o mod_acstypes.o mod_analyt.o
       mod_input_vars.o mod_IOtools.o mod_grid.o mod_global.o

mod_analyt.o : mod_analyt.f90 mod_grid.o mod_FDiff_vars.o mod_mpifun.o
       mod_input_vars.o mod_global.o

mod_init.o : mod_init.f90 mod_input_vars.o mod_mpifun.o mod_boundary.o
       mod_source.o mod_analyt.o mod_IC.o mod_IOtools.o mod_grid.o mod_global.o

mod_IOtools.o : mod_IOtools.f90 mod_plot3d.o mod_source.o mod_analyt.o
     mod_FDiff_vars.o mod_mpifun.o mod_input_vars.o mod_acstypes.o mod_grid.o
     mod_global.o

mod_acstypes.o : mod_acstypes.f90 mod_global.o

mod_maccormack_cyl.o : mod_maccormack_cyl.f90 mod_analyt.o mod_source.o
    mod_boundary.o mod_IOtools.o mod_input_vars.o mod_grid.o mod_FDiff_vars.o
    mod_global.o

mod_maccormack_sph.o : mod_maccormack_sph.f90 mod_analyt.o mod_source.o
    mod_boundary.o mod_IOtools.o mod_input_vars.o mod_grid.o mod_FDiff_vars.o
    mod_global.o

mod_source.o : mod_source.f90 mod_FDiff_vars.o mod_grid.o mod_analyt.o
      mod_input_vars.o mod_global.o
D.2 Running the Simulation

As this is a model in development, many of the input parameters are deprecated, and some have multiple functions depending on the context. The important parameters are described at the bottom of the file. The function of any other parameter may be traced through the subroutine read_input in the file init.fpp. A sample submit file is provided for use with parallel computing platforms using the Portable Batch System (PBS) scheduler.

D.2.1 input

--- Geometry ---
radius = 0.0087
length = 16.6424d0
taper = 0.0d0
Req_pos = -1.0d0

--- Grid ---
CFL = 0.4d0
BL points = 2
__minimum grid spacing
delta1_x1 = 0.0832d0
delta2_x1 = 0.0832d0
delta1_x2 = 0.00135d0
delta2_x2 = 0.0000091d0

__maximum grid spacing
Delta_x1 = 0.0832d0
Delta_x2 = 0.00135d0

__grid size
i_size = 200
j_size = 60

--- Timing ---
txmin = 0
txmax = 100
drv_pdur = 700 0 16

--- Initial Condition ---
ic_choice = 0
theta(deg)= 0.0
infile =
P_tgt = 195000.0
k = 0.0
bigB = 0.0

--- Source ---
freq = 100.0
fdrv type = lin
Accel.0 = 97500.0d0
Accel.1 = 97500.0d0
Accel.2 = 1785.7d0
Pos1 [%L] = 12.5d0
Pos2 [%L] = 87.5d0
SPpk [T] = 0.25d0
FPdel [T] = 0.0d0
FPram[T] = 0.5d0

--- Boundaries ---
wall_BC = rig
ctr_BC = sym
end1_BC = moc
dend2_BC = moc

wall_TBC = adi
dend1_TBC = adi
dend2_TBC = adi

wall_VBC = ns
dend1_VBC = sl
dend2_VBC = sl

--- Calculation ---
gravity = F
exagT = 1.0d0
exagV = 1.0d0

--- Output ---
output = per
outarg1 = 10
outarg2 = 0
setname = ADA
format = p3d
anlyt T = F
mean U = bin
--- Probe ---
probe out = F
datapos = 2
prb_Xi = 2
prb_Eta = 10
prb_start = 0
prb_stop = 0
p_p_cyc = 100

--- Restart ---
interval = 1000
drv_rint = 2 0 16

Choices:
ic_choice
 1 = Gaussian pulse on z-axis
 2 = Centered Gaussian pulse
 3 = Gaussian pulse at z=length/5
 4 = Gaussian pulse at r=length/5
 5 = Standing wave
 6 = Standing wave based on damped natural frequency
 7 = Read initial pressure, z-velocity, and drive phase angle
 8 = Try starting with pressure and velocity at piston phase
 9 = Read initialization from restart file
 0 = default: quiescent medium

Boundary
rig = rigid
sym = symmetric
per = periodic
pis = piston (left and right boundaries only)
moc = absorbing

Output
reg = Write a file every N (=100) time steps
per = Write acoustic variables for one period after filetimex
pks = Write periodic maximum pressure values to a file
end = Write files after a certain time

Format
bin = Fortran unformatted binary (acsdata type structure)
p3d = Plot3D format
ncf = netCDF
D.2.2 submit

```bash
#PBS -j oe
#PBS -r n
#PBS -N NDstrADa
#PBS -l nodes=4:ppn=2
#PBS -l walltime=23:59:00

# The program to be run.
setenv PROG "${HOME}/work/Doctoral_Research/NonreflBC/bin/stream_ndL.xc"

cd $PBS_O_WORKDIR

# Print nodenames to stdout.
setenv NPROCS 'wc -l < $PBS_NODEFILE'
echo This jobs runs on the following $NPROCS processors:
echo 'cat $PBS_NODEFILE'
echo "Job started at: 'date'"

if (${NPROCS} > 1) then
time mpirun ${PROG}
else
time ${PROG}
endif

echo "Job finished at: 'date'"
```

D.3 Restarting the Simulation

Periodically the simulation writes out a binary file of model variables. On parallel systems each processor writes its portion of the computational domain to the shared file system. To restart the simulation using these files, they must be merged using the post program listed below. This program reads the file ppinput provided here. Once the restart files are merged, the simulation can be restarted by specifying ic_choice = 9 and the name of the infile in the input file.

D.3.1 ppinput

```plaintext
--- Action ---
merging    = T
format     = bin
variables  = all

--- File Choice ---
output     = per
```
setname = GRc_522_00:16
tx digits = 1
pr digits = 1
start ndx = 1
stop ndx = 1
step ndx = 1
D.4 Program Listing

D.4.1 main.fpp

PROGRAM stream

!=============================================================================
! This program calculates nonlinear acoustic streaming in a closed tube.
! The computation space is a 2-dimensional cross section of half a cylinder.
!
! Here's the domain:
! rigid wall
! +-------------------------------------------------------------+
! (j) | |
! r^ 1 | | end 2
! | | |
! | | |
! r=0|------------------- --- --------------------|--
! z=0 center line - symmetric BC
! ----> z (i)
!
!=============================================================================

USE global
USE init
USE IO_tools
USE analyt
USE maccormack_sph
USE maccormack_cyl

implicit none

integer(i4b) :: stat

! Initialize the program: Setup variables and computational space according
! to the input file.
! call initialize(stat)

if (stat == 0) then

! Write initial condition to a file.
! lastout = txmin
! if (analytic) then
! call calc_anl(txmin)
! end if
! call write_acsvars(txmin)

! Propagate the wave.
! if (taper == 0.0d0) then
! call propagate_cyl()
else
call propagate_sph()
end if

! Write the final result to a file.
call write_acsvars(txmax)
end if

! Clean up and put away the toys.
call finalize()

END PROGRAM stream
D.4.2 global.fpp

MODULE global
!
! This module contains the global variables for stream2d.
!

! Precision
integer, parameter :: DP = kind(1.0d0)
integer, parameter :: SP = kind(1.0)
integer, parameter :: i4b = Selected_Int_Kind(9)

! Constants
real(DP) :: Pi
real(DP) :: Pisq
real(DP) :: P_ref
real(DP) :: G
real(DP) :: onethd, twothds, fourthds

! Gas variables
real(DP) :: P0, C0, C02, rho0, rho02, T0, Tconst, mu0, k0, Cp, Cv, Rgas
real(DP) :: D3cf, D4cf, SBCcf1, SBCcf2, BoAcf, NLS2cf, delta_K, Prandtl
real(DP) :: gamma_ratio

! Coefficients
real(DP) :: PoRHOCv0, ONEoRHOCv0, PoRHO0, ONEoRHO0, MUoRHO0, twothdsMU0
real(DP) :: ONEoRHO02

! Grid variables
integer, parameter :: BP = 1       ! Number of boundary points
integer :: ni, nj
integer :: ni_global, nj_global
integer :: i_offset, j_offset
integer :: ni_main, nj_main

! Global boundary variables
logical :: periodic
character(len=3) :: end1_BCgl, end2_BCgl, end1_TBCgl, end2_TBCgl
character(len=2) :: end1_VBCgl, end2_VBCgl

! Timing variables
integer :: periodx, subperx, txdigs
real(DP) :: deltat, dtdx
!
real(DP), dimension(:,:), allocatable :: dtdz, dtdr

! Source variables
real(DP) :: omega
real(DP) :: wvn_k        ! wavenumber
real(DP) :: srcphi
real(DP) :: A0_last, A0_tgt

! Output variables
  integer :: lastout, prdigs
  character(len=15) :: analoutfile, slicefile, periodfile
  character(len=20) :: prbfile

! Analysis variables
  type timept_type
    integer :: timex
    real(DP) :: value
  end type timept_type
  type(timept_type) :: timepoint
  integer :: n_sum

! Main calculation arrays
! Acoustic variables:
  integer, parameter :: nACSvar = 5
  integer, parameter :: nSACSvar = 7
  real(DP), dimension(:,,:,:), allocatable :: acs
  real(DP), dimension(:,,:,:), allocatable :: sumacs

! Indexing parameters for acs array.
  integer, parameter :: RHOx = 1
  integer, parameter :: Ux = 2
  integer, parameter :: Vx = 3
  integer, parameter :: Tx = 4
  integer, parameter :: Px = 5

CONTAINS

FUNCTION cat(str1, str2) RESULT(str3)

  implicit none

  character(len=*) :: str1, str2
  character(len=len(str1)+len(str2)) :: str3

  str3(1:len(str1)) = str1
  str3(len(str1)+1:len(str3)) = str2

END FUNCTION cat

END MODULE global
D.4.3 init.fpp

MODULE init
!===========================================================================
! This module contains the subroutines that are used to initialize the
! whole stream2d program. There is one main subroutine called init_stream
! that calls the rest.
!
! Table of contents:
! SUBROUTINE initialize()
! SUBROUTINE parameters()
! SUBROUTINE read_input()
! SUBROUTINE print_vars()
! SUBROUTINE alloc_arrays()
! SUBROUTINE finalize()
!
!===========================================================================

USE global
implicit none

CONTAINS
!===========================================================================
SUBROUTINE initialize(init_stat)
! =======================================================================
! | This subroutine initializes the whole stream2d program by setting |
! | a few global variables which will govern the program's behavior. |
! | This is read from an input file. |
! =======================================================================

USE grid
USE IO_tools
USE IC
USE analyt
USE source
USE boundary
#ifdef HAVE_MPI
USE mpi_fun
#endif

implicit none

integer(i4b), intent(OUT) :: init_stat

! Initialize number of digits for processor id in output file names.
prdigs = 0

#ifdef HAVE_MPI
! Initialize MPI variables. (mod_mpifun)
call orient_mpi()
#endif

! Read the input file. (this module)
call read_input()

! Define constants and physical parameters. (this module)
call parameters()

! Calculate grid variables: i_size, j_size, deltat, and periodx (mod_grid)
call init_grid()

! Initialize output files. (mod_IOtools)
call init_out()

! Print out the variables. (this module)
call print_vars()

! Once i_size and j_size are known, allocate the acs calculation arrays.
call alloc_arrays()

call init_analyt()

! Create a grid. (mod_grid)
call makegrid()

! Set up volume velocity calculation. (mod_analyt)
call init_Uvol()

! Set initial conditions. (mod_IC)
call init_cond(init_stat)

! Allocate source variables.
call init_source()

call init_bounds()

RETURN

END SUBROUTINE initialize

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------

SUBROUTINE parameters()
! =======================================================================
! | This subroutine assigns values to constants and parameters used |
! | throughout the program. |
! =======================================================================

USE input_vars
implicit none

real(DP) :: BetaT, BoverA, amu, R0

! Constants
Pi = 3.141592653589793d0
PiSq = Pi*Pi
P_ref = 20.0e-6 ! reference pressure in Pascals
G = 9.8d0 ! acceleration due to gravity (m/s)
R0 = 8314.0d0 ! Universal gas constant (J/kg/K)
onethd = 1.0d0 / 3.0d0
twothds = 2.0d0 / 3.0d0
tfourthds = 4.0d0 / 3.0d0

! Gas variables (Helium)
P0 = 3100000.0d0 ! Ambient pressure
C0 = 832.12d0 ! Speed of sound (m/s)
T0 = 200.0d0 ! Mean temperature (Kelvin)
BetaT = 0.005d0 ! Coefficient of thermal expansion (1/K)
rho0 = 7.462d0 ! Mean density (kg/m^3)
amu = 4.003d0 ! average molecular weight
Rgas = R0/amu ! gas constant
gamma_ratio = 1.67d0 ! Ratio of specific heats

Cp = 5192.0d0 ! Specific heat at const. press. (J/kg/K)
Cv = Cp/gamma_ratio ! Specific heat at const. vol. (J/kg/K)

mu0 = 0.000015133d0 ! Bulk viscosity (kg/s/m)
u0 = mu0*exagV*exagV

k0 = 0.11402d0 ! Thermal conductivity (W/m/K)
k0 = k0*exagT*exagT

! Derived constants
Prandtl = mu0*Cp / k0
C02 = C0*C0
rho02 = rho0*rho0
BoverA = 1.0d0 - gamma_ratio ! B/A nonlinearity ratio
BoAcf = 0.5d0*BoverA/rho0 ! B/A coefficient for first nonlin. state eq.
Tconst = T0*BetaT/(rho0*Cp)

! Linear
PoRHOCv0 = P0/(rho0*Cv) ! Coef. for linear temperature eq.

! Nonlinear
ONEoRHOCv0 = 1.0d0/(rho0*Cv)
PoRHO0 = P0/rho0

! Low-order dissipative
ONEoRHO0 = 1.0d0 / rho0
ONEoRHO02 = 1.0d0 / rho02
twothdsMU0 = twothds * mu0
MUoRHO0 = mu0 / rho0
D3cf = C02*rho0*BetaT*T0/Cp ! Coefficient for 1st dissipative state eq.
D4cf = T0/Cp ! Coefficient for 2nd dissipative state eq.
NLS2cf = 0.5d0/(rho0*C02) ! Coefficient for second nonlinear state eq.
SBCcf1 = Cp/T0 ! First coefficient for entropy BC
SBCcf2 = BetaT/rho0 ! Second coefficient for entropy BC

! Source variables
omega = fsrc * 2.0d0 * Pi ! Source frequency in radians/sec.
srcphi = 0.0d0 !Pi/2.0d0 ! Phase shift of source.
wvn_k = omega / C0 ! wavenumber

RETURN

END SUBROUTINE parameters

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE read_input
!==========================================================================
! This subroutine reads from a file called input and assigns values to program variables.
!==========================================================================

USE input_vars
USE IO_tools
#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
character(len=5), parameter :: fname = 'input'
integer, parameter :: IU = 10
integer, parameter :: tabstop = 12
integer :: ierr, istat
#ifdef HAVE_MPI
if (myid == MASTER) then
#endif
! Read input file
open (UNIT=IU, FILE=fname, STATUS='OLD', ACTION='READ', &
FORM='FORMATTED', IOSTAT=ierr)
! --- Geometry ---
call skipline(IU, 1)
call skiptab(IU, tabstop)
read(IU, "'(f8.6)'", IOSTAT=istat) radius
call skiptab(IU, tabstop)
read(IU, "'(f8.6)'", IOSTAT=istat) L
call skiptab(IU, tabstop)
read(IU, "'(f8.6)'", IOSTAT=istat) taper
call skiptab(IU, tabstop)
#endif
read(IU, "(f8.6)", IOSTAT=istat) Req_pos

! --- Grid ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(f4.2)", IOSTAT=istat) CFL
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) nBL
call skipline(IU, 1)
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) min_delta1_x1
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) min_delta2_x1
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) min_delta1_x2
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) min_delta2_x2
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) max_delta_x1
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) max_delta_x2
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) i_size
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) j_size

! --- Timing ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) txmin
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) txmax
call skiptab(IU, tabstop)
read(IU, "(i4,2i2)", IOSTAT=istat) drv_pdur
! default:
if (drv_pdur(3) == 0) drv_pdur = (/ 1, 0, 16 /)

! --- Initial Condition ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(i1)", IOSTAT=istat) ic_choice
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) theta
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) IC_file
call skiptab(IU, tabstop)
read(IU, "(f18.3)", IOSTAT=istat) P_tgt
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) k
call skiptab(IU, tabstop)
read(IU, "(f12.9)", IOSTAT=istat) bigB

! --- Source ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(f12.8)", IOSTAT=istat) fsrc
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) fdrv_type
call skiptab(IU, tabstop)
read(IU, "(f12.4)", IOSTAT=istat) S_0
call skiptab(IU, tabstop)
read(IU, "(f12.4)", IOSTAT=istat) S_1
call skiptab(IU, tabstop)
read(IU, "(f12.4)", IOSTAT=istat) S_2
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) src_pos1
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) src_pos2
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) SPpeak
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) SARamp_rate
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) SARamp_dur

! --- Boundaries ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) wall_BC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) ctr_BC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end1_BC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end2_BC
call skipline(IU, 1)
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) wall_TBC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end1_TBC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end2_TBC
call skipline(IU, 1)
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) wall_VBC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end1_VBC
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) end2_VBC
! --- Calculation ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(11)", IOSTAT=istat) gravity
call skiptab(IU, tabstop)
read(IU, "(f12.6)", IOSTAT=istat) exagT
call skiptab(IU, tabstop)
read(IU, "(f12.6)", IOSTAT=istat) exagV

! --- Output ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) output
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) outarg1
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) outarg2
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) setname
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) outfnt
call skiptab(IU, tabstop)
read(IU, "(11)", IOSTAT=istat) analytic
call skiptab(IU, tabstop)
read(IU, "(a)", IOSTAT=istat) streamout

! --- Analysis ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(11)", IOSTAT=istat) probing
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) datapos
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) prb_xi
call skiptab(IU, tabstop)
read(IU, "(i4)", IOSTAT=istat) prb_eta
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) prb_start
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) prb_stop
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) prb_per_cyc
call skiptab(IU, tabstop)
read(IU, "(f8.4)", IOSTAT=istat) Uvol_pos

! --- Restart ---
call skipline(IU, 2)
call skiptab(IU, tabstop)
read(IU, "(i9)", IOSTAT=istat) restint
call skiptab(IU, tabstop)
read(IU, "(i4,2i2)", IOSTAT=istat) drv_rint
! Close the file
close (UNIT=IU)
#ifdef HAVE_MPI
    end if
#endif

! Master process broadcasts input data to all processes.
call share_input_mpi()
#endif

! Adjust boundary condition specs periodic BC.
  periodic = .false.        ! Initialize to false

  if (end1_BC  'per' .or. end2_BC  'per') then ! both have to be 'per'
    end1_BC = 'per'
    end2_BC = 'per'
  end if

  if (end1_BC == 'per') then
    periodic = .true.
    end1_TBC = 'per'
    end2_TBC = 'per'
    end1_VBC = 'pd'
    end2_VBC = 'pd'
  end if

! Adjust boundary condition specs for multiple processors.
  end1_BCgl = end1_BC
  end2_BCgl = end2_BC
  end1_TBCgl = end1_TBC
  end2_TBCgl = end2_TBC
  end1_VBCgl = end1_VBC
  end2_VBCgl = end2_VBC

#ifdef HAVE_MPI
  if (myid > MASTER) then
    end1_BC = "mpi"
    end1_TBC = "mpi"
    end1_VBC = "mp"
  end if
  if (myid < nprocs-1) then
    end2_BC = "mpi"
    end2_TBC = "mpi"
    end2_VBC = "mp"
  end if
#endif

RETURN

END SUBROUTINE read_input
!----------------------------------------------------------------------
SUBROUTINE print_vars()

! This subroutine prints out the values of the variables that have been assigned in the initialization of the program.

USE IO_tools
USE grid
USE IC
USE input_vars
USE source

#ifdef HAVE_MPI
USE mpi_fun
#endif

character(len=5) :: dx_units

if (taper == 0.0d0) then
  dx_units = "(m)"
else
  dx_units = "(rad)"
end if

#ifdef HAVE_MPI
if (myid == MASTER) then
#endif

print *, "--- Geometry ---"
print *, "radius = ", radius, "(m)"
print *, "length = ", L, "(m)"
print *, "taper = ", taper, "(rad)"
print *, "Req_pos = ", Req_pos

print *, "--- Grid ---"
print *, "CFL = ", CFL
print *, "i_size = ", ni
print *, "j_size = ", nj

#ifdef HAVE_MPI
print *, "ni_global = ", ni_global
print *, "nj_global = ", nj_global
#endif

print *, "min_delta1_x1 = ", min_delta1_x1, "(m)"
print *, "min_delta2_x1 = ", min_delta2_x1, "(m)"
print *, "min_delta1_x2 = ", min_delta1_x2, dx_units
print *, "min_delta2_x2 = ", min_delta2_x2, dx_units
print *, "max_delta_x1 = ", max_delta_x1, "(m)"
print *, "max_delta_x2 = ", max_delta_x2, dx_units
print *, "alph1_x1 = ", alph1_x1
print *, "alph1_x2 = ", alph1_x2
print *, "alph2_x1 = ", alph2_x1
print *, "alph_x2 = ", alph_x2
print *, "--- Timing ---"
print *, "txmin = ", txmin
print *, "txmax = ", txmax
print *, "deltat = ", deltat, ",(sec)"
print *, "periodx = ", periodx
print *, "drv_pdur = ", drv_pdur
print *, "--- Initial Condition ---"
print *, "ic_choice = ", ic_choice
print *, "theta = ", theta
print *, "IC_file = ", IC_file
print *, "P_tgt = ", P_tgt
print *, "k = ", k
print *, "bigB = ", bigB
print *, "--- Source ---"
print *, "omega = ", omega
print *, "fdrv_type = ", fdrv_type
print *, "Accel.0 = ", S_0
print *, "Accel.1 = ", S_1
print *, "Accel.2 = ", S_2
print *, "Pos1 [% L] = ", src_pos1
print *, "Pos2 [% L] = ", src_pos2
print *, "SPpeak [T] = ", SPpeak
print *, "SAramp_rate = ", SAramp_rate
print *, "SAramp_dur[T] = ", SAramp_dur
print *, "--- Boundaries ---"
print *, "wall_BC = ", wall_BC
print *, "ctr_BC = ", ctr_BC
print *, "end1_BC = ", end1_BC
print *, "end2_BC = ", end2_BC
print *, "wall_TBC = ", wall_TBC
print *, "end1_TBC = ", end1_TBC
print *, "end2_TBC = ", end2_TBC
print *, "wall_VBC = ", wall_VBC
print *, "end1_VBC = ", end1_VBC
print *, "end2_VBC = ", end2_VBC
print *, "--- Calculation ---"
print *, "gravity = ", gravity
print *, "exagT = ", exagT
print *, "exagV = ", exagV
print *, "--- Output ---"
print *, "output = ", output
print *, "outarg1 = ", outarg1
print *, "outarg2 = ", outarg2
print *, "setname = ", setname
print *, "format = ", outfmt
print *, "analytic = ", analytic
print *, "streamout = ", streamout
print *, "--- Probe ---"
print *, "probe out = ", probing
print *, "datapos = ", datapos
print *, "prb_xi = ", prb_xi
print *, "prb_eta = ", prb_eta
print *, "prb_start = ", prb_start
print *, "prb_stop = ", prb_stop
print *, "p_p_cyc = ", prb_per_cyc
print *, "Uvol = ", Uvol_pos
print *, "--- Restart ---"
print *, "interval = ", restint
print *, "drv_rint = ", drv_rint

#ifdef HAVE_MPI
endif
#endif
RETURN
END SUBROUTINE print_vars

SUBROUTINE alloc_arrays()
USE input_vars
implicit none
integer :: alloc_stat
!
Allocate the main calculation arrays.
alloc_stat = 0
allocate(acs(ni,nj,nACSvar), STAT=alloc_stat)
if (alloc_stat > 0) print *, "Problem allocating memory for acs."
if (streamout .ne. 'non') then
    alloc_stat = 0
    allocate(sumacs(ni,nj,nSACSvar), STAT=alloc_stat)
    if (alloc_stat > 0) print *, "Problem allocating memory for sumacs."
end if
RETURN
END SUBROUTINE alloc_arrays

SUBROUTINE finalize()
!
The subroutine Finalize deallocates and finalizes the computation.

USE input_vars
USE analyt
USE source
USE IO_tools
USE grid
#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
integer :: astat

call fin_analyt()
call fin_source()
call fin_IO()
call fin_grid()
!
! Deallocate the main calculation arrays. 
astat = 0
dallocate(acs, STAT=astat)
if (astat > 0) print *, "Problem deallocating memory for acs."

if (streamout .ne. 'non') then
    astat = 0
    deallocate(sumacs, STAT=astat)
    if (astat > 0) print *, "Problem deallocating memory for sumacs."
end if
#endif HAVE_MPI
call cleanup_mpi()
#endif
RETURN
END SUBROUTINE finalize
!
END MODULE init
D.4.4  acstypes.fpp

MODULE acs_types
!
! This module contains the data structures used for IO operations with
! acoustic variables and file data.
!
USE global
implicit none
!
! Derived type definition for acoustic variables set.
type acsvars_type
  real(DP) :: rho        ! acoustic density (kg/m\(^3\))
  real(DP) :: u          ! velocity in the z-direction (m/s)
  real(DP) :: v          ! velocity in the r-direction (m/s)
  real(DP) :: t          ! acoustic temperature (K)
  real(DP) :: p          ! acoustic pressure (Pa)
end type acsvars_type
!
! Derived type definition for acoustic data file structure.
type acsdata_type
  integer(i4b) :: Xi
  integer(i4b) :: Eta
  real(DP) :: grid_x1
  real(DP) :: grid_x2
  type(acsvars_type) :: vars
end type acsdata_type

END MODULE acs_types
D.4.5 analyt.fpp

MODULE analyt
!===========================================================================
! The analyt module contains the tools needed for calculating analytic
! expressions of acoustic quantities for comparison with the finite
! difference model.
!===========================================================================

USE global
USE input_vars
#ifdef HAVE_MPI
USE mpi_fun
#endif

implicit none
real(DP) :: omega0 ! natural frequency
real(DP) :: delta_V ! viscous boundary layer thickness
real(DP) :: BLTmin ! minimum boundary layer thickness
real(DP) :: k_cyl ! wavenumber
real(DP) :: radph ! phase in radians

real(DP), dimension(:,,:), allocatable :: T_anl, P_anl, U_anl
real(DP) :: P_ampl
integer(i4b) :: Xi_mon, Eta_mon, P_min_tx, P_max_tx, PA_monitor_tx
real(DP), dimension(:,), allocatable :: CSareas
real(DP) :: Uvol_max
real(DP) :: Umax
integer(i4b) :: Uvolx ! Index of volume velocity position
integer(i4b), parameter :: UVOLiou = 300
character(len=11) :: Uvolfile = 'VolVelocity'

CONTAINS
!---------------------------------------------------------------------------
SUBROUTINE init_analyt()
!=========================================================================
! Allocate variables for analytic solutions.
!=========================================================================

USE grid
implicit none
integer(i4b) :: istat
real(DP) :: lambda

radph = theta * Pi / 180.0d0
k_cyl = Pi / L
if (fdrv_type == 'PBC' .or. fdrv_type == 'PRS' .or. &
    fdrv_type == 'lin') then
  omega0 = omega ! driven frequency
  lambda = 2.0d0*Pi*C0/omega0
else
  lambda = 2.0d0 * L
  omega0 = 2.0d0*Pi*C0/lambda ! natural frequency
end if

delta_K = sqrt(2.0d0*k0/(omega0*rho0*Cp))
delta_V = sqrt(2.0d0*mu0/(omega0*rho0))
BLTmin = min(delta_K, delta_V)
P_ampl = 10.0d0**(real(SPL,DP)/20.0d0)*P_ref*sqrt(2.0)

if (analytic) then
  istat = 0
  allocate(T_anl(ni,nj-1), STAT=istat)
  allocate(P_anl(ni,nj-1), STAT=istat)
  allocate(U_anl(ni,nj-1), STAT=istat)
end if

! Set up the environment for monitoring pressure amplitude.
  Xi_mon = 2
  Eta_mon = nint(real(nj_global) / 2.0)
  P_min = 0.0d0
  P_max = 0.0d0
RETURN

END SUBROUTINE init_analyt
!---------------------------------------------------------------------------

SUBROUTINE init_Uvol()
! ======================================================================
! Set up the environment for volume velocity calculation. 
! ======================================================================
USE grid
implicit none

integer(i4b) :: istat
real(DP), dimension(ni_main) :: Uvol_pos_diff
real(DP) :: Uvol_tubepos
integer(i4b) :: Uvolx_gl
logical :: Uvolfile_exists

! Initialize maximum volume velocity.
  Uvol_max = 0.0d0
Umax = 0.0d0
istat = 0
allocate(CSareas(nj), STAT=istat)

! Find the global index, processor, and local index of the volume velocity
! location.
Uvol_tubepos = L * Uvol_pos/100.0d0 ! global position
Uvol_pos_diff = abs(tube_x1(i_main,1) - Uvol_tubepos) ! local distance
Uvolx = minloc(Uvol_pos_diff) ! local index

! Adjust position of Uvol cross section to nearest index.
Uvolx_gl = Uvolx(1) ! Initial global index
Uvol_tubepos = tube_x1(Uvolx_gl,1) ! Initial global position

#define HAVE_MPI
   call find_x1pos_mpi(Uvol_pos_diff, Uvol_proc_id)
#endif

if (taper > 0.0d0) then
   CSareas = 2.0d0*Pi*(Rho1+tube_x1(Uvolx(1),1))**2.0d0 * &
   (1.0d0-dcos(tube_x2(Uvolx(1),:)))
else
   CSareas = Pi*(tube_x2(Uvolx(1),:))**2.0d0
end if

! Initialize maximum volume velocity output file.
#define HAVE_MPI
   call get_real_mpi(Uvol_tubepos, Uvol_proc_id)
#endif
if (myid == MASTER) then
#endif
   inquire(FILE=Uvolfile, EXIST=Uvolfile_exists)

   if (output == 'per' .and. &
   .not. (Uvolfile_exists .and. ic_choice == 9)) then
      open(UVOLiou, FILE=Uvolfile, STATUS='REPLACE', ACTION='WRITE', &
      IOSTAT=istat)

      write(UVOLiou,"(a,f10.4)") "Axial Location: ", Uvol_tubepos
      write(UVOLiou,"(a)") "Period Max Uvol (m^3/s)"

      close(UVOLiou)
   end if
#endif
#endif

RETURN
END SUBROUTINE init_Uvol

!---------------------------------------------------------------------------
SUBROUTINE max_Uvol(timex)
! Track the maximum absolute value of volume velocity calculated at a
! specified position along the axis of the tube. Report the maximum
! for each period of oscillation.
!
! Parallel impl: All processors compute Uvol for the nearest
! location. Only the value from the subdomain containing the global
! nearest index is printed.
!
USE grid
implicit none
integer(i4b), intent(IN) :: timex
real(DP) :: Uvol
real(DP), dimension(nj) :: massflux
integer(i4b) :: cyclex
integer(i4b) :: istat !, i

! Compute volume velocity at the index nearest the specified axial location.
! Calculate mass flux.
massflux = acs(Uvolx(1),:,Ux)*(rho0+acs(Uvolx(1),:,RHOx))

! Integrate over circular cross section area and divide by Rho_0.
Uvol = (CSareas(1)*massflux(1) + &
       sum((CSareas(2:nj)-CSareas(1:(nj-1)))*massflux(1:(nj-1)))/2.0d0 + &
       sum((CSareas(2:nj)-CSareas(1:(nj-1)))*massflux(2:nj))/2.0d0) / rho0

! Update Uvol_max
Uvol_max = max(Uvol_max, abs(Uvol))

if ( output == 'per' .and. modulo(timex - txmin, periodx) == 0 ) then
  cyclex = int(real(timex) / real(periodx))
  ! Print out Uvol_max after each period.
  ifdef HAVE_MPI
    call get_real_mpi(Uvol_max, Uvol_proc_id)
  ifdef
  endif
  if (myid == MASTER) then
    open(UVOLiou, FILE=Uvolfile, STATUS='OLD', ACCESS='SEQUENTIAL', &
         POSITION='APPEND', ACTION='WRITE', IOSTAT=istat)
    write(UVOLiou, "(i6,1es12.4)") cyclex, Uvol_max
    close(UVOLiou)
  ifdef HAVE_MPI
endif
SUBROUTINE max_Uvol
!
end if
#endif

! Reset Uvol_max after printing.
Uvol_max = 0.0d0
end if

RETURN

END SUBROUTINE max_Uvol

SUBROUTINE calc_anl(timex)
!
 USE grid
implicit none

integer(i4b), intent(IN) :: timex
integer(i4b) :: BPrel
real(DP) :: wt

wt = timex*deltat*omega0

! Relative breakpoint is the halfway point of the tube length having index
! relative to the local grid.
BPrel = int(real(ni_global)/2.0) - i_offset

P_anl = P_ampl*cos(k_cyl*tube_x1(:,2:nj))*cos(wt - radph)

if (wall_TBC == 'adi') then
  if (BPrel < 1 .and. end2_TBCgl == 'iso') then ! near end2
    T_anl = (P_anl - &
              P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-(L-tube_x1(:,2:nj))/delta_K) * &
              cos(timex*deltat*omega0 - radph - &
                 (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
  else if (BPrel > ni-1 .and. end1_TBCgl == 'iso') then ! near end1
    T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-tube_x1(:,2:nj)/delta_K) * cos(timex*deltat*omega0 - &
                radph - tube_x1(:,2:nj)/delta_K))/(rho0*Cp)
  else ! calculate both
    if (end1_TBCgl == 'iso') then
      T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
                         P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj))* &
                         exp(-tube_x1(1:BPrel,2:nj)/delta_K) * &
                         cos(timex*deltat*omega0 - &
                            radph - tube_x1(1:BPrel,2:nj)/delta_K)) / (rho0*Cp)
    else
      T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
                exp(-(L-tube_x1(:,2:nj))/delta_K) * &
                cos(timex*deltat*omega0 - &
                   radph - (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
    end if
  end if
else if (BPrel < 1 .and. end2_TBCgl == 'iso') then ! near end2
  T_anl = (P_anl - &
            P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
            exp(-(L-tube_x1(:,2:nj))/delta_K) * &
            cos(timex*deltat*omega0 - radph - &
               (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
else if (BPrel > ni-1 .and. end1_TBCgl == 'iso') then ! near end1
  T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
            exp(-tube_x1(:,2:nj)/delta_K) * cos(timex*deltat*omega0 - &
               radph - tube_x1(:,2:nj)/delta_K))/(rho0*Cp)
else ! calculate both
  if (end1_TBCgl == 'iso') then
    T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
                         P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj))* &
                         exp(-tube_x1(1:BPrel,2:nj)/delta_K) * &
                         cos(timex*deltat*omega0 - &
                            radph - tube_x1(1:BPrel,2:nj)/delta_K)) / (rho0*Cp)
  else
    T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-(L-tube_x1(:,2:nj))/delta_K) * &
              cos(timex*deltat*omega0 - &
                 radph - (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
  end if
end if

! Relative breakpoint is the halfway point of the tube length having index
! relative to the local grid.
BPrel = int(real(ni_global)/2.0) - i_offset

P_anl = P_ampl*cos(k_cyl*tube_x1(:,2:nj))*cos(wt - radph)

if (wall_TBC == 'adi') then
  if (BPrel < 1 .and. end2_TBCgl == 'iso') then ! near end2
    T_anl = (P_anl - &
              P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-(L-tube_x1(:,2:nj))/delta_K) * &
              cos(timex*deltat*omega0 - radph - &
                 (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
  else if (BPrel > ni-1 .and. end1_TBCgl == 'iso') then ! near end1
    T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-tube_x1(:,2:nj)/delta_K) * cos(timex*deltat*omega0 - &
                radph - tube_x1(:,2:nj)/delta_K))/(rho0*Cp)
  else ! calculate both
    if (end1_TBCgl == 'iso') then
      T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
                         P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj))* &
                         exp(-tube_x1(1:BPrel,2:nj)/delta_K) * &
                         cos(timex*deltat*omega0 - &
                            radph - tube_x1(1:BPrel,2:nj)/delta_K)) / (rho0*Cp)
  else
    T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-(L-tube_x1(:,2:nj))/delta_K) * &
              cos(timex*deltat*omega0 - &
                 radph - (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
  end if
else if (BPrel < 1 .and. end2_TBCgl == 'iso') then ! near end2
  T_anl = (P_anl - &
            P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
            exp(-(L-tube_x1(:,2:nj))/delta_K) * &
            cos(timex*deltat*omega0 - radph - &
               (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
else if (BPrel > ni-1 .and. end1_TBCgl == 'iso') then ! near end1
  T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
            exp(-tube_x1(:,2:nj)/delta_K) * cos(timex*deltat*omega0 - &
              radph - tube_x1(:,2:nj)/delta_K))/(rho0*Cp)
else ! calculate both
  if (end1_TBCgl == 'iso') then
    T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
                         P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj))* &
                         exp(-tube_x1(1:BPrel,2:nj)/delta_K) * &
                         cos(timex*deltat*omega0 - &
                            radph - tube_x1(1:BPrel,2:nj)/delta_K)) / (rho0*Cp)
  else
    T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
              exp(-(L-tube_x1(:,2:nj))/delta_K) * &
              cos(timex*deltat*omega0 - &
                 radph - (L-tube_x1(:,2:nj))/delta_K)) / (rho0*Cp)
  end if
end if
cos(timex*deltat*omega0 - radph - &
tube_x1(1:BPrel,2:nj)/delta_K)}/(rho0*Cp)
end if
if (end2_TBCgl == 'iso') then
  T_anl((BPrel+1):ni,:) = (P_anl((BPrel+1):ni,:) - &
P_ampl*cos(k_cyl*tube_x1(BPrel+1:ni,2:nj)) * &
exp(-L-tube_x1((BPrel+1):ni,2:nj))/delta_K) * &
cos(timex*deltat*omega0 - radph - &
(L-tube_x1((BPrel+1):ni,2:nj))/delta_K)) / (rho0*Cp)
end if
else ! wall_TBC == 'iso'
if (BPrel < 1 .and. end2_TBCgl == 'adi') then ! near end2
  T_anl = (P_anl - &
P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
exp(-(radius-tube_x2(:,2:nj))/delta_K) * &
cos(timex*deltat*omega0 - radph - &
(radius-tube_x2(:,2:nj))/delta_K)) / (rho0*Cp)
else if (BPrel > ni-1 .and. end1_TBCgl == 'adi') then ! near end1
  T_anl = (P_anl - P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
exp(-radius-tube_x2(:,2:nj))/delta_K) * &
cos(timex*deltat*omega0 - radph - (radius-tube_x2(:,2:nj)) / &
delta_K))/(rho0*Cp)
else ! calculate both
  if (end1_TBCgl == 'adi') then
    T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj)))* &
exp(-radius-tube_x2(1:BPrel,2:nj))/delta_K) * &
cos(timex*deltat*omega0 - radph - &
(radius-tube_x2(1:BPrel,2:nj))/delta_K))/(rho0*Cp)
  end if
  if (end2_TBCgl == 'adi') then
    T_anl(BPrel+1:ni,:) = (P_anl((BPrel+1):ni,:) - &
P_ampl*cos(k_cyl*tube_x1(BPrel+1:ni,2:nj)) * &
exp(-( radius-tube_x2((BPrel+1):ni,2:nj))/delta_K) * &
cos(timex*deltat*omega0 - radph - &
(radius-tube_x2((BPrel+1):ni,2:nj))/delta_K)) / (rho0*Cp)
  end if
end if
if (BPrel < 1 .and. end2_TBCgl == 'iso') then ! near end2  TAG
  T_anl = (P_anl - &
P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
(exp(-L-tube_x1(:,2:nj))/delta_K) * &
cos(wt-radph-(L-tube_x1(:,2:nj))/delta_K) + &
exp(-radius-tube_x2(:,2:nj))/delta_K) * &
cos(wt-radph-(radius-tube_x2(:,2:nj))/delta_K) - &
exp(-L-tube_x1(:,2:nj)+radius - &
tube_x2(:,2:nj))/ delta_K) * &
cos(wt-radph-(L-tube_x1(:,2:nj) + &
radius-tube_x2(:,2:nj))/delta_K))/(rho0*Cp)
else if (BPrel > ni-1 .and. end1_TBCgl == 'iso') then ! near end1
  T_anl = (P_anl - &
  P_ampl*cos(k_cyl*tube_x1(:,2:nj)) * &
  (exp(-tube_x1(:,2:nj)/delta_K) * &
   cos(wt-radph-tube_x1(:,2:nj)/delta_K) + &
  exp(-(radius-tube_x2(:,2:nj))/delta_K) * &
   cos(wt-radph-(radius-tube_x2(:,2:nj))/delta_K) - &
  exp(-(tube_x1(:,2:nj)+radius-tube_x2(:,2:nj)) / &
   delta_K) * cos(wt-radph-(tube_x1(:,2:nj) + &
   radius-tube_x2(:,2:nj))/delta_K)))/(rho0*Cp)
else
  if (end1_TBCgl == 'iso') then
    T_anl(1:BPrel,:) = (P_anl(1:BPrel,:) - &
    P_ampl*cos(k_cyl*tube_x1(1:BPrel,2:nj)) * &
    (exp(-tube_x1(1:BPrel,2:nj)/delta_K) * &
     cos(wt-radph-tube_x1(1:BPrel,2:nj)/delta_K) + &
    exp(-(radius-tube_x2(1:BPrel,2:nj))/delta_K) * &
    cos(wt-radph-(radius-tube_x2(1:BPrel,2:nj))/delta_K) - &
    exp(-(tube_x1(1:BPrel,2:nj)+radius-tube_x2(1:BPrel,2:nj)) / &
     delta_K) * cos(wt-radph-(tube_x1(1:BPrel,2:nj) + &
     radius-tube_x2(1:BPrel,2:nj))/delta_K)))/(rho0*Cp)
  end if
  if (end2_TBCgl == 'iso') then
    T_anl((BPrel+1):ni,:) = (P_anl((BPrel+1):ni,:) - &
    P_ampl*cos(k_cyl*tube_x1((BPrel+1):ni,2:nj)) * &
    (exp(-(L-tube_x1((BPrel+1):ni,2:nj))/delta_K) * &
     cos(wt-radph-(L-tube_x1((BPrel+1):ni,2:nj))/delta_K) + &
    exp(-(radius-tube_x2((BPrel+1):ni,2:nj))/delta_K) * &
    cos(wt-radph-(radius-tube_x2((BPrel+1):ni,2:nj))/delta_K) - &
    exp(-(L-tube_x1((BPrel+1):ni,2:nj)+radius - &
     tube_x2((BPrel+1):ni,2:nj)) / delta_K) * &
    cos(wt-radph-(L-tube_x1((BPrel+1):ni,2:nj) + &
    radius-tube_x2((BPrel+1):ni,2:nj))/delta_K)))/(rho0*Cp)
  end if
end if

end if

if (wall_VBC == 'ns') then
  U_anl = P_ampl*sin(k_cyl*tube_x1(:,2:nj)) * &
  (sin(wt-radph) - exp(-(radius-tube_x2(:,2:nj))/delta_V) * &
   sin(wt-radph-(radius-tube_x2(:,2:nj))/delta_V))/(C0*rho0)
end if

RETURN

END SUBROUTINE calc_anl

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE fin_analyt()
USE input_vars

implicit none

integer :: istat

if (analytic) then
   istat = 0
   deallocate(T_anl, STAT=istat)
   deallocate(P_anl, STAT=istat)
   deallocate(U_anl, STAT=istat)
end if

istat = 0
deallocate(CSareas, STAT=istat)

RETURN

END SUBROUTINE fin_analyt
D.4.6   boundary.fpp

MODULE boundary
!===========================================================================
! This module contains the subroutines that handle boundary conditions
! for the finite difference approximation.
!===========================================================================

USE global
USE input_vars
USE FDiff_vars
USE grid
USE source

implicit none

interface Neumann_BCxi
  module procedure Neumann_BC_xi1, Neumann_BC_xi2
end interface
interface Neumann_BCeta
  module procedure Neumann_BC_eta1, Neumann_BC_eta2
end interface
interface Neumann_BCeta_offset
  module procedure NeumannBC_offset_eta1, NeumannBC_offset_eta2
end interface
interface Dirichlet_BCeta_offset
  module procedure DirichletBC_offset_eta1, DirichletBC_offset_eta2
end interface

interface Neumann4_BCxi
  module procedure Neumann4_BC_xi1, Neumann4_BC_xi2
end interface
interface Neumann4_BCeta
  module procedure Neumann4_BC_eta1, Neumann4_BC_eta2
end interface

integer(i4b), dimension(3) :: border_end1, border_end2
integer(i4b), dimension(3) :: border_ctr, border_wall
integer(i4b), dimension(5) :: border4_end1, border4_end2
integer(i4b), dimension(5) :: border4_ctr, border4_wall
logical :: aperiodic = .false.

CONTAINS
!---------------------------------------------------------------------------
SUBROUTINE init_bounds()
! ========================================================================
! | Initialize boundary variables. | |
! ========================================================================

implicit none
border_end1 = (/ 1, 2, 3 /
border_end2 = (/ ni, ni-1, ni-2 /)
border_ctr = (/ 1, 2, 3 /)
border_wall = (/ nj, nj-1, nj-2 /)

border4_end1 = (/ 1, 2, 3, 4, 5 /
border4_end2 = (/ ni, ni-1, ni-2, ni-3, ni-4 /)
border4_ctr = (/ 1, 2, 3, 4, 5/) 
border4_wall = (/ nj, nj-1, nj-2, nj-3, nj-4 /)

RETURN

END SUBROUTINE init_bounds

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BC_xi(BC_arr)
! ========================================================================
! | This subroutine sets the boundary conditions on the Xi-min and Xi-max |
! | sides of the Wb or Wbp arrays. The extents of the domain are       |
! | ni and nj. The third dimension is either 5 or 6.                |
! | The extra two indices come from the boundaries at i=1, i=ni.     |
! ========================================================================

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
Real(DP), dimension(:,:,:), intent(INOUT) :: BC_arr

#ifdef HAVE_MPI
! Local arrays.
Real(DP), dimension(nj,size(BC_arr,3)) :: recv_end1
Real(DP), dimension(nj,size(BC_arr,3)) :: recv_end2
Real(DP), dimension(nj,size(BC_arr,3)) :: send_end1
Real(DP), dimension(nj,size(BC_arr,3)) :: send_end2
#endif

!!** NONLINEAR with LOW-ORDER DISSIPATION **
!! End_1 -- where z, xi, and i are minimum
select case (end1_BC)
case ('rig')
! Rigid boundary
BC_arr(1,j_main,Ux) = 0.0d0
endif

if (end1_VBC == 'sl') then ! slip
call Neumann_BCxi(BC_arr, border_end1, j_main, Vx)
else ! no slip
BC_arr(1,j_main,Vx) = 0.0d0
endif
end if

if (end1_TBC == 'src') then
    BC_arr(1,:,Px) = Pressure_src
    BC_arr(1,:,Tx) = Temperature_src(1,:)
else if (end1_TBC == 'iso') then
    BC_arr(1,j_main,Tx) = 0.0d0
else ! Adiabatic
    call Neumann_BCxi(BC_arr, border_end1, j_main, Tx)
end if
call Neumann4_BCxi(BC_arr, border4_end1, j_main, Px)

! Corners:
! Rigid wall
BC_arr(1,nj,Ux) = 0.0d0
BC_arr(1,nj,Vx) = 0.0d0
if (wall_TBC == 'iso') then
    BC_arr(1,nj,Tx) = 0.0d0
else ! Adiabatic
    call Neumann_BCeta(BC_arr, 1, border_wall, Tx)
end if
call Neumann4_BCeta(BC_arr, 1, border4_wall, Px)

! Symmetric center line
BC_arr(1,1,Ux) = 0.0d0
if (end1_VBC == 'sl') then ! slip
    call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Vx)
else ! no slip
    BC_arr(1,1,Vx) = 0.0d0
end if
call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Px)

case ('pis')
if (fdrv_type == 'PBC') then
! Piston boundary
    BC_arr(1,:,Ux) = Velocity_src * Jac(1,:)
else ! Pressure boundary-layer source and others = same as rigid
    BC_arr(1,j_main,Ux) = 0.0d0
end if

if (end1_VBC == 'sl') then ! slip
    call Neumann_BCxi(BC_arr, border_end1, j_main, Vx)
else ! no slip
    BC_arr(1,j_main,Vx) = 0.0d0
end if

if (end1_TBC == 'iso') then
    BC_arr(1,j_main,Tx) = 0.0d0
else ! Adiabatic
    call Neumann_BCxi(BC_arr, border_end1, j_main, Tx)
end if
  call Neumann_BCx1(BC_arr, border_end1, j_main, Px)

  ! Corners:
  if (fdrv_type == 'PRS') then ! Pressure source treat same as rigid.
    BC_arr(1,nj,Ux) = 0.0d0 ! Rigid wall
    BC_arr(1,1,Ux) = 0.0d0 ! Symmetric center line
  end if

  ! Rigid wall
  BC_arr(1,nj,Vx) = 0.0d0
  if (wall_TBC == 'iso') then
    BC_arr(1,nj,Tx) = 0.0d0
  else ! Adiabatic
    call Neumann_BCeta(BC_arr, 1, border_wall, Tx)
  end if
  call Neumann_BCeta(BC_arr, 1, border_wall, Px)

  ! Symmetric center line
  BC_arr(1,1,Vx) = 0.0d0
  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Tx)
  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Px)

  case ('stp')
  ! Piston boundary starting pulse
  BC_arr(1,:,Ux) = Velocity_SP * Jac(1,:)

  if (end1_VBC == 'sl') then ! slip
    call Neumann_BCx1(BC_arr, border_end1, j_main, Vx)
  else ! no slip
    BC_arr(1,j_main,Vx) = 0.0d0
  end if

  if (end1_TBC == 'iso') then
    BC_arr(1,j_main,Tx) = 0.0d0
  else ! Adiabatic
    call Neumann_BCx1(BC_arr, border_end1, j_main, Tx)
  end if
  call Neumann_BCx1(BC_arr, border_end1, j_main, Px)

  ! Corners:
  ! Rigid wall
  BC_arr(1,nj,Vx) = 0.0d0
  if (wall_TBC == 'iso') then
    BC_arr(1,nj,Tx) = 0.0d0
  else ! Adiabatic
    call Neumann_BCeta(BC_arr, 1, border_wall, Tx)
  end if
  call Neumann_BCeta(BC_arr, 1, border_wall, Px)

  ! Symmetric center line
BC_arr(1,1,Vx) = 0.0d0
  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Px)

end select

#ifdef MAKE_LIN
  ! Density -- State equation
  BC_arr(1,:,RHOx) = (BC_arr(1,:,Px) / Rgas - Rho0*BC_arr(1,:,Tx)) / T0
#else
  ! Density -- State equation
  acs(1,:,Tx) = BC_arr(1,:,Tx) * iJac(1,:)
  BC_arr(1,:,RHOx) = (BC_arr(1,:,Px) / Rgas - Rho0*BC_arr(1,:,Tx)) / (T0 + acs(1,:,Tx))
#endif

!! End_2 -- where Z, Xi, and i are maximum
select case (end2_BC)

case ('rig')
  ! Rigid boundary, no-slip velocity, isothermal
  BC_arr(ni,j_main,Ux) = 0.0d0
  if (end2_VBC == 'sl') then ! slip
    call Neumann_BCxi(BC_arr, border_end2, j_main, Vx)
  else ! no slip
    BC_arr(ni,j_main,Vx) = 0.0d0
  end if

if (end2_TBC == 'src') then
  BC_arr(ni,:,Px) = Pressure_src
  BC_arr(ni,:,Tx) = Temperature_src(ni,:)
else if (end2_TBC == 'iso') then
  BC_arr(ni,j_main,Tx) = 0.0d0
else ! Adiabatic
  call Neumann_BCxi(BC_arr, border_end2, j_main, Tx)
end if
  call Neumann4_BCxi(BC_arr, border4_end2, j_main, Px)

! Corners:
!   Rigid wall
  BC_arr(ni,nj,Ux) = 0.0d0
  BC_arr(ni,nj,Vx) = 0.0d0
  if (wall_TBC == 'iso') then
    BC_arr(ni,nj,Tx) = 0.0d0
  else ! Adiabatic
    call Neumann_BCeta(BC_arr, ni, border_wall, Tx)
  end if
  call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

! Symmetric center line
BC_arr(ni,1,Ux) = 0.0d0
if (end2_VBC == 'sl') then  ! slip
    call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Vx)
else  ! no slip
    BC_arr(ni,1,Vx) = 0.0d0
end if
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Px)

case ('pis')
    if (fdrv_type == 'PBC') then
        ! Rigid boundary, no-slip velocity, isothermal
        BC_arr(ni,:,Ux) = Velocity_src * Jac(ni,:)
    else  ! Pressure boundary-layer source and others = same as rigid
        BC_arr(ni,j_main,Ux) = 0.0d0
    end if

    if (end2_VBC == 'sl') then  ! slip
        call Neumann_BCxi(BC_arr, border_end2, j_main, Vx)
    else  ! no slip
        BC_arr(ni,j_main,Vx) = 0.0d0
    end if

    if (end2_TBC == 'iso') then
        BC_arr(ni,j_main,Tx) = 0.0d0
    else  ! Adiabatic
        call Neumann_BCxi(BC_arr, border_end2, j_main, Tx)
    end if

! fourth order

BC_arr(ni,j_main,Px) = Jac(ni,j_main) * &
(48.0d0*BC_arr(ni-1,j_main,Px)*iJac(ni-1,j_main) - &
  36.0d0*BC_arr(ni-2,j_main,Px)*iJac(ni-2,j_main) + &
  16.0d0*BC_arr(ni-3,j_main,Px)*iJac(ni-3,j_main) - &
  3.0d0*BC_arr(ni-4,j_main,Px)*iJac(ni-4,j_main) ) / 25.0d0

! Corners:
    if (fdrv_type == 'PRS') then  ! Pressure source treat same as rigid.
        BC_arr(ni,nj,Vx) = 0.0d0 ! Rigid wall
        BC_arr(ni,1,Ux) = 0.0d0 ! Symmetric center line
    end if

    ! Rigid wall
    BC_arr(ni,nj,Vx) = 0.0d0
    if (wall_TBC == 'iso') then
        BC_arr(ni,nj,Tx) = 0.0d0
    else  ! Adiabatic
        call Neumann_BCeta(BC_arr, ni, border_wall, Tx)
    end if

    call Neumann_BCeta(BC_arr, ni, border_wall, Px)
! Symmetric center line
BC_arr(ni,1,Vx) = 0.0d0
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Px)

case ('stp')
! Piston boundary starting pulse.
! Rigid boundary, no-slip velocity, isothermal
BC_arr(ni,:,Ux) = Velocity_SP * Jac(ni,:
if (end2_VBC == 'sl') then ! slip
   call Neumann_BCxi(BC_arr, border_end2, j_main, Vx)
else ! no slip
   BC_arr(ni,j_main,Vx) = 0.0d0
end if
if (end2_TBC == 'iso') then
   BC_arr(ni,j_main,Tx) = 0.0d0
else ! Adiabatic
   call Neumann_BCxi(BC_arr, border_end2, j_main, Tx)
end if
call Neumann_BCxi(BC_arr, border_end2, j_main, Px)

! Corners:
! Rigid wall
BC_arr(ni,nj,Vx) = 0.0d0
if (wall_TBC == 'iso') then
   BC_arr(ni,nj,Tx) = 0.0d0
else ! Adiabatic
   call Neumann_BCeta(BC_arr, ni, border_wall, Tx)
end if
call Neumann_BCeta(BC_arr, ni, border_wall, Px)

! Symmetric center line
BC_arr(ni,1,Vx) = 0.0d0
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Px)

end select

#ifdef MAKE_LIN
! Density -- State equation
BC_arr(ni,:,RHOx) = (BC_arr(ni,:,Px) / Rgas - Rho0*BC_arr(ni,:,Tx)) / T0
#else
! Density -- State equation
acs(ni,:,Tx) = BC_arr(ni,:,Tx) * iJac(ni,:
BC_arr(ni,:,RHOx) = (BC_arr(ni,:,Px) / Rgas - Rho0*BC_arr(ni,:,Tx)) / &
(T0 + acs(ni,:,Tx))
#endif

#ifdef HAVE_MPI
! Use local arrays because dummy arrays are not assignable by subroutines.
if (end1_BC == 'mpi' .and. end2_BC == 'mpi') then
    send_end1 = BC_arr(2,:,:)
    send_end2 = BC_arr(ni-1,:,:)
    call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,:,:) = recv_end1
    BC_arr(ni,:,:) = recv_end2
else if (end1_BC == 'per' .and. end2_BC == 'mpi') then
    send_end1 = BC_arr(3,:,:)
    send_end2 = BC_arr(ni-1,:,:)
    call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,:,:) = recv_end1
    BC_arr(ni,:,:) = recv_end2
    if (aperiodic) then
        BC_arr(1,:,Ux) = -BC_arr(1,:,Ux)
        BC_arr(1,:,Tx) = -BC_arr(1,:,Tx)
        BC_arr(1,:,RHOx) = -BC_arr(1,:,RHOx)
    end if
else if (end1_BC == 'mpi' .and. end2_BC == 'per') then
    send_end1 = BC_arr(2,:,:)
    send_end2 = BC_arr(ni-2,:,:)
    call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,:,:) = recv_end1
    BC_arr(ni,:,:) = recv_end2
    if (aperiodic) then
        BC_arr(ni,:,Ux) = -BC_arr(ni,:,Ux)
        BC_arr(ni,:,Tx) = -BC_arr(ni,:,Tx)
        BC_arr(ni,:,RHOx) = -BC_arr(ni,:,RHOx)
    end if
else if (end2_BC == 'mpi') then
    send_end2 = BC_arr(ni-1,:,:)
    call share_hibound_mpi(send_end2, recv_end2)
    BC_arr(ni,:,:) = recv_end2
else if (end1_BC == 'mpi') then
    send_end1 = BC_arr(2,:,:)
    call share_lobound_mpi(send_end1, recv_end1)
    BC_arr(1,:,:) = recv_end1
end if
#endif
if (end1_BC == 'per' .and. end2_BC == 'per') then
    BC_arr(1,:,:) = BC_arr(ni-2,:,:)
    BC_arr(ni,:,:) = BC_arr(3,:,:)
    if (aperiodic) then
        BC_arr(1,:,Ux) = -BC_arr(1,:,Ux)
        BC_arr(ni,:,Ux) = -BC_arr(ni,:,Ux)
BC_arr(1,:,Tx) = -BC_arr(1,:,Tx)
BC_arr(ni,:,Tx) = -BC_arr(ni,:,Tx)
BC_arr(1,:,RHOx) = -BC_arr(1,:,RHOx)
BC_arr(ni,:,RHOx) = -BC_arr(ni,:,RHOx)
end if
end if
RETURN
END SUBROUTINE BC_xi

!---------------------------------------------------------------------------

SUBROUTINE BC_eta(BC_arr)
! ========================================================================
! | This subroutine sets the boundary conditions on the Eta-min and        |
! | Eta-max sides of the Wb or Wbp arrays. The extents of the domain are  |
! | ni and nj. The third dimension is either 5 or 6.                     |
! | The extra two indices come from the boundaries at j=1, j=ni.          |
! ========================================================================

#elifdef HAVE_MPI
USE mpi_fun
#endif

implicit none
Real(DP), dimension(:,:,:), intent(INOUT) :: BC_arr

#ifdef HAVE_MPI
! Corner point arrays.
Real(DP), dimension(size(BC_arr,3)) :: recv_end1
Real(DP), dimension(size(BC_arr,3)) :: recv_end2
Real(DP), dimension(size(BC_arr,3)) :: send_end1
Real(DP), dimension(size(BC_arr,3)) :: send_end2
#endif

! Centerline -- where r, eta, and j are minimum
! assume symmetric boundary
call Neumann_BCeta_offset(BC_arr, i_main, border_ctr, Ux)
call Dirichlet_BCeta_offset(BC_arr, i_main, border_ctr,Vx)
call Neumann_BCeta_offset(BC_arr, i_main, border_ctr, Tx)
call Neumann_BCeta_offset(BC_arr, i_main, border_ctr, Px)

! Wall -- where R, Eta, and j are maximum
! assume rigid boundary, no-slip velocity U, isothermal
! Slip velocity, rigid wall
if (wall_VBC == 'sl') then ! slip
    call Neumann_BCeta(BC_arr, i_main, border_wall, Ux)
else ! no slip...
BC_arr(i_main,nj,Ux) = 0.0d0
end if

BC_arr(i_main,nj,Vx) = 0.0d0

if (wall_TBC == 'iso') then
    BC_arr(i_main,nj,Tx) = 0.0d0
else ! Adiabatic
    call Neumann_BCeta(BC_arr, i_main, border_wall, Tx)
end if

call Neumann4_BCeta(BC_arr, i_main, border4_wall, Px)

! Corners for both Centerline and Wall:
select case (end1_BC)
case ("rig")
! Rigid End_1
    BC_arr(1,1,Ux) = 0.0d0
    BC_arr(1,nj,Ux) = 0.0d0

    if (end1_VBC == 'sl') then
        call Neumann_BCxi(BC_arr, border_end1, 1, Vx)
    else
        BC_arr(1,1,Vx) = 0.0d0
        end if
    BC_arr(1,nj,Vx) = 0.0d0

    if (end1_TBC == 'iso') then ! Isothermal
        BC_arr(1,1,Tx) = 0.0d0
        BC_arr(1,nj,Tx) = 0.0d0
    else ! Adiabatic
        call Neumann_BCxi(BC_arr, border_end1, 1, Tx)
        call Neumann_BCxi(BC_arr, border_end1, nj, Tx)
    end if

call Neumann4_BCxi(BC_arr, border4_end1, 1, Px)
call Neumann4_BCxi(BC_arr, border4_end1, nj, Px)

case ("pis")
! Piston End_1
if (fdrv_type == 'PRS') then
    BC_arr(1,1,Ux) = 0.0d0
    BC_arr(1,nj,Ux) = 0.0d0
else ! 'PBC'
    BC_arr(1,1,Ux) = Velocity_src(1) * Jac(1,1)
    BC_arr(1,nj,Ux) = Velocity_src(nj) * Jac(1,nj)
end if

    BC_arr(1,1,Vx) = 0.0d0
    BC_arr(1,nj,Vx) = 0.0d0
if (end1_TBC == 'iso') then ! Isothermal
  BC_arr(1,1,Tx) = 0.0d0
  BC_arr(1,nj,Tx) = 0.0d0
else ! Adiabatic
  call Neumann_BCxi(BC_arr, border_end1, 1, Tx)
  call Neumann_BCxi(BC_arr, border_end1, nj, Tx)
end if

call Neumann_BCxi(BC_arr, border_end1, 1, Px)
call Neumann_BCxi(BC_arr, border_end1, nj, Px)

case ("stp")
  ! Piston End_1
  BC_arr(1,1,Ux) = Velocity_SP(1) * Jac(1,1)
  BC_arr(1,nj,Ux) = Velocity_SP(nj) * Jac(1,nj)

  BC_arr(1,1,Vx) = 0.0d0
  BC_arr(1,nj,Vx) = 0.0d0

if (end1_TBC == 'iso') then ! Isothermal
  BC_arr(1,1,Tx) = 0.0d0
  BC_arr(1,nj,Tx) = 0.0d0
else ! Adiabatic
  call Neumann_BCxi(BC_arr, border_end1, 1, Tx)
  call Neumann_BCxi(BC_arr, border_end1, nj, Tx)
end if

call Neumann_BCxi(BC_arr, border_end1, 1, Px)
call Neumann_BCxi(BC_arr, border_end1, nj, Px)

case ("moc")
  ! Nonreflecting End_1
  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Ux)
  if (wall_VBC == 'sl') then
    call Neumann_BCeta(BC_arr, 1, border_wall, Ux)
  else
    BC_arr(1,nj,Ux) = 0.0d0
  end if

  call Dirichlet_BCeta_offset(BC_arr, 1, border_ctr, Vx)
  BC_arr(1,nj,Vx) = 0.0d0

  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Tx)
  if (wall_TBC == 'iso') then
    BC_arr(1,nj,Tx) = 0.0d0
  else ! Adiabatic
    call Neumann_BCeta(BC_arr, 1, border_wall, Tx)
  end if

  call Neumann_BCeta_offset(BC_arr, 1, border_ctr, Px)
call Neumann4_BCeta(BC_arr, 1, border4_wall, Px)
end select

select case (end2_BC)
case ("rig")
! Rigid End_2
   BC_arr(ni,1,Ux) = 0.0d0
   BC_arr(ni,nj,Ux) = 0.0d0

   if (end2_VBC == 'sl') then
      call Neumann_BCxi(BC_arr, border_end2, 1, Vx)
   else
      BC_arr(ni,1,Vx) = 0.0d0
   end if
   BC_arr(ni,nj,Vx) = 0.0d0

   if (end2_TBC == 'iso') then ! Isothermal
      BC_arr(ni,1,Tx) = 0.0d0
      BC_arr(ni,nj,Tx) = 0.0d0
   else ! Adiabatic
      call Neumann_BCxi(BC_arr, border_end2, 1, Tx)
      call Neumann_BCxi(BC_arr, border_end2, nj, Tx)
   end if
   call Neumann4_BCxi(BC_arr, border4_end2, 1, Px)
   call Neumann4_BCxi(BC_arr, border4_end2, nj, Px)

case ("pis")
! Piston End_2
   if (fdrv_type == 'PRS') then
      BC_arr(ni,1,Ux) = 0.0d0
      BC_arr(ni,nj,Ux) = 0.0d0
   else ! 'PBC'
      BC_arr(ni,1,Ux) = Velocity_src(1) * Jac(ni,1)
      BC_arr(ni,nj,Ux) = Velocity_src(nj) * Jac(ni,nj)
   end if
   BC_arr(ni,1,Vx) = 0.0d0
   BC_arr(ni,nj,Vx) = 0.0d0

   if (end2_TBC == 'iso') then ! Isothermal
      BC_arr(ni,1,Tx) = 0.0d0
      BC_arr(ni,nj,Tx) = 0.0d0
   else ! Adiabatic
      call Neumann_BCxi(BC_arr, border_end2, 1, Tx)
      call Neumann_BCxi(BC_arr, border_end2, nj, Tx)
   end if

! fourth order
```fortran
BC_arr(ni,1,Px) = Jac(ni,1) * &
(48.0d0*BC_arr(ni-1,1,Px)*iJac(ni-1,1) - &
36.0d0*BC_arr(ni-2,1,Px)*iJac(ni-2,1) + &
16.0d0*BC_arr(ni-3,1,Px)*iJac(ni-3,1) - &
3.0d0*BC_arr(ni-4,1,Px)*iJac(ni-4,1) ) / 25.0d0
! fourth order
BC_arr(ni,nj,Px) = Jac(ni,nj) * &
(48.0d0*BC_arr(ni-1,nj,Px)*iJac(ni-1,nj) - &
36.0d0*BC_arr(ni-2,nj,Px)*iJac(ni-2,nj) + &
16.0d0*BC_arr(ni-3,nj,Px)*iJac(ni-3,nj) - &
3.0d0*BC_arr(ni-4,nj,Px)*iJac(ni-4,nj) ) / 25.0d0

! case ("stp")
Piston End_2
BC_arr(ni,1,Ux) = Velocity_SP(1) * Jac(ni,1)
BC_arr(ni,nj,Ux) = Velocity_SP(nj) * Jac(ni,nj)
BC_arr(ni,1,Vx) = 0.0d0
BC_arr(ni,nj,Vx) = 0.0d0

if (end2_TBC == 'iso') then ! Isothermal
  BC_arr(ni,1,Tx) = 0.0d0
  BC_arr(ni,nj,Tx) = 0.0d0
else ! Adiabatic
  call Neumann_BCxi(BC_arr, border_end2, 1, Tx)
  call Neumann_BCxi(BC_arr, border_end2, nj, Tx)
end if

! Adiabatic boundary conditions at the end

! case ("moc")
Nonreflecting End_2
call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Ux)
if (wall_VBC == 'sl') then
  call Neumann_BCeta(BC_arr, ni, border_wall, Ux)
else
  BC_arr(ni,nj,Ux) = 0.0d0
end if

call Dirichlet_BCeta_offset(BC_arr, ni, border_ctr, Vx)
BC_arr(ni,nj,Vx) = 0.0d0

call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Tx)
if (wall_TBC == 'iso') then
  BC_arr(ni,nj,Tx) = 0.0d0
else ! Adiabatic
  call Neumann_BCeta(BC_arr, ni, border_wall, Tx)
end if

call Neumann_BCeta_offset(BC_arr, ni, border_ctr, Px)
```

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)

call Neumann4_BCeta(BC_arr, ni, border4_wall, Px)
end if

else if (end1_BC == 'mpi') then
    send_end1 = BC_arr(2,1,:)
call share_lobound_mpi(send_end1, recv_end1)
    BC_arr(1,1,:) = recv_end1

else if (end2_BC == 'mpi') then
    send_end2 = BC_arr(ni-1,1,:)
call share_hibound_mpi(send_end2, recv_end2)
    BC_arr(ni,1,:) = recv_end2
end if

! Wall -- where R, Eta, and j are maximum
if (end1_BC == 'mpi' .and. end2_BC == 'mpi') then
    send_end1 = BC_arr(2,nj,:)
    send_end2 = BC_arr(ni-1,nj,:)
call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,nj,:) = recv_end1
    BC_arr(ni,nj,:) = recv_end2
else if (end1_BC == 'per' .and. end2_BC == 'mpi') then
    send_end1 = BC_arr(3,nj,:)
    send_end2 = BC_arr(ni-1,nj,:)
call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,nj,:) = recv_end1
    BC_arr(ni,nj,:) = recv_end2
    if (aperiodic) then
        BC_arr(1,nj,Ux) = -BC_arr(1,nj,Ux)
        BC_arr(1,nj,Tx) = -BC_arr(1,nj,Tx)
        BC_arr(1,nj,RHOx) = -BC_arr(1,nj,RHOx)
    end if
else if (end1_BC == 'mpi' .and. end2_BC == 'per') then
    send_end1 = BC_arr(2,nj,:)
    send_end2 = BC_arr(ni-2,nj,:)
call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)
    BC_arr(1,nj,:) = recv_end1
    BC_arr(ni,nj,:) = recv_end2
    if (aperiodic) then
        BC_arr(ni,nj,Ux) = -BC_arr(ni,nj,Ux)
        BC_arr(ni,nj,Tx) = -BC_arr(ni,nj,Tx)
        BC_arr(ni,nj,RHOx) = -BC_arr(ni,nj,RHOx)
    end if
else if (end1_BC == 'mpi') then
    send_end1 = BC_arr(2,nj,:)
call share_lobound_mpi(send_end1, recv_end1)
    BC_arr(1,nj,:) = recv_end1
else if (end2_BC == 'mpi') then
    send_end2 = BC_arr(ni-1,nj,:)
call share_hibound_mpi(send_end2, recv_end2)
    BC_arr(ni,nj,:) = recv_end2
end if

#endif

if (end1_BC == 'per' .and. end2_BC == 'per') then
    BC_arr(1,1,:) = BC_arr(ni-2,1,:)
    BC_arr(1,nj,:) = BC_arr(ni-2,nj,:)
    BC_arr(ni,1,:) = BC_arr(3,1,:)
    BC_arr(ni,nj,:) = BC_arr(3,nj,:)
    if (aperiodic) then
        BC_arr(1,1,Ux) = -BC_arr(1,1,Ux)
        BC_arr(1,nj,Ux) = -BC_arr(1,nj,Ux)
        BC_arr(ni,1,Ux) = -BC_arr(ni,1,Ux)
        BC_arr(ni,nj,Ux) = -BC_arr(ni,nj,Ux)
        BC_arr(1,1,Tx) = -BC_arr(1,1,Tx)
        BC_arr(1,nj,Tx) = -BC_arr(1,nj,Tx)
        BC_arr(ni,1,Tx) = -BC_arr(ni,1,Tx)
        BC_arr(ni,nj,Tx) = -BC_arr(ni,nj,Tx)
        BC_arr(1,1,RHOx) = -BC_arr(1,1,RHOx)
        BC_arr(1,nj,RHOx) = -BC_arr(1,nj,RHOx)
        BC_arr(ni,1,RHOx) = -BC_arr(ni,1,RHOx)
        BC_arr(ni,nj,RHOx) = -BC_arr(ni,nj,RHOx)
    end if
end if
RETURN

END SUBROUTINE BC_eta

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE Neumann_BC_xi1(array,xrng,yrng,acsv)
implicit none
Real(DP), dimension(:,:,:), intent(INOUT) :: array
integer(i4b), dimension(3), intent(IN) :: xrng
integer(i4b), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng(1),yrng,acsv) = Jac(xrng(1),yrng) * onethd * &
   ( 4.0d0*array(xrng(2),yrng,acsv)*iJac(xrng(2),yrng) - &
     array(xrng(3),yrng,acsv)*iJac(xrng(3),yrng) )
RETURN
END SUBROUTINE Neumann_BC_xi1

!---------------------------------------------------------------------------

SUBROUTINE Neumann_BC_xi2(array,xrng,yrng,acsv)

implicit none

Real(DP), dimension(::,::,:), intent(INOUT) :: array
integer(i4b), dimension(3), intent(IN) :: xrng
integer(i4b), dimension(nj_main), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng(1),yrng,acsv) = Jac(xrng(1),yrng) * onethd * &
( 4.0d0*array(xrng(2),yrng,acsv)*iJac(xrng(2),yrng) - &
array(xrng(3),yrng,acsv)*iJac(xrng(3),yrng) )

RETURN

END SUBROUTINE Neumann_BC_xi2

!---------------------------------------------------------------------------

SUBROUTINE Neumann4_BC_xi1(array,xrng,yrng,acsv)
! =======================================================================
! | Fourth-order one-sided difference for 1D (point) boundary. |
! =======================================================================

implicit none

Real(DP), dimension(::,::,:), intent(INOUT) :: array
integer(i4b), dimension(5), intent(IN) :: xrng
integer(i4b), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng(1),yrng,acsv) = Jac(xrng(1),yrng) * 0.04d0 * &
(48.0d0*array(xrng(2),yrng,acsv)*iJac(xrng(2),yrng) - &
36.0d0*array(xrng(3),yrng,acsv)*iJac(xrng(3),yrng) + &
16.0d0*array(xrng(4),yrng,acsv)*iJac(xrng(4),yrng) - &
3.0d0*array(xrng(5),yrng,acsv)*iJac(xrng(5),yrng) )

RETURN

END SUBROUTINE Neumann4_BC_xi1

!---------------------------------------------------------------------------

SUBROUTINE Neumann4_BC_xi2(array,xrng,yrng,acsv)
! =======================================================================
! | Fourth-order one-sided difference for 2D (line) boundary. |
! =======================================================================
implicit none

Real(DP), dimension(:, :, :), intent(INOUT) :: array
integer(i4b), dimension(5), intent(IN) :: xrng
integer(i4b), dimension(nj_main), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng(1), yrng, acsv) = Jac(xrng(1), yrng) * 0.04d0 * &
(48.0d0*array(xrng(2), yrng, acsv)*iJac(xrng(2), yrng) - &
36.0d0*array(xrng(3), yrng, acsv)*iJac(xrng(3), yrng) + &
16.0d0*array(xrng(4), yrng, acsv)*iJac(xrng(4), yrng) - &
3.0d0*array(xrng(5), yrng, acsv)*iJac(xrng(5), yrng) )

RETURN

END SUBROUTINE Neumann4_BC_xi2

SUBROUTINE Neumann_BC_eta1(array, xrng, yrng, acsv)

implicit none

Real(DP), dimension(:, :, :), intent(INOUT) :: array
integer(i4b), intent(IN) :: xrng
integer(i4b), dimension(3), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng, yrng(1), acsv) = Jac(xrng, yrng(1)) * onethd * &
(4.0d0*array(xrng, yrng(2), acsv)*iJac(xrng, yrng(2)) - &
array(xrng, yrng(3), acsv)*iJac(xrng, yrng(3)) )

RETURN

END SUBROUTINE Neumann_BC_eta1

SUBROUTINE Neumann_BC_eta2(array, xrng, yrng, acsv)

implicit none

Real(DP), dimension(:, :, :), intent(INOUT) :: array
integer(i4b), dimension(ni_main), intent(IN) :: xrng
integer(i4b), dimension(3), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng, yrng(1), acsv) = Jac(xrng, yrng(1)) * onethd * &
(4.0d0*array(xrng, yrng(2), acsv)*iJac(xrng, yrng(2)) - &
array(xrng, yrng(3), acsv)*iJac(xrng, yrng(3)) )
RETURN

END SUBROUTINE Neumann_BC_eta2

!---------------------------------------------------------------

SUBROUTINE Neumann4_BC_eta1(array,xrng,yrng,acsv)
! | Fourth-order one-sided difference for 1D (point) boundary. |
! ! ---------------------------------------------------------------

implicit none

Real(DP), dimension(:, :, :), intent(INOUT) :: array
integer(i4b), intent(IN) :: xrng
integer(i4b), dimension(5), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

dimensions(xrng,yrng(1),acsv) = Jac(xrng,yrng(1)) * 0.04d0 * &
(48.0d0*array(xrng,yrng(2),acsv)*iJac(xrng,yrng(2)) - &
36.0d0*array(xrng,yrng(3),acsv)*iJac(xrng,yrng(3)) + &
16.0d0*array(xrng,yrng(4),acsv)*iJac(xrng,yrng(4)) - &
3.0d0*array(xrng,yrng(5),acsv)*iJac(xrng,yrng(5)) )

RETURN

END SUBROUTINE Neumann4_BC_eta1

!---------------------------------------------------------------

SUBROUTINE Neumann4_BC_eta2(array,xrng,yrng,acsv)
! | Fourth-order one-sided difference for 2D (line) boundary. |
! ! ---------------------------------------------------------------

implicit none

Real(DP), dimension(:, :, :), intent(INOUT) :: array
integer(i4b), dimension(ni_main), intent(IN) :: xrng
integer(i4b), dimension(5), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng,yrng(1),acsv) = Jac(xrng,yrng(1)) * 0.04d0 * &
(48.0d0*array(xrng,yrng(2),acsv)*iJac(xrng,yrng(2)) - &
36.0d0*array(xrng,yrng(3),acsv)*iJac(xrng,yrng(3)) + &
16.0d0*array(xrng,yrng(4),acsv)*iJac(xrng,yrng(4)) - &
3.0d0*array(xrng,yrng(5),acsv)*iJac(xrng,yrng(5)) )

RETURN
END SUBROUTINE Neumann4_BC_eta2
-------------------------------------------------------------------
SUBROUTINE NeumannBC_offset_eta1(array, xng, yrng, acsv)

   implicit none

   real(DP), dimension(:,,:), intent(INOUT) :: array
   integer(i4b), intent(IN) :: xng
   integer(i4b), dimension(3), intent(IN) :: yrng
   integer(i4b), intent(IN) :: acsv

   array(xng,yrng(1),acsv) = Jac(xng,yrng(1)) * 0.5d0 * &
                        ( 3.0d0*array(xng,yrng(2),acsv)*iJac(xng,yrng(2)) - &
                          array(xng,yrng(3),acsv)*iJac(xng,yrng(3)) )

RETURN
END SUBROUTINE NeumannBC_offset_eta1
-------------------------------------------------------------------
SUBROUTINE NeumannBC_offset_eta2(array, xng, yrng, acsv)

   implicit none

   real(DP), dimension(:,,:), intent(INOUT) :: array
   integer(i4b), dimension(ni_main), intent(IN) :: xng
   integer(i4b), dimension(3), intent(IN) :: yrng
   integer(i4b), intent(IN) :: acsv

   array(xng,yrng(1),acsv) = Jac(xng,yrng(1)) * 0.5d0 * &
                        ( 3.0d0*array(xng,yrng(2),acsv)*iJac(xng,yrng(2)) - &
                          array(xng,yrng(3),acsv)*iJac(xng,yrng(3)) )

RETURN
END SUBROUTINE NeumannBC_offset_eta2
-------------------------------------------------------------------
SUBROUTINE DirichletBC_offset_eta1(array, xng, yrng, acsv)

   implicit none

   real(DP), dimension(:,,:), intent(INOUT) :: array
   integer(i4b), intent(IN) :: xng
   integer(i4b), dimension(3), intent(IN) :: yrng
   integer(i4b), intent(IN) :: acsv
array(xrng,yrng(1),acsv) = Jac(xrng,yrng(1)) * & 
( twothds*array(xrng,yrng(2),acsv)*iJac(xrng,yrng(2)) - & 
0.2d0*array(xrng,yrng(3),acsv)*iJac(xrng,yrng(3)) )

RETURN

END SUBROUTINE DirichletBC_offset_eta1

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------

SUBROUTINE DirichletBC_offset_eta2(array, xrng, yrng, acsv)
implicit none
real(DP), dimension(:,:,:), intent(INOUT) :: array
integer(i4b), dimension(ni_main), intent(IN) :: xrng
integer(i4b), dimension(3), intent(IN) :: yrng
integer(i4b), intent(IN) :: acsv

array(xrng,yrng(1),acsv) = Jac(xrng,yrng(1)) * & 
( twothds*array(xrng,yrng(2),acsv)*iJac(xrng,yrng(2)) - & 
0.2d0*array(xrng,yrng(3),acsv)*iJac(xrng,yrng(3)) )

RETURN

END SUBROUTINE DirichletBC_offset_eta2

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------

SUBROUTINE BCgr_xi(u_x1, u_x2, v_x1, v_x2, t_x1)
! =========================================================================
! | This subroutine sets the boundary conditions for the velocity |
! | gradients u_x1, u_x2, v_x1, v_x2, and the temperature gradient t_x1 |
! | in the x1 direction. |
! | Add boundary conditions in the x2-direction for u_x2 and v_x2. |
! =========================================================================

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
! Local arrays.
Real(DP), dimension(nj_main,5) :: send_end1, send_end2
Real(DP), dimension(nj_main,5) :: recv_end1, recv_end2
#endif

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:,), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:), intent(INOUT) :: t_x1

#ifdef HAVE_MPI
USE mpi_fun
#endif
implicit none
real(DP), dimension(:,:), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:,), intent(INOUT) :: u_x1, u_x2, v_x1, v_x2
real(DP), dimension(:,:)
** End_1 (min x1,xi,i) boundary condition
select case (end1_BC)

! Rigid boundary:
case ("rig")

! Axial velocity -- Dirichlet BC
  u_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Ux) + &
                      4.0d0*acs(2,j_main,Ux) - acs(3,j_main,Ux)) * Xi_x1(1,j_main)
  u_x2(1,j_main) = 0.0d0

! Radial velocity
  if (end1_VBC == 'sl') then ! Slip velocity in radial direction
    ! Neumann BC: Normal gradient of radial velocity is zero at the end.
    v_x1(1,j_main) = 0.0d0
  else ! No slip
    ! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
    v_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Vx) + &
                       4.0d0*acs(2,j_main,Vx) - acs(3,j_main,Vx)) * Xi_x1(1,j_main)
  end if

! Non-zero gradient along the boundary
  v_x2(1,j_main) = 0.5d0 * (acs(1,j_p_one,Vx) - &
                       acs(1,j_m_one,Vx)) * Eta_x2(1,j_main)

! Temperature
  if (end1_TBC == 'iso') then ! Isothermal -- Dirichlet BC
    t_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Tx) + &
                      4.0d0*acs(2,j_main,Tx) - acs(3,j_main,Tx)) * Xi_x1(1,j_main)
  else ! Adiabatic -- Neumann BC
    t_x1(1,j_main) = 0.0d0
  end if

! Piston source:
case ("pis")

! Axial velocity -- Dirichlet BC
  u_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Ux) + &
                      4.0d0*acs(2,j_main,Ux) - acs(3,j_main,Ux)) * Xi_x1(1,j_main)

! Allow radial variation in axial velocity. 12jun09
  u_x2(1,j_main) = 0.5d0 * (acs(1,j_p_one,Ux) - &
                      acs(1,j_m_one,Ux)) * Eta_x2(1,j_main)
! Radial velocity
if (end1_VBC == 'sl') then ! Slip velocity in radial direction
  ! Neumann BC: Normal gradient of radial velocity is zero at the end.
  v_x1(1,j_main) = 0.0d0

  ! Non-zero gradient along the boundary
  v_x2(1,j_main) = 0.5d0 * (acs(1,j_p_one,Vx) - &
    acs(1,j_m_one,Vx)) * Eta_x2(1,j_main)
else ! No slip
  ! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
  v_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Vx) + &
    4.0d0*acs(2,j_main,Vx) - acs(3,j_main,Vx)) * Xi_x1(1,j_main)

  ! Zero radial velocity -> zero gradient along the boundary
  v_x2(1,j_main) = 0.0d0
end if

! Temperature
if (end1_TBC == 'iso') then
  ! Isothermal -- Dirichlet BC
  t_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Tx) + &
    4.0d0*acs(2,j_main,Tx) - acs(3,j_main,Tx)) * Xi_x1(1,j_main)
else
  ! Adiabatic -- Neumann BC
  t_x1(1,j_main) = 0.0d0
end if

! Starting Pulse Piston source:
case ("stp")
  ! Axial velocity -- Dirichlet BC
  u_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Ux) + &
    4.0d0*acs(2,j_main,Ux) - acs(3,j_main,Ux)) * Xi_x1(1,j_main)

  ! Allow radial variation in axial velocity. 12jun09
  u_x2(1,j_main) = 0.5d0 * (acs(1,j_p_one,Ux) - &
    acs(1,j_m_one,Ux)) * Eta_x2(1,j_main)

  ! Radial velocity
  if (end1_VBC == 'sl') then ! Slip velocity in radial direction
    ! Neumann BC: Normal gradient of radial velocity is zero at the end.
    v_x1(1,j_main) = 0.0d0

    ! Non-zero gradient along the boundary
    v_x2(1,j_main) = 0.5d0 * (acs(1,j_p_one,Vx) - &
      acs(1,j_m_one,Vx)) * Eta_x2(1,j_main)
  else ! No slip
    ! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
    v_x1(1,j_main) = 0.5d0 * (-3.0d0*acs(1,j_main,Vx) + &
      4.0d0*acs(2,j_main,Vx) - acs(3,j_main,Vx)) * Xi_x1(1,j_main)
  end if
186

\[ 4.0d0 \cdot \text{acs}(2,j_{\text{main}},Vx) - \text{acs}(3,j_{\text{main}},Vx) \] \* \( \text{Xi}_{x1}(1,j_{\text{main}}) \)

! Zero radial velocity -> zero gradient along the boundary
\[ v_{x2}(1,j_{\text{main}}) = 0.0d0 \]

end if

! Temperature
if (end1_TBC == 'iso') then
! Isothermal -- Dirichlet BC
\[ t_{x1}(1,j_{\text{main}}) = 0.5d0 \cdot (-3.0d0 \cdot \text{acs}(1,j_{\text{main}},Tx) + &\]
\[ 4.0d0 \cdot \text{acs}(2,j_{\text{main}},Tx) - \text{acs}(3,j_{\text{main}},Tx)) \] \* \( \text{Xi}_{x1}(1,j_{\text{main}}) \)
else
! Adiabatic -- Neumann BC
\[ t_{x1}(1,j_{\text{main}}) = 0.0d0 \]
end if

end select

!!! End_2 (max Z,Xi,i) boundary condition
select case (end2_BC)

! Rigid boundary:
case ("rig")

! Axial velocity -- Dirichlet BC
\[ u_{x1}(ni,j_{\text{main}}) = -0.5d0 \cdot (-3.0d0 \cdot \text{acs}(ni,j_{\text{main}},Ux) + &\]
\[ 4.0d0 \cdot \text{acs}(ni-1,j_{\text{main}},Ux) - \text{acs}(ni-2,j_{\text{main}},Ux)) \] \* \( \text{Xi}_{x1}(ni,j_{\text{main}}) \)
\[ u_{x2}(ni,j_{\text{main}}) = 0.0d0 \]

! Radial velocity
if (end2_VBC == 'sl') then ! Slip velocity in radial direction
! Neumann BC: Normal gradient of radial velocity is zero at the end.
\[ v_{x1}(ni,j_{\text{main}}) = 0.0d0 \]
! Non-zero gradient along the boundary
\[ v_{x2}(ni,j_{\text{main}}) = 0.5d0 \cdot (\text{acs}(ni,j_{\text{p_one}},Vx) - &\]
\[ \text{acs}(ni,j_{\text{m_one}},Vx)) \] \* \( \text{Eta}_{x2}(ni,j_{\text{main}}) \)
else ! No slip
! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
\[ v_{x1}(ni,j_{\text{main}}) = -0.5d0 \cdot (-3.0d0 \cdot \text{acs}(ni,j_{\text{main}},Vx) + &\]
\[ 4.0d0 \cdot \text{acs}(ni-1,j_{\text{main}},Vx) - \text{acs}(ni-2,j_{\text{main}},Vx)) \] \* \( \text{Xi}_{x1}(ni,j_{\text{main}}) \)

! Zero radial velocity -> zero gradient along the boundary
\[ v_{x2}(ni,j_{\text{main}}) = 0.0d0 \]
end if

! Temperature
if (end1_TBC == 'iso') then
! Isothermal -- Dirichlet BC
  t_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Tx) + & 
    4.0d0*acs(ni-1,j_main,Tx) - acs(ni-2,j_main,Tx)) * Xi_x1(ni,j_main)
else
  ! Adiabatic -- Neumann BC
  t_x1(ni,j_main) = 0.0d0
end if

! Piston source:
case ("pis") ! same as rigid except that the xi-dir
  ! velocity gradient is one-sided FDiff.

! Axial velocity -- Dirichlet BC
  u_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Ux) + & 
    4.0d0*acs(ni-1,j_main,Ux) - acs(ni-2,j_main,Ux)) * Xi_x1(ni,j_main)
  ! Allow radial variation in axial velocity.
  u_x2(ni,j_main) = 0.5d0 * (acs(ni,j_p_one,Ux) - & 
    acs(ni,j_m_one,Ux)) * Eta_x2(ni,j_main)

! Radial velocity
  if (end2_VBC == 'sl') then ! Slip velocity in radial direction
    ! Neumann BC: Normal gradient of radial velocity is zero at the end.
    v_x1(ni,j_main) = 0.0d0
  else ! No slip
    ! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
    v_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Vx) + & 
      4.0d0*acs(ni-1,j_main,Vx) - acs(ni-2,j_main,Vx)) * Xi_x1(ni,j_main)
  end if
  ! Zero radial velocity -> zero gradient along the boundary
  v_x2(ni,j_main) = 0.0d0

end if

! Temperature
  if (end1_TBC == 'iso') then
    ! Isothermal -- Dirichlet BC
    t_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Tx) + & 
      4.0d0*acs(ni-1,j_main,Tx) - acs(ni-2,j_main,Tx)) * Xi_x1(ni,j_main)
  else
    ! Adiabatic -- Neumann BC
    t_x1(ni,j_main) = 0.0d0
  end if

! Starting Pulse Piston source:
case ("stp") ! same as rigid except that the xi-dir
  ! velocity gradient is one-sided FDiff.
! Axial velocity -- Dirichlet BC
u_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Ux) +
4.0d0*acs(ni-1,j_main,Ux) - acs(ni-2,j_main,Ux)) * Xi_x1(ni,j_main)

! Allow radial variation in axial velocity.
u_x2(ni,j_main) = 0.5d0 * (acs(ni,j_p_one,Ux) - &
acs(ni,j_m_one,Ux)) * Eta_x2(ni,j_main)

! Radial velocity
if (end2_VBC == 'sl') then ! Slip velocity in radial direction
  ! Neumann BC: Normal gradient of radial velocity is zero at the end.
v_x1(ni,j_main) = 0.0d0
endif

! Non-zero gradient along the boundary
v_x2(ni,j_main) = 0.5d0 * (acs(ni,j_p_one,Vx) - &
acs(ni,j_m_one,Vx)) * Eta_x2(ni,j_main)
else ! No slip
! Dirichlet BC: Normal gradient of radial velocity may be non-zero.
v_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Vx) + &
4.0d0*acs(ni-1,j_main,Vx) - acs(ni-2,j_main,Vx)) * Xi_x1(ni,j_main)
endif

! Zero radial velocity -> zero gradient along the boundary
v_x2(ni,j_main) = 0.0d0
end if

! Temperature
if (end1_TBC == 'iso') then
  ! Isothermal -- Dirichlet BC
t_x1(ni,j_main) = -0.5d0 * (-3.0d0*acs(ni,j_main,Tx) + &
4.0d0*acs(ni-1,j_main,Tx) - acs(ni-2,j_main,Tx)) * Xi_x1(ni,j_main)
else ! Adiabatic -- Neumann BC
t_x1(ni,j_main) = 0.0d0
endif
end select

#ifdef HAVE_MPI
if (end1_BC == 'mpi' .and. end2_BC == 'mpi') then
  send_end1(:,1) = u_x1(2,j_main)
send_end1(:,2) = v_x1(2,j_main)
send_end1(:,3) = u_x2(2,j_main)
send_end1(:,4) = v_x2(2,j_main)
send_end1(:,5) = t_x1(2,j_main)
send_end2(:,1) = u_x1(ni-1,j_main)
send_end2(:,2) = v_x1(ni-1,j_main)
send_end2(:,3) = u_x2(ni-1,j_main)
#endif
send_end2(:,4) = v_x2(ni-1,j_main)
send_end2(:,5) = t_x1(ni-1,j_main)

call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)

u_x1(1,j_main) = recv_end1(:,1)
v_x1(1,j_main) = recv_end1(:,2)
u_x2(1,j_main) = recv_end1(:,3)
v_x2(1,j_main) = recv_end1(:,4)
t_x1(1,j_main) = recv_end1(:,5)

u_x1(ni,j_main) = recv_end2(:,1)
v_x1(ni,j_main) = recv_end2(:,2)
u_x2(ni,j_main) = recv_end2(:,3)
v_x2(ni,j_main) = recv_end2(:,4)
t_x1(ni,j_main) = recv_end2(:,5)

else if (end1_BC == 'per' .and. end2_BC == 'mpi') then
   send_end1(:,1) = u_x1(3,j_main)
send_end1(:,2) = v_x1(3,j_main)
send_end1(:,3) = u_x2(3,j_main)
send_end1(:,4) = v_x2(3,j_main)
send_end1(:,5) = t_x1(3,j_main)

   send_end2(:,1) = u_x1(ni-1,j_main)
send_end2(:,2) = v_x1(ni-1,j_main)
send_end2(:,3) = u_x2(ni-1,j_main)
send_end2(:,4) = v_x2(ni-1,j_main)
send_end2(:,5) = t_x1(ni-1,j_main)

call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)

u_x1(1,j_main) = recv_end1(:,1)
v_x1(1,j_main) = recv_end1(:,2)
u_x2(1,j_main) = recv_end1(:,3)
v_x2(1,j_main) = recv_end1(:,4)
t_x1(1,j_main) = recv_end1(:,5)

u_x1(ni,j_main) = recv_end2(:,1)
v_x1(ni,j_main) = recv_end2(:,2)
u_x2(ni,j_main) = recv_end2(:,3)
v_x2(ni,j_main) = recv_end2(:,4)
t_x1(ni,j_main) = recv_end2(:,5)

else if (end1_BC == 'mpi' .and. end2_BC == 'per') then
   send_end1(:,1) = u_x1(2,j_main)
send_end1(:,2) = v_x1(2,j_main)
send_end1(:,3) = u_x2(2,j_main)
send_end1(:,4) = v_x2(2,j_main)
send_end1(:,5) = t_x1(2,j_main)
send_end2(:,1) = u_x1(ni-2,j_main)
send_end2(:,2) = v_x1(ni-2,j_main)
send_end2(:,3) = u_x2(ni-2,j_main)
send_end2(:,4) = v_x2(ni-2,j_main)
send_end2(:,5) = t_x1(ni-2,j_main)

call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)

u_x1(1,j_main) = recv_end1(:,1)
v_x1(1,j_main) = recv_end1(:,2)
u_x2(1,j_main) = recv_end1(:,3)
v_x2(1,j_main) = recv_end1(:,4)
t_x1(1,j_main) = recv_end1(:,5)

u_x1(ni,j_main) = recv_end2(:,1)
v_x1(ni,j_main) = recv_end2(:,2)
u_x2(ni,j_main) = recv_end2(:,3)
v_x2(ni,j_main) = recv_end2(:,4)
t_x1(ni,j_main) = recv_end2(:,5)

else if (end1_BC == 'mpi') then
    send_end1(:,1) = u_x1(2,j_main)
send_end1(:,2) = v_x1(2,j_main)
send_end1(:,3) = u_x2(2,j_main)
send_end1(:,4) = v_x2(2,j_main)
send_end1(:,5) = t_x1(2,j_main)

call share_lobound_mpi(send_end1, recv_end1)

u_x1(1,j_main) = recv_end1(:,1)
v_x1(1,j_main) = recv_end1(:,2)
u_x2(1,j_main) = recv_end1(:,3)
v_x2(1,j_main) = recv_end1(:,4)
t_x1(1,j_main) = recv_end1(:,5)

else if (end2_BC == 'mpi') then
    send_end2(:,1) = u_x1(ni-1,j_main)
send_end2(:,2) = v_x1(ni-1,j_main)
send_end2(:,3) = u_x2(ni-1,j_main)
send_end2(:,4) = v_x2(ni-1,j_main)
send_end2(:,5) = t_x1(ni-1,j_main)

call share_hibound_mpi(send_end2, recv_end2)

u_x1(ni,j_main) = recv_end2(:,1)
v_x1(ni,j_main) = recv_end2(:,2)
u_x2(ni,j_main) = recv_end2(:,3)
v_x2(ni,j_main) = recv_end2(:,4)
t_x1(ni,j_main) = recv_end2(:,5)

end if
if (end1_BC == 'per' .and. end2_BC == 'per') then
  u_x1(1,j_main) = u_x1(ni-2,j_main)
  v_x1(1,j_main) = v_x1(ni-2,j_main)
  u_x2(1,j_main) = u_x2(ni-2,j_main)
  v_x2(1,j_main) = v_x2(ni-2,j_main)
  t_x1(1,j_main) = t_x1(ni-2,j_main)
  u_x1(ni,j_main) = u_x1(3,j_main)
  v_x1(ni,j_main) = v_x1(3,j_main)
  u_x2(ni,j_main) = u_x2(3,j_main)
  v_x2(ni,j_main) = v_x2(3,j_main)
  t_x1(ni,j_main) = t_x1(3,j_main)
end if

RETURN

END SUBROUTINE BCgr_xi

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BCgr_eta(u_x1, u_x2, v_x1, v_x2, t_x2)
! =========================================================================
! | This subroutine sets the boundary conditions for the velocity |
! | gradients u_x1, u_x2, v_x1, v_x2, and the temperature gradient t_x2 |
! | in the x2-direction. |
! | Add boundary conditions in the x1-direction for u_x1 and v_x1. |
! =========================================================================
implicit none

real(DP), dimension(:,:), intent(OUT) :: u_x1, u_x2, v_x1, v_x2, t_x2

!!! Centerline (min r,eta,j) boundary condition
! Assume symmetric.
! du/dz: center difference along the boundary
u_x1(i_main,1) = 0.5d0 * (acs(i_p_one,1,Ux) - &
                     acs(i_m_one,1,Ux)) * Xi_x1(i_main,1)

! du/dr: one-sided FD in r-direction
u_x2(i_main,1) = 0.5d0 * (-3.0d0*acs(i_main,1,Ux) + &
                      4.0d0*acs(i_main,2,Ux) - acs(i_main,3,Ux)) * Eta_x2(i_main,1)

! dv/dz: center difference along the boundary
v_x1(i_main,1) = 0.5d0 * (acs(i_p_one,1,Vx) - &
                      acs(i_m_one,1,Vx)) * Xi_x1(i_main,1)
! dv/dr: one-sided FD in r-direction
v_x2(i_main,1) = 0.5d0 * (-3.0d0*acs(i_main,1,Vx) + &
   4.0d0*acs(i_main,2,Vx) - acs(i_main,3,Vx)) * Eta_x2(i_main,1)

! dT/dr: Adiabatic -- one-sided FD in r-direction
t_x2(i_main,1) = 0.5d0 * (-3.0d0*acs(i_main,1,Tx) + &
   4.0d0*acs(i_main,2,Tx) - acs(i_main,3,Tx)) * Eta_x2(i_main,1)

** Wall (max R,Eta,j) boundary condition
select case (wall_BC)

  case ("rig") ! Rigid boundary, no-slip velocity
  v_x1(i_main,nj) = 0.0d0
  v_x2(i_main,nj) = -0.5d0 * (-3.0d0*acs(i_main,nj,Vx) + &
    4.0d0*acs(i_main,nj-1,Vx) - acs(i_main,nj-2,Vx)) * &
    Eta_x2(i_main,nj)
  u_x2(i_main,nj) = -0.5d0 * (-3.0d0*acs(i_main,nj,Ux) + &
    4.0d0*acs(i_main,nj-1,Ux) - acs(i_main,nj-2,Ux)) * &
    Eta_x2(i_main,nj)

  if (wall_VBC == 'sl') then ! Slip velocity in axial direction
    u_x1(i_main,nj) = 0.5d0 * (acs(i_p_one,nj,Ux) - &
      acs(i_m_one,nj,Ux)) * Xi_x1(i_main,nj)
  else ! No slip
    u_x1(i_main,nj) = 0.0d0
  end if

  ! Temperature -- both isothermal and adiabatic
  t_x2(i_main,nj) = -0.5d0 * (-3.0d0*acs(i_main,nj,Tx) + &
    4.0d0*acs(i_main,nj-1,Tx) - acs(i_main,nj-2,Tx)) * Eta_x2(i_main,nj)

  case ("sym") ! Symmetric
  case ("per") ! Periodic
  case ("abs") ! Absorbing
  end select

RETURN

END SUBROUTINE BCgr_eta

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BCgrad_xi(u_x1, v_x1, t_x1, p_x1)
```fortran
#ifdef HAVE_MPI
USE mpi_fun
#endif

implicit none

real(DP), dimension(:,::), intent(INOUT) :: u_x1, v_x1, t_x1
real(DP), dimension(:,::), OPTIONAL, intent(INOUT) :: p_x1

#ifdef HAVE_MPI
! Local arrays.
Real(DP), dimension(nj,4) :: send_end1, send_end2
Real(DP), dimension(nj,4) :: recv_end1, recv_end2
#endif

!** End_1 (min x1,xi,i) boundary condition
select case (end1_BC)
case ('rig')
  u_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Ux) + 
                     4.0d0*acs(2,:,Ux) - acs(3,:,Ux)) * Xi_x1(1,:)
  if (end1_VBC == 'sl') then
    v_x1(1,:) = 0.0d0
  else
    v_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Vx) + 
                         4.0d0*acs(2,:,Vx) - acs(3,:,Vx)) * Xi_x1(1,:)
  end if
  if (end1_TBC == 'adi') then
    t_x1(1,:) = 0.0d0
  else
    t_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Tx) + 
                          4.0d0*acs(2,:,Tx) - acs(3,:,Tx)) * Xi_x1(1,:) 
  end if
  if (present(p_x1)) then
    p_x1(1,:) = 0.0d0
  end if
end case
case ('moc')
  u_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Ux) + 
                     4.0d0*acs(2,:,Ux) - acs(3,:,Ux)) * Xi_x1(1,:)
  v_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Vx) + 
                     4.0d0*acs(2,:,Vx) - acs(3,:,Vx)) * Xi_x1(1,:)
  t_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Tx) + 
                     4.0d0*acs(2,:,Tx) - acs(3,:,Tx)) * Xi_x1(1,:) 
  if (present(p_x1)) then
    p_x1(1,:) = 0.5d0 * (-3.0d0*acs(1,:,Px) + 
                      4.0d0*acs(2,:,Px) - acs(3,:,Px)) * Xi_x1(1,:)
  end if
end case
end select

!** End_2 (max x1,xi,i) boundary condition
```
select case (end2_BC)

case ('rig')
  u_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Ux) + &
                  4.0d0*acs(ni-1,:,Ux) - acs(ni-2,:,Ux)) * Xi_x1(ni,:)
  if (end1_VBC == 'sl') then
    v_x1(ni,:) = 0.0d0
  else
    v_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Vx) + &
                  4.0d0*acs(ni-1,:,Vx) - acs(ni-2,:,Vx)) * Xi_x1(ni,:)
  end if
  if (end1_TBC == 'adi') then
    t_x1(ni,:) = 0.0d0
  else
    t_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Tx) + &
                  4.0d0*acs(ni-1,:,Tx) - acs(ni-2,:,Tx)) * Xi_x1(ni,:)
  end if
  if (present(p_x1)) then
    p_x1(ni,:) = 0.0d0
  end if
end case

case ('moc')
  u_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Ux) + &
                  4.0d0*acs(ni-1,:,Ux) - acs(ni-2,:,Ux)) * Xi_x1(ni,:)
  v_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Vx) + &
                  4.0d0*acs(ni-1,:,Vx) - acs(ni-2,:,Vx)) * Xi_x1(ni,:)
  t_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Tx) + &
                  4.0d0*acs(ni-1,:,Tx) - acs(ni-2,:,Tx)) * Xi_x1(ni,:)
  if (present(p_x1)) then
    p_x1(ni,:) = -0.5d0 * (-3.0d0*acs(ni,:,Px) + &
                  4.0d0*acs(ni-1,:,Px) - acs(ni-2,:,Px)) * Xi_x1(ni,:)
  end if
end case

end select

#elseif HAVE_MPI
  if (end1_BC == 'mpi'.and. end2_BC == 'mpi') then
    send_end1(:,1) = u_x1(2,:)
    send_end1(:,2) = v_x1(2,:)
    send_end1(:,3) = t_x1(2,:)
    if (present(p_x1)) send_end1(:,4) = p_x1(2,:)

    send_end2(:,1) = u_x1(ni-1,:)
    send_end2(:,2) = v_x1(ni-1,:)
    send_end2(:,3) = t_x1(ni-1,:)
    if (present(p_x1)) send_end2(:,4) = p_x1(ni-1,:)

    call share_bounds mpi(send_end1, send_end2, recv_end1, recv_end2)

    u_x1(1,:) = recv_end1(:,1)
    v_x1(1,:) = recv_end1(:,2)
t_x1(1,:) = recv_end1(:,3)
if (present(p_x1)) p_x1(1,:) = recv_end1(:,4)

u_x1(ni,:) = recv_end2(:,1)
v_x1(ni,:) = recv_end2(:,2)
t_x1(ni,:) = recv_end2(:,3)
if (present(p_x1)) p_x1(ni,:) = recv_end2(:,4)

else if (end1_BC == 'per' .and. end2_BC == 'mpi') then
    send_end1(:,1) = u_x1(3,:)
    send_end1(:,2) = v_x1(3,:)
    send_end1(:,3) = t_x1(3,:)
    if (present(p_x1)) send_end1(:,4) = p_x1(3,:)

    send_end2(:,1) = u_x1(ni-1,:)
    send_end2(:,2) = v_x1(ni-1,:)
    send_end2(:,3) = t_x1(ni-1,:)
    if (present(p_x1)) send_end2(:,4) = p_x1(ni-1,:)

    call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)

    u_x1(1,:) = recv_end1(:,1)
v_x1(1,:) = recv_end1(:,2)
t_x1(1,:) = recv_end1(:,3)
if (present(p_x1)) p_x1(1,:) = recv_end1(:,4)

    u_x1(ni,:) = recv_end2(:,1)
v_x1(ni,:) = recv_end2(:,2)
t_x1(ni,:) = recv_end2(:,3)
if (present(p_x1)) p_x1(ni,:) = recv_end2(:,4)

else if (end1_BC == 'mpi' .and. end2_BC == 'per') then
    send_end1(:,1) = u_x1(2,:)
    send_end1(:,2) = v_x1(2,:)
    send_end1(:,3) = t_x1(2,:)
    if (present(p_x1)) send_end1(:,4) = p_x1(2,:)

    send_end2(:,1) = u_x1(ni-2,:)
    send_end2(:,2) = v_x1(ni-2,:)
    send_end2(:,3) = t_x1(ni-2,:)
    if (present(p_x1)) send_end2(:,4) = p_x1(ni-2,:)

    call share_bounds_mpi(send_end1, send_end2, recv_end1, recv_end2)

    u_x1(1,:) = recv_end1(:,1)
v_x1(1,:) = recv_end1(:,2)
t_x1(1,:) = recv_end1(:,3)
if (present(p_x1)) p_x1(1,:) = recv_end1(:,4)

    u_x1(ni,:) = recv_end2(:,1)
v_x1(ni,:) = recv_end2(:,2)
t_x1(ni,:) = recv_end2(:,3)
if (present(p_x1)) p_x1(ni,:) = recv_end2(:,4)
else if (end1_BC == 'mpi') then
  send_end1(:,1) = u_x1(2,:)
  send_end1(:,2) = v_x1(2,:)
  send_end1(:,3) = t_x1(2,:)
  if (present(p_x1)) send_end1(:,4) = p_x1(2,:)
  call share_lobound_mpi(send_end1, recv_end1)
  u_x1(1,:) = recv_end1(:,1)
  v_x1(1,:) = recv_end1(:,2)
  t_x1(1,:) = recv_end1(:,3)
  if (present(p_x1)) p_x1(1,:) = recv_end1(:,4)
end if
#endif
RETURN
END SUBROUTINE BCgrad_xi

!---------------------------------------------------------------------------

SUBROUTINE BCgrad_eta(u_x2, v_x2, t_x2, p_x2)

implicit none

real(DP), dimension(1,:), intent(OUT) :: u_x2, v_x2, t_x2
real(DP), dimension(1,:), OPTIONAL, intent(OUT) :: p_x2

*** Centerline (min r,eta,j) boundary condition
u_x2(:,1) = 0.5d0 * (-3.0d0*acs(:,1,Ux) + &
                      4.0d0*acs(:,1,Vx) - acs(:,3,Ux)) * Eta_x2(:,1)
v_x2(:,1) = 0.5d0 * (-3.0d0*acs(:,1,Vx) + &
                      4.0d0*acs(:,1,Vx) - acs(:,3,Vx)) * Eta_x2(:,1)
t_x2(:,1) = 0.5d0 * (-3.0d0*acs(:,1,Tx) + &
4.0d0*acs(:,2,Tx) - acs(:,3,Tx)) * Eta_x2(:,1)
if (present(p_x2)) then
  p_x2(:,1) = 0.5d0 * (-3.0d0*acs(:,1,Px) + &
4.0d0*acs(:,2,Px) - acs(:,3,Px)) * Eta_x2(:,1)
end if

!** Wall (max R,Eta,j) boundary condition
u_x2(:,nj) = -0.5d0 * (-3.0d0*acs(:,nj,Ux) + &
4.0d0*acs(:,nj-1,Ux) - acs(:,nj-2,Ux)) * Eta_x2(:,nj)
v_x2(:,nj) = -0.5d0 * (-3.0d0*acs(:,nj,Vx) + &
4.0d0*acs(:,nj-1,Vx) - acs(:,nj-2,Vx)) * Eta_x2(:,nj)
t_x2(:,nj) = -0.5d0 * (-3.0d0*acs(:,nj,Tx) + &
4.0d0*acs(:,nj-1,Tx) - acs(:,nj-2,Tx)) * Eta_x2(:,nj)
if (present(p_x2)) then
  p_x2(:,nj) = -0.5d0 * (-3.0d0*acs(:,nj,Px) + &
4.0d0*acs(:,nj-1,Px) - acs(:,nj-2,Px)) * Eta_x2(:,nj)
end if

RETURN

END SUBROUTINE BCgrad_eta
!---------------------------------------------------------------------------
END MODULE boundary
D.4.7 FDiff_vars.fpp

MODULE FDiff_vars
!============================================================================
! This module contains the global variables for the finite difference operation.
!
! Table of contents:
! SUBROUTINE alloc_FDvars(isize, jsize, n_i, n_j)
! SUBROUTINE dealloc_FDvars()
!============================================================================

USE global

implicit none

! Computational array W_bar (ni,nj,5):
  real(DP), dimension(:,,:), allocatable :: Wb

! Computational array W_bar_prime (ni,nj,5):
  real(DP), dimension(:,,:), allocatable :: Wbp

! Computational arrays Fbar or Gbar (ni,nj,4):
  real(DP), dimension(:,,:), allocatable, target :: FGbar

! Computational arrays SFbar or SGbar (ni_main,nj_main,4):
! Try (ni,nj,4) bct 22jul10
  real(DP), dimension(:,,:), allocatable, target :: SFGbar

! Space-dependent acceleration source:
  real(DP), dimension(:,,:), allocatable :: fdrv

! Indices for central differencing:
  integer, dimension(:,), allocatable :: i_p_one, i_m_one
  integer, dimension(:,), allocatable :: j_p_one, j_m_one

! Public routine
  Public :: alloc_FDvars, dealloc_FDvars !, alloc_FDdiss, dealloc_FDdiss

CONTAINS
!------------------------------------------------------------------
SUBROUTINE alloc_FDvars()

  implicit none

  integer :: astat

  astat = 0
  allocate(Wb(ni,nj,nACSvar), STAT=astat)
  allocate(Wbp(ni,nj,nACSvar), STAT=astat)
  allocate(FGbar(ni,nj,nACSvar-1), STAT=astat)
Allocate(SFGbar(ni,nj,nACSvar-1), STAT=astat)
allocate(fdrv(ni,nj), STAT=astat)
allocate(i_p_one(ni_main), STAT=astat)
allocate(i_m_one(ni_main), STAT=astat)
allocate(j_p_one(nj_main), STAT=astat)
allocate(j_m_one(nj_main), STAT=astat)
RETURN

END SUBROUTINE alloc_FDvars

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE dealloc_FDvars()

integer :: astat

astat = 0
dealloca(Wb, STAT=astat)
dealocate(Wbp, STAT=astat)
deallocate(FGbar, STAT=astat)
deallocate(SFGbar, STAT=astat)
deallocate(fdrv, STAT=astat)
deallocate(i_p_one, STAT=astat)
deallocate(i_m_one, STAT=astat)
deallocate(j_p_one, STAT=astat)
deallocate(j_m_one, STAT=astat)

RETURN

END SUBROUTINE dealloc_FDvars

!----------------------------------------------------------------------------
END MODULE FDiff_vars
D.4.8   grid.fpp

MODULE grid
!=========================================================================
! This module contains the subroutines for grid generation and grid
! stretching.
!=========================================================================

USE global
USE input_vars

implicit none

! Index variables:
integer, dimension(:,), allocatable :: i_main, j_main

! Physical grid array:
real(DP), dimension(:,,:), allocatable, target :: tube_x1, tube_x2
! Grid array in cartesian coordinates for use with spherical tube:
real(DP), dimension(:,,:), allocatable, target :: tube_x, tube_y
real(DP), dimension(:,,:), pointer :: Xgrid, Ygrid
! Inner and outer radii of tube in spherical coordinates:
real(DP) :: Rho1, Rho2
! Radius in spherical coordinates for which the arc length of the taper
! angle equals the radius in cylindrical coordinates.
real(DP) :: RhoR
! 1/R factor in spherical coordinates.
real(DP), dimension(:,,:), allocatable :: rcp_tubeR
! Grid parameters:
real(DP) :: alph1_x1, alph2_x1, alph_x2
! Gravity variables:
real(DP), target :: g_x1, g_x2

! Metrics and Jacobian:
real(DP), dimension(:,,:), allocatable, target :: Xi_x1, Eta_x2
real(DP), dimension(:,,:), allocatable, target :: X1_xi, X2_eta
real(DP), dimension(:,,:), allocatable :: Jac, iJac ! Jacobian and inv

! Private
integer, private :: i_totalsize, j_totalsize

CONTAINS
!=========================================================================
SUBROUTINE init_grid()
!=========================================================================
! This subroutine initializes the main index arrays, i_main and j_main
! and calculates the timing variables, deltat, dtdz and dt/dr as well
! as the period in timesteps.
!=========================================================================

implicit none
integer :: i, j
real(DP) :: t_step

! Calculate major radii for spherical coordinates.
if (taper > 0.0d0) then
  ! Calculate reference radius of spherical tube.
  RhoR = radius/taper

! Calculate inner spherical radius of tapered tube.
if (Req_pos < 0.0d0) then
  ! Maintain equal volume between straight and tapered tube.
  Rho1 = 0.5d0*(dsqrt(2.0d0*radius**2.0d0/(1.0d0-dcos(taper)) - &
                    L*L/3.0d0) - L)
elseif (Req_pos > 1.0d0) then ! Limit Req_pos to (0,1)
  Rho1 = RhoR - L
else
  ! Instead of maintaining equal volume between straight and tapered tubes,
  ! use the same radius at z = L*Req_pos.
  Rho1 = RhoR - L*Req_pos
  print *, 'Rho1 = ', Rho1
end if

Rho2 = Rho1 + L
end if

! Figure out how big the grid should be (ni, nj).
call grid_sizer()

! Allocate memory for the grid arrays.
call alloc_grid()

! Point output grid variables to the arrays appropriate to the coord. sys.
if (taper == 0.0d0) then
  Xgrid => tube_x1
  Ygrid => tube_x2
else
  Xgrid => tube_x
  Ygrid => tube_y
end if

! Assign the main index arrays.
i_main = (/ (i+1, i=1, ni_main) /)
j_main = (/ (j+1, j=1, nj_main) /)

! Calculate Delta t.
! First find time step in seconds according to coordinate system.
if (taper == 0.0d0) then
  t_step = min(min(min_delta1_x1,min_delta2_x1), &
                min(min_delta1_x2,min_delta2_x2)) / C0
else
   t_step = min(min(min_delta1_x1,min_delta2_x1), &
               Rho1*min(min_delta1_x2,min_delta2_x2)) / C0
end if

! Scale the time step for stability by the CFL criterion.
deltat = CFL * t_step

! Make adjustments if the model is driven.
! Calculate the period in timesteps assuring that the subperiod durations
! are an integer number of time steps.
subperx = nint(1.0d0 / (drv_pdur(3) * fsrc * deltat))
periodx = drv_pdur(3) * subperx

! Adjust deltat and CFL number to accommodate integer periodx.
deltat = 1.0d0 / (fsrc * periodx)
CFL = deltat / t_step

if (output == 'per') then
! Calculate and overwrite txmax according to number of cycles specified
! in the input.
   txmax = txmin + periodx * &
       int(real(drv_pdur(1)*drv_pdur(3) + drv_pdur(2)) / real(drv_pdur(3)))
end if

! Calculate gravity components.
if (gravity) then
   g_x1 = -G * dcos(taper)
g_x2 = -G * dsin(taper)
else
   g_x1 = 0.0d0
g_x2 = 0.0d0
end if

RETURN

END SUBROUTINE init_grid

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE grid_sizer() ! The stretching parameter alph_x signifies a uniform grid in either direction. In that case, the user may specify either *size or min_delta_x*, and the other value will be calculated. If alph_x is greater than zero that signifies grid refinement, and the user must provide both min_ and max_delta_x in order for *size to be calculated.

#ifdef HAVE_MPI
USE mpi_fun
#endif

implicit none

integer :: IM, JM

! Initialize IM and JM relative to i_size and j_size, which are input
! variables indicating the number of grid points the user expects.
IM = i_size
JM = j_size

! Initialize offsets.
i_offset = 0
j_offset = 0

! X1 direction:
! Uniform grid:
if (min_delta1_x1 .ne. 0.0d0 .and. &
   (min_delta1_x1 == max_delta_x1 .or. IM == 0)) then
   ! Calculate IM to nearest integer.
   IM = nint(L/min_delta1_x1) + 1
   ! Adjust DXmin and DXmax to accommodate integer IM value.
   min_delta1_x1 = L/(IM-1.0d0)
   max_delta1_x1 = min_delta1_x1
end if

! Uniform grid with periodic boundaries at the tube ends:
if (periodic) then
   ! Calculate IM to nearest integer and add two more ghost points.
   IM = nint(L/min_delta1_x1) + 3
   ! Adjust DXmin and DXmax to accommodate integer IM value.
   min_delta1_x1 = L/(IM-3.0d0)
   max_delta1_x1 = min_delta1_x1
end if

! Otherwise use IM for size of refined grid.

! X2 direction:
if (taper == 0.0) then
   ! Uniform grid:
   if (min_delta1_x2 .ne. 0.0d0 .and. &
      (min(min_delta1_x2,min_delta2_x2) == max_delta_x2 .or. JM == 0)) then
      ! Calculate JM to nearest integer.
      JM = nint(radius/min_delta1_x2 + 0.5d0)
      ! Adjust DXmin and DXmax to accommodate integer IM value.
      min_delta1_x2 = radius/(JM-0.5d0)
      max_delta1_x2 = min_delta1_x2
   end if

! Otherwise use JM for size of refined grid.
else
  ! Uniform grid:
  if (min_delta1_x2 .ne. 0.0d0 .and. 
    (min(min_delta1_x2,min_delta2_x2) == max_delta_x2 .or. 
     JM == 0)) then
    ! Calculate JM to nearest integer.
    JM = nint(taper/min_delta1_x2 + 0.5d0)
    ! Adjust DXmin and DXmax to accommodate integer IM value.
    min_delta1_x2 = taper/(JM-0.5d0)
    max_delta_x2 = min_delta1_x2
  end if

  ! Otherwise use JM for size of refined grid.
end if

! Define the local grid size.
i = IM
j = JM

! Initialize the global grid size.
i_global = IM
j_global = JM

#ifdef HAVE_MPI

! Domain decomposition is in the (i-, x1-)direction only. Subtract the
! overlapped grid points from the total on all processors. Then add them back
! to the individual count.
i = floor(real(ni_global-2)/real(nprocs_x1)) + 2

! Define offset for multiple processors in x1-direction.
i_offset = myid *(ni - 2)

! The last processor gets the extra grid points, but i_offset has to be
! calculated before ni is redefined for this processor.
if (myid == nprocs_x1 - 1) then
  ni = ni_global - i_offset
end if

! No need to distribute grid in x2-direction.
j = j_global

#endif

! Define the size of interior FDiff space.
i_main = ni - 2
j_main = nj - 2

RETURN

END SUBROUTINE grid_sizer
SUBROUTINE makegrid()

! | This subroutine calculates the physical grid and stores it as z and r variables in the main acoustic variables space. |

#ifdef HAVE_MPI
    USE mpi_fun
#endif

implicit none

ingeger :: i, j
integer :: GP ! number of ghost points
integer, dimension(ni) :: Xi
real(DP), dimension(ni_global) :: XiN
integer, dimension(nj) :: Eta
real(DP), dimension(nj) :: EtaN
real(DP), dimension(ni_global) :: Xi_global
real(DP), dimension(ni_global) :: x1_globalgrid
real(DP), dimension(ni_global) :: x1_globalmetric
real(DP), dimension(ni) :: x1_gridarr, x1_metarr
real(DP), dimension(nj) :: x2_gridarr
real(DP), dimension(nj) :: x2_metarr
real(DP), dimension(ni,nj) :: X1_eta, X2_xi ! X1_xi, X2_eta are global
real(DP) :: X2_max
real(DP) :: dxN0, dxN1, dXiN, dEtaN, s0, s1, A, B, dy
real(DP), dimension(ni_global) :: u
real(DP), dimension(nj) :: v

if (periodic) then
    GP = 1 ! one ghost point on either end
else
    GP = 0
end if

! Define the computational domain indices.
Xi = (/ (i_offset + i - 1 - GP, i=1,ni) /)
Xi_global = (/ (i - 1 - GP, i=1,ni_global) /)
XiN = real(Xi_global,DP) / real((ni_global - 1),DP)

Eta = (/ (j - 1, j=1,nj) /)
EtaN = real(Eta,DP) / real((nj - 1),DP)

! Define X2_max depending on taper.
if (taper > 0.0d0) then
  X2_max = taper
else
  X2_max = radius
endif

! X1-direction:
! Uniform grid:
if (periodic) then
  ! Grid:
  tube_x1 = spread((Xi * min_delta1_x1), 2, nj)
  ! Metrics:
  x1_metarr = L / real(ni_global - 1 - 2*GP, DP)
  X1_xi = spread(x1_metarr, 2, nj)
  X1_eta = 0.0d0
elseif (min_delta1_x1 == max_delta_x1) then
  ! Grid:
  tube_x1 = spread((Xi * min_delta1_x1), 2, nj)
  ! Metrics:
  x1_metarr = L / real(ni_global - 1, DP)
  X1_xi = spread(x1_metarr, 2, nj)
  X1_eta = 0.0d0
else
  ! Grid:
  dxN0 = min_delta1_x1 / L
  dxN1 = min_delta2_x1 / L
  dXiN = 1.0d0 / real(ni_global - 1, DP)
  s0 = dXiN / dxN0
  s1 = dXiN / dxN1
  A = dsqrt(s0/s1)
  B = dsqrt(s0*s1)
  dy = asinh(B)
  u = 0.5d0 * (1.0d0 + dtanh(dy*(XiN - 0.5d0)) / dtanh(dy/2.0d0))
  x1_globalgrid = L * u / (A + (1.0d0-A) * u)
  x1_gridarr = x1_globalgrid((i_offset+1):(i_offset+ni))
  ! Metrics:
  ! Calculate metrics by 2nd-order center difference
  x1_globalmetric(2:ni_global-1) = 0.5d0*(x1_globalgrid(3:ni_global) - &
    x1_globalgrid(1:(ni_global-2)))
  x1_globalmetric(1) = 0.5d0*(-3.0d0*x1_globalgrid(1) + &
    4.0d0*x1_globalgrid(2) - &
    x1_globalgrid(3))
  x1_globalmetric(ni_global) = -0.5d0 * &
    (-3.0d0*x1_globalgrid(ni_global) + &
    4.0d0*x1_globalgrid(ni_global-1) - &
    x1_globalgrid(ni_global-2))
  x1_metarr = x1_globalmetric(i_offset+1:i_offset+ni)
\begin{verbatim}
X1_xi = spread(x1_metarr, 2, nj)
X1_eta = 0.0d0
end if

! X2-direction:
if (min(min_delta1_x2, min_delta2_x2) == max_delta_x2) then
  ! Grid:
tube_x2 = spread(((Eta+0.5d0) * min_delta1_x2), 1, ni)
  ! Metrics:
x2_metarr = X2_max / (nj_global - 0.5d0)
X2_eta = spread(x2_metarr, 1, ni)
X2_xi = 0.0d0
else
  ! Grid:
dxN0 = min_delta1_x2 / (X2_max - 0.5d0*min_delta1_x2)
dxN1 = min_delta2_x2 / (X2_max - 0.5d0*min_delta1_x2)
dEtaN = 1.0d0 / real(nj - 1, DP)
s0 = dEtaN / dxN0
s1 = dEtaN / dxN1
A = dsqrt(s0/s1)
B = dsqrt(s0*s1)
dy = isinch(B)
v = 0.5d0 * (1.0d0 + dtanh(dy*(EtaN - 0.5d0)) / dtanh(0.5d0*dy))
x2_gridarr = (X2_max - 0.5d0*min_delta1_x2) * v / (A + (1.0d0-A)*v) + &
  0.5d0*min_delta1_x2
tube_x2 = spread(x2_gridarr, 1, ni)
  ! Metrics:
x2_metarr(2:nj-1) = 0.5d0*(x2_gridarr(3:nj) - x2_gridarr(1:nj-2))
x2_metarr(1) = 0.5d0*(-3.0d0*x2_gridarr(1) + &
  4.0d0*x2_gridarr(2) - &
  x2_gridarr(3))
x2_metarr(nj) = -0.5d0*(-3.0d0*x2_gridarr(nj) + &
  4.0d0*x2_gridarr(nj-1) - &
  x2_gridarr(nj-2))
X2_eta = spread(x2_metarr, 1, ni)
X2_xi = 0.0d0
end if

! Extra calculations for tapered tube.
if (taper > 0.0d0) then
  ! Calculate grid arrays in cartesian coordinates.
tube_x = (Rho1 + tube_x1) * dcos(tube_x2) - Rho1
tube_y = (Rho1 + tube_x1) * dsin(tube_x2)
  ! Calculate 1/R for spherical coordinates.
rcp_tubeR = 1.0d0 / (Rho1 + tube_x1)
end if
\end{verbatim}
! Calculate the Jacobian and reciprocal in terms of the metrics.
Jac = (X1_xi * X2_eta - X1_eta * X2_xi)
iJac = 1.0d0 / Jac

! Calculate the inverse metrics dXi/dz, dXi/dr, dEta/dz, and dEta/dr
! in terms of the metrics and the Jacobian.
Xi_x1 = X2_eta * iJac
Xi_x2 = -X1_eta * iJac
Eta_x1 = -X2_xi * iJac
Eta_x2 = X1_xi * iJac

RETURN
END SUBROUTINE makegrid

FUNCTION isinch(H) RESULT(xout)
! Compute the inverse of the function H = sinh(x)/x
! Assume H > 1.
implicit none
real(DP) :: H
real(DP) :: xout
real(DP) :: v, w, HH
if (H < 2.7829681d0) then
  HH = H - 1.0d0
  xout = dsqrt(6.0d0*HH) * (1.0d0 - 0.15d0*HH + &
                0.057321429d0*HH**2.0d0 - 0.024907295d0*HH**3.0d0 + &
                0.0077424461d0*HH**4.0d0 - 0.0010794123d0*HH**5.0d0)
else
  v = dlog(H)
  w = 1.0d0/H - 0.028527431d0
  xout = v + (1.0d0 + 1.0d0/v)*dlog(2.0d0*v) - 0.02041793d0 + &
           0.24902722d0*w + 1.9496443d0*w**2.0d0 - &
           2.6294547d0*w**3.0d0 + 8.56796911d0*w**4.0d0
endif
END FUNCTION isinch
FUNCTION dacosh(xarg) RESULT(yout)

    implicit none

    double precision :: xarg, yout

    yout = dlog(xarg + dsqrt(xarg**2.0d0 - 1.0d0))

END FUNCTION dacosh

SUBROUTINE alloc_grid()

! This subroutine allocates memory for the main index arrays, grid, and metric arrays.
! 07oct05 btuttle created

integer :: astat

! Main index arrays.
    astat = 0
    allocate(i_main(ni_main), STAT=astat)
    allocate(j_main(nj_main), STAT=astat)

! Grid arrays.
    astat = 0
    allocate(tube_x1(ni, nj), STAT=astat)
    allocate(tube_x2(ni, nj), STAT=astat)
    if (taper > 0.0d0) then
        astat = 0
        allocate(tube_x(ni, nj), STAT=astat)
        allocate(tube_y(ni, nj), STAT=astat)
        allocate(rcp_tubeR(ni, nj), STAT=astat)
    end if

! Arrays for the metrics.
    astat = 0
    allocate(Xi_x1(ni, nj), STAT=astat)
    allocate(X1_xi(ni, nj), STAT=astat)
    allocate(X2_eta(ni, nj), STAT=astat)
    allocate(Eta_x2(ni, nj), STAT=astat)

! Array for the Jacobian.
    astat = 0
    allocate(Jac(ni, nj), STAT=astat)
    allocate(iJac(ni, nj), STAT=astat)
RETURN

END SUBROUTINE alloc_grid

SUBROUTINE fin_grid()

! This subroutine deallocates the main index arrays, grid, and metric arrays.
!
! 04nov09 btuttle created
!
! =======================================================================
!

implicit none

integer :: astat

astat = 0
dealocate(i_main, STAT=astat)
deallocate(j_main, STAT=astat)
deallocate(tube_x1, STAT=astat)
deallocate(tube_x2, STAT=astat)

if (taper > 0.0d0) then
  astat = 0
  deallocate(tube_x, STAT=astat)
deallocate(tube_y, STAT=astat)
deallocate(rcp_tubeR, STAT=astat)
end if

astat = 0
dealocate(Xi_x1, STAT=astat)
deallocate(X1_xi, STAT=astat)
deallocate(X2_eta, STAT=astat)
deallocate(Eta_x2, STAT=astat)
deallocate(Jac, STAT=astat)
deallocate(iJac, STAT=astat)

RETURN

END SUBROUTINE fin_grid

END MODULE grid
D.4.9 IC.fpp

MODULE IC

!===========================================================================
! This module contains the initial condition subroutines for stream2d.
!
! Table of contents:
! SUBROUTINE init_cond()
! SUBROUTINE gaussian_z(density_ampl)
! SUBROUTINE gauss_ctr(density_ampl)
! SUBROUTINE gauss_line(density_ampl)
! SUBROUTINE gauss_r_half(density_ampl)
! SUBROUTINE std_wav(density_ampl)
! SUBROUTINE std_wav_damped(density_ampl)
! SUBROUTINE trav_wav(P_ampl)
!===========================================================================

USE global
USE grid

implicit none

CONTAINS

!---------------------------------------------------------------------------
SUBROUTINE init_cond(ICstat)
!===========================================================================
!
! | This subroutine defines the initial condition of the acoustic |
! | variables array. |
!
! | In test mode, this is a gaussian pulse in density centered in the |
! | domain. In continue mode, the values of all variables are read |
! | from a file. Otherwise, everything is zero. |
!
USE IO_tools
USE input_vars
USE analyt

integer(i4b), intent(OUT) :: ICstat
real(DP) :: wvn, rho_ampl, U_ampl

ICstat = 0

P_ampl = P_tgt
rho_ampl = P_ampl / C02
U_ampl = P_ampl / (C0 * rho0)

select case (ic_choice)

  case (1) ! Gaussian pulse on z-axis

  ICstat = 1

end select
call gaussian_z(rho_ampl)

case (2) ! Centered Gaussian pulse
    call gauss_ctr(rho_ampl)

case (3) ! Gaussian pulse at z=length/5
    call gauss_line(rho_ampl)

case (4) ! Gaussian pulse at z=length/2
    call gauss_z_half(rho_ampl)

case (5) ! Standing wave
    call std_wav(U_ampl)

case (6) ! Standing wave based on damped natural frequency
    call std_wav_damped(rho_ampl)

case (7) ! Read initial pressure, z-velocity, and drive phase angle from a file.
    call read_slice()

case (8) ! Traveling wave
    call trav_wav(P_ampl)

case (9) ! Read acoustic variables from a Fortran binary unformatted file.
    ! Get initial time index (txmin) from acsdata file.
    ! Get source amplitude (S_0), and source phase (phi) from input file.
    call read_ICfile(ICstat)

case default ! quiescent medium
    acs(i_main,j_main,RHOx) = 0.0d0
    acs(i_main,j_main,Ux) = 0.0d0
    acs(i_main,j_main,Vx) = 0.0d0
    acs(i_main,j_main,Tx) = 0.0d0
    acs(i_main,j_main,Px) = 0.0d0

end select

RETURN

END SUBROUTINE init_cond

!---------------------------------------------------------------------------

SUBROUTINE gaussian_z(density_ampl)
!===========================================================================
! This subroutine puts a gaussian pulse on the z-axis.
!===========================================================================

real(DP), intent(IN) :: density_ampl

!---------------------------------------------------------------------------

END SUBROUTINE gaussian_z(density_ampl)
where ( tube_x2 < 0.5 )

acs(i_main,j_main,RHOx) = &
density_ampl*exp(-9000000.*tube_x2(i_main,j_main)**2)
elsewhere
acs(i_main,j_main,RHOx) = 0.0
end where
acs(i_main,j_main,Ux) = 0.0
acs(i_main,j_main,Vx) = 0.0

RETURN

END SUBROUTINE gaussian_z

SUBROUTINE gauss_ctr(density_ampl)
!
! | This subroutine puts a gaussian pulse in the center of the domain. |
! | ------------------------------------------------------------------ |
!
real(DP), intent(IN) :: density_ampl
real(DP) :: Xmid, Ymid, ln2, pwidth, pwsq
real(DP), dimension(size(acs,1),size(acs,2)) :: mask

! Find the X1 midpoint.
Xmid = Xgrid(int(ni_main/2.0),int(nj_main/2.0))
! Find the X2 midpoint.
Ymid = Ygrid(int(ni_main/2.0),int(nj_main/2.0))
! Find the pulse width.
pwidth = L/10.0
pwsq = pwidth*pwidth
ln2 = log(2.0)
! Calculate distance from the center.
mask = sqrt( (Xgrid - Xmid)**2 + (Ygrid - Ymid)**2)
! Put a pulse in the middle.
where ( mask < 4*pwidth )
acs(:,:,RHOx) = density_ampl*exp(-ln2 * &
  (((Xgrid-Xmid)**2)+((Ygrid-Ymid)**2))/pwsq)
elsewhere
acs(:,:,RHOx) = 0.0
end where
acs(:,:,Ux) = 0.0

RETURN

END SUBROUTINE gauss_ctr

SUBROUTINE gauss_line(density_ampl)
!
! ------------------------------------------------------------------
USE input_vars

implicit none

real(DP), intent(IN) :: density_ampl
real(DP) :: Gauss_width
real(DP), dimension(size(acs,1),size(acs,2)) :: Gaussian, tube_SR
real(DP), dimension(size(acs,1),size(acs,2)) :: GaussianU
real(DP) :: Pampl, Xs, Rs, tw0

tw0 = 0.0d0
Pampl = C02*density_ampl
Gauss_width = 6d0*max_delta_x1
Xs = src_pos1*L/100d0
tube_SR = Rho1+tube_x1
Rs = Rho1 + Xs
Gaussian = dexp(-0.5d0*((tube_SR - Rs)/Gauss_width)**2.0d0)
GaussianU = dexp(-0.5d0*((tube_x1 - 1.001d0*Xs)/Gauss_width)**2.0d0)

if (taper > 0.0d0) then
  acs(:,:,RHOx) = density_ampl * Gaussian * Rs*rcp_tubeR
  acs(:,:,Vx) = 0.0d0
  acs(:,:,Ux) = Pampl/(rho0*C0) * Gaussian * Rs*rcp_tubeR
  acs(:,:,Tx) = Pampl/(rho0*Cp) * Gaussian * Rs*rcp_tubeR
  acs(:,:,Px) = Rgas*( rho0 * acs(:,:,Tx) + T0*acs(:,:,RHOx) )
else
  acs(:,:,RHOx) = density_ampl * dexp(-0.5d0* &
      (((tube_x1-src_pos1*L/100d0)/Gauss_width)**2d0))
  acs(:,:,Ux) = 0.0d0
  acs(:,:,Vx) = 0.0d0
  acs(:,:,Px) = acs(:,:,RHOx) * CO2
  acs(:,:,Tx) = acs(:,:,Px) / (rho0*Cp)
end if

RETURN

END SUBROUTINE gauss_line

SUBROUTINE gauss_z_half(density_ampl)

real(DP), intent(IN) :: density_ampl

! This subroutine puts a gaussian pulse in density, pressure, and temperature at z=length/2. Velocity is zero.

! This subroutine puts a gaussian pulse in density, pressure, and temperature at a position along the x1 axis.

! =======================================================================

! ---------------------------------------------------------------------------

! ---------------------------------------------------------------------------

! SUBROUTINE gauss_z_half(density_ampl)

! =---------------------------------------------------------------------------

! This subroutine puts a gaussian pulse in density, pressure, and temperature at z=length/2. Velocity is zero.

! =---------------------------------------------------------------------------

! real(DP), intent(IN) :: density_ampl
integer :: i_pos, j_pos
real(DP), dimension(size(acs,1),size(acs,2)) :: mask

! Find the 1/2 point of z-axis
i_pos = ni_main/2 + 1
j_pos = nj_main/2
! Calculate distance from the center of the pulse
mask = sqrt( (tube_x1-tube_x1(i_pos+1,j_pos+1))**2 )
! Put a pulse at 1/5 z-axis
where ( mask < (L/5.0d0) )
   acs(:,:,RHOx) = density_ampl*exp(-100.0d0*mask**2)
elsewhere
   acs(:,:,RHOx) = 0.0d0
end where
acs(:,:,Ux) = 0.0d0
acs(:,:,Vx) = 0.0d0
acs(:,:,Px) = acs(:,:,RHOx) * C02
acs(:,:,Tx) = Tconst * acs(:,:,Px)
RETURN

END SUBROUTINE gauss_z_half

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE gauss_r_half(density_ampl)
! =======================================================================
! | This subroutine puts a gaussian pulse at r = r2/r. |
! =======================================================================
real(DP), intent(IN) :: density_ampl
real(DP) :: r_mid, ln2, pwidth, pwsq, r2 !, z_mid

! Find the 1/2 point of r2-axis
r_mid = r2/2.0
! Find the pulse width.
pwidth = r2/20.0
pwsq = pwidth*pwidth
ln2 = log(2.0)

where (abs(tube_x2 - r_mid) < 4*pwidth)
   acs(:,:,RHOx) = density_ampl*exp(-ln2*((tube_x2-r_mid)**2)/pwsq)
elsewhere
   acs(:,:,RHOx) = 0.0
end where
acs(i_main,j_main,Vx) = acs(i_main,j_main,RHOx) * C0 / rho0
acs(i_main,j_main,Ux) = 0.0
RETURN
SUBROUTINE std_wav(velocity_ampl)

! This subroutine puts a standing wave along the length of the tube in the z-direction.

USE input_vars
USE analyt

implicit none

real(DP), intent(IN) :: velocity_ampl
real(DP) :: nrmlz
real(DP) :: wt

wt = txmin*deltat*omega0

if (taper == 0.0d0) then

! Velocity
acs(:,:,Ux) = velocity_ampl*dsin(k_cyl*tube_x1) * &
            dsin(txmin*deltat*omega0 - radph)
acs(:,:,Vx) = 0.0

! Initialize Pressure and Temperature accordingly.
acs(:,:,Px) = P_ampl * dcos(k_cyl*tube_x1) * &
            dcos(txmin*deltat*omega0 - radph)
acs(:,:,Tx) = Tconst*acs(:,:,Px)

else

! Calculate density distribution
nrmlz = k*Rho1/(sin(k*Rho1) - bigB*cos(k*Rho1))
acs(i_main,j_main,Ux) = 0.0
acs(i_main,j_main,Vx) = 0.0
acs(i_main,j_main,Px) = C02*acs(i_main,j_main,RHOx)
acs(i_main,j_main,Tx) = Tconst*acs(i_main,j_main,Px)

end if

RETURN
SUBROUTINE std_wav_damped(density_ampl)
!
! | This subroutine puts a standing wave along the length of the tube in the z-direction according to the damped natural frequency. |
!
real(DP), intent(IN) :: density_ampl
real(DP) :: k

! Find wavenumber at damped natural frequency computed from previous run. 
k = omega / C0
! Find period in timesteps (overwriting previous calculation).
! periodx = 2 * nint(ni_main / CFL)
! Calculate phase angle to offset distribution at z=0
! phase = Pi - delta_z * ni_main * k
! Calculate density distribution
acs(i_main,j_main,RHOx)=density_ampl*cos(k*tube_x1(i_main,j_main)+theta)
! Set z-velocity to zero
acs(i_main,j_main,Ux) = 0.0
! Set r-velocity to zero
acs(i_main,j_main,Vx) = 0.0

RETURN
END SUBROUTINE std_wav_damped

SUBROUTINE trav_wav(pressure_ampl)
!
! | Initial condition traveling wave along the length of the tube with spherical phasing. |
!
implicit none

real(DP), intent(IN) :: pressure_ampl
real(DP) :: psi

psi = Pi/2d0
acs(:,:,Px) = pressure_ampl*dcos(-wvn_k*(tube_x1)-psi) * Rho1/(Rho1+tube_x1)
acs(:,:,Vx) = 0.0d0
acs(:,:,Ux) = pressure_ampl/(rho0*C0)*(dcos(-wvn_k*(tube_x1)-psi) + dsin(-wvn_k*(tube_x1)-psi)/(wvn_k*(Rho1+tube_x1))) * &
Rho1/(Rho1+tube_x1)
acs(:,:,RHOx) = pressure_ampl*dcos(-wvn_k*(tube_x1)-psi)/C02 * &
Rho1/(Rho1+tube_x1)
acs(:,:,Tx) = pressure_ampl*dcos(-wvn_k*(tube_x1)-psi)/(rho0*Cp) * &
Rho1/(Rho1+tube_x1)

RETURN

END SUBROUTINE trav_wav

!---------------------------------------------------------------------------
SUBROUTINE read_ICfile(IC_status)
! ===================================================================
! | Reads Fortran unformatted binary file written as program output | |
! | or as postprocessing combination of multiprocessor output.      | |
! | Assigns data to acs array as initial condition and inserts time | |
! | index as starting timex.                                        | |
! ===================================================================

USE input_vars
USE IO_tools
USE acs_types
USE grid
USE source
#ifdef HAVE_MPI
USE mpi_fun
#endif

implicit none

integer, intent(OUT) :: IC_status
integer, dimension(2) :: data_size
type(acsdata_type), dimension(:,,:), allocatable :: acs_array
logical :: theres_a_file
integer(i4b) :: astat, ierr, i, i_limit, IM, JM
integer, parameter :: IOU = 120
real(DP) :: icf_time, diff
integer(i4b), dimension(2) :: icf_size
integer(i4b), dimension(4) :: icf_range
#ifdef HAVE_MPI
    type(acsdata_type), dimension(:,,:), allocatable :: overlap, tmp_array
    integer(i4b) :: IM_last, N
#endif

! Initialize status
IC_status = 0
astat = 0
ierr = 0
#ifdef HAVE_MPI
! Initialize output
#endif
if (myid == MASTER) then !!** MASTER TASK **
#endif
!

! Check for existence of IC_file.
inquire(FILE = IC_file, EXIST = theres_a_file)
if (theres_a_file) then!

! Find the size needed for reading in the data.
  data_size = read_filesize(IC_file)
else
  data_size = (/ 0, 0 /)
  print *, "IC_file: File not found."
end if
#endif HAVE_MPI
end if !!** end MASTER TASK **
!! Broadcast the total array size to all / Receive data size from MASTER.
call bcast_intarray_mpi(data_size, MASTER)
#endif
if (sum(data_size) > 0) then

! Initialize my data size to the total data size for 1-CPU usage.
  IM = data_size(1)
  JM = data_size(2)
#endif HAVE_MPI
!! N-CPU: My data size is ((the total data size - ghost points) / number of processors) + ghost points.
  IM = floor(real(data_size(1)-2)/real(nprocs_x1)) + 2
  JM = data_size(2)

!! The last processor's data size is that plus the modulus.
! Take the modulus of the data size without ghost points.
! Ghost points have already been added to IM and need not be added again.
  IM_last = data_size(1) - (nprocs_x1-1) * (IM - 2)

!! If I'm the last processor then my data size is the last processor's.
if (myid == nprocs_x1 - 1) then
  IM = IM_last
end if
#endif
!

! Allocate array of type acs_data for storage of the loaded data.
allocate(acs_array(IM, JM), STAT=astat)
if (astat .ne. 0) print *, "Problem allocating acs_array."
#endif HAVE_MPI
!! Create the MPI datatype to transmit acsdata_type arrays.
call make_acsdata_mpitype(acs_array(1,1))
if (myid == MASTER) then !!** MASTER TASK **
  astat = 0
  allocate(overlap(2, JM), STAT=astat)
  if (astat .ne. 0) print *, "Problem allocating overlap."
#endif

! Specify how much of the file to read.
i_limit = IM

! Open the IC file.
open (UNIT=IOU, FILE=IC_file, STATUS='OLD', ACTION='READ', &
FORM='UNFORMATTED', IOSTAT=ierr)
if (ierr .ne. 0) print *, "Problem opening IC_file."

! Read header information.
read (IOU, IOSTAT=ierr) icf_size
read (IOU, IOSTAT=ierr) icf_range
read (IOU, IOSTAT=ierr) txmin
read (IOU, IOSTAT=ierr) icf_time
read (IOU, IOSTAT=ierr) S_0
read (IOU, IOSTAT=ierr) phi

! Read the first portion of the array from the file.
do i = 1, i_limit
   read (IOU) acs_array(i,:)
end do

#ifdef HAVE_MPI
! Save the last two rows that overlap with the next array.
overlap = acs_array(i_limit-1:i_limit,:)  
end if  /* end MASTER TASK */

!! Broadcast txmin, S_0, and phi to all processors.
!! Receive txmin, S_0, and phi from MASTER.
call bcast_ICheader_mpi(txmin, S_0, phi)

!! Broadcast IC_status to all / Receive IC_status (?) (not necessary)
!! Loop over number of processors.
do N = 1, nprocs - 1
   if (myid == MASTER) then  /* MASTER TASK */
   !! For the last processor, read to the last processor's data size.
   if (N == nprocs - 1) then
      i_limit = IM_last
   !! Allocate the tmp_array for the last processor.
   astat = 0
   allocate(tmp_array(IM_last, JM), STAT=astat)
   if (astat .ne. 0) print *, "Problem allocating tmp_array."
   else
   !! Allocate the tmp_array for the other processors.
   astat = 0
   allocate(tmp_array(IM, JM), STAT=astat)
   if (astat .ne. 0) print *, "Problem allocating tmp_array."
   end if
   !! Insert the overlap rows at the beginning of the tmp_array.
end if
tmp_array(1:2,:) = overlap

!! Read in the array from the file up to the data size calculated.
    do i = 3, i_limit
        read (IOU) tmp_array(i,:)
    end do

!! Send the tmp_array to processor N.
    call share_ICarray_mpi(tmp_array, N)

!! Save the last two rows that overlap with the next array.
    overlap = tmp_array(i_limit-1:i_limit,:)

!! Deallocate the tmp_array
    astat = 0
    deallocate(tmp_array, STAT=astat)
    if (astat .ne. 0) print *, "Problem deallocating tmp_array."
else                   \** end MASTER TASK **

!! Receive acs_array from MASTER.
    call share_ICarray_mpi(acs_array, N)

end if

!! End of Loop
end do

#endif

! Compare the grid loaded from the file to the one that was generated.
! Check that they are the same size
    if (sum( (/ ni, nj /) - (/ IM, JM /) ) == 0 ) then
        ! Check that the physical grid is the same.
        diff = sum(sum(acs_array(:,:,)%grid_x1-Xgrid(:,:,),2),1) + &
               sum(sum(acs_array(:,:,)%grid_x2-Ygrid(:,:,),2),1)
        if ( abs(diff) > 0.01d0 * min(min_delta1_x1, min_delta2_x1, &
                                     min_delta1_x2, min_delta2_x2) ) then
            print *, "IC_file: Physical grid is not the same."
            print *, "difference is ", diff
            IC_status = 1
        end if
    else
        print *, "IC_file: Data size is not the same."
        print *, "ni = ", ni
        print *, "nj = ", nj
        print *, "IM = ", IM
        print *, "JM = ", JM
        IC_status = 1
    end if

#ifdef HAVE_MPI
!! Sum IC error status and distribute to all. (MPI_Allreduce)
call dist_ICstat_mpi(IC_status)

if (myid == MASTER) then            !!** MASTER TASK **
#endif
    ! Close the IC file.
    close (IOU)
#endif HAVE_MPI

astat = 0
deallocate(overlap, STAT=astat)
if (astat .ne. 0) print *, "Problem deallocating overlap."
end if

! If IC status is not zero return without assigning array.
if (IC_status > 0) then
    astat = 0
    deallocate(acs_array, STAT=astat)
    if (astat .ne. 0) print *, "Problem deallocating acs_array."

    RETURN
end if

! Assign acs quantities from the temporary array to the acs array.
acs(:,:,RHOx) = acs_array(:,:,)%vars%rho
acs(:,:,Ux) = acs_array(:,:,)%vars%u
acs(:,:,Vx) = acs_array(:,:,)%vars%v
acs(:,:,Tx) = acs_array(:,:,)%vars%t
acs(:,:,Px) = acs_array(:,:,)%vars%p

astat = 0
deallocate(acs_array, STAT=astat)
if (astat .ne. 0) print *, "Problem deallocating acs_array."
else
    IC_status = 1
end if

RETURN

END SUBROUTINE read_ICfile

!---------------------------------------------------------------------------
END MODULE IC
MODULE input_vars

USE global

implicit none

real(DP) :: radius
real(DP) :: L
real(DP) :: taper
real(DP) :: Req_pos
real(DP) :: CFL
real(DP) :: min_delta1_x1
real(DP) :: min_delta2_x1
real(DP) :: min_delta1_x2
real(DP) :: min_delta2_x2
real(DP) :: max_delta_x1
real(DP) :: max_delta_x2
integer(i4b) :: nBL
integer(i4b) :: i_size
integer(i4b) :: j_size
integer(i4b) :: txmin
integer(i4b) :: txmax
integer(i4b), dimension(3) :: drv_pdur
integer(i4b) :: ic_choice
real(DP) :: theta
character(len=15) :: IC_file
integer(i4b) :: SPL
real(DP) :: P_tgt
real(DP) :: k
real(DP) :: bigB
real(DP) :: S_0
real(DP) :: S_1
real(DP) :: S_2
real(DP) :: fsrc
real(DP) :: phi
real(DP) :: src_pos1
real(DP) :: src_pos2
real(DP) :: SPpeak
real(DP) :: SAramp_rate
real(DP) :: SAramp_dur
character(len=3) :: fdrv_type
character(len=3) :: wall_BC
character(len=3) :: ctr_BC
character(len=3) :: end1_BC
character(len=3) :: end2_BC
character(len=3) :: wall_TBC
character(len=3) :: end1_TBC
character(len=3) :: end2_TBC
character(len=2) :: wall_VBC
character(len=2) :: end1_VBC
character(len=2) :: end2_VBC
logical :: gravity
logical :: analytic
character(len=3) :: streamout
logical :: probing
real(DP) :: exagT
real(DP) :: exagV
character(len=3) :: output
integer(i4b) :: outarg1
integer(i4b) :: outarg2
character(len=13) :: setname
character(len=3) :: outfmt
integer(i4b) :: datapos
integer(i4b) :: prb_xi
integer(i4b) :: prb_eta
integer(i4b) :: prb_start
integer(i4b) :: prb_stop
integer(i4b) :: prb_per_cyc
integer(i4b) :: restint
integer(i4b), dimension(3) :: drv_rint
real(DP) :: Uvol_pos

END MODULE input_vars
D.4.11 IOtools.fpp

MODULE IO_tools
!======================================================================
!
! This is a module full of handy tools for writing and reading files
! of acoustics data. The original goal was to create a utility to
! write data from each node of a PC cluster to the NFS server in Fortran
! unformatted binary format, combine those files later, and convert
! the data into a format that TecPlot can read.
!
! Table of contents:
! type acsvars_type
! type acsdata_type
! FUNCTION filenamer(fileset, number) RESULT(filename)
! FUNCTION int2str(intnumber, length) RESULT(strnumber)
! FUNCTION count_files(fileset) result(numfiles)
! SUBROUTINE fileout(filetimex)
! SUBROUTINE init_analysis()
! SUBROUTINE init_file(filename)
! SUBROUTINE write_timept()
! SUBROUTINE write_slice()
! SUBROUTINE read_slice()
! SUBROUTINE write_acsvars(filetimex)
! SUBROUTINE write_acsfile(filename, data)
! SUBROUTINE read_extents(filename, filesize, low1, low2, hi1, hi2)
! FUNCTION read_filesize(filename) result(header)
! SUBROUTINE read_acsdata(filename, data)
!======================================================================

USE global
USE grid
USE acs_types

implicit none

integer, parameter :: prbiou = 100
integer, parameter :: OSiou = 200
integer(i4b) :: prb_step
integer(i4b) :: p_digs, d_digs
integer(i4b), dimension(3) :: drv_pmax

CONTAINS

FUNCTION filenamer(fileset, timex, num) RESULT(filename)
! This function generates a file name from a root name called
! "fileset" and two integers, a time index and a processor number. It translates the integers into character strings
! and appends them to the end of the name.
USE input_vars

implicit none

character(len=*) :: fileset
integer :: timex
integer, optional :: num
character(len=len(fileset)+txdigs+prdigs+1) :: filename
character(len=prdigs) :: numstr
character(len=txdigs) :: txstr
integer(i4b) :: cyclex, subcycx

if (output == 'per') then ! For forced vib. name files by period and frac.
cyclex = int(real(timex) / real(periodx))
subcycx = int(real(drv_pmax(3)) * real(modulo(timex,periodx)) &
/ real(periodx))
txstr = cat(cat(cat(cat(cat('_',int2str(cyclex,p_digs)),"_"), &
int2str(subcycx,d_digs)),":"),int2str(drv_pmax(3),d_digs))
else ! Otherwise name files by time index.
txstr = int2str(timex,txdigs)
end if

if (present(num)) then
  numstr = int2str(num,prdigs)
  filename = cat(cat(trim(fileset) , txstr) , numstr)
else
  filename = cat(trim(fileset) , txstr)
end if

END FUNCTION filenamer

FUNCTION int2str(intnumber, length) RESULT(strnumber)
! This function converts an integer into a string representing that same number, but preceded with zeros to fill the length specified in the input.
integer :: intnumber, length, numlen
character(len=length) :: strnumber
character(len=4) :: lenstr

if (intnumber > 1000000000) then
  print *, "Input range of function exceeded: int2str(intnumber, length)"
  write (*,*(a,i10)) "intnumber = ", intnumber
STOP
end if

if (intnumber == 0) then
    numlen = 1
else
    numlen = int(log10(real(intnumber))) + 1
end if
write(UNIT=lenstr, FMT="(a,i1,a)"") "(i", numlen, ")"
write(UNIT=strnumber, FMT=lenstr) intnumber
do
    if (len(trim(strnumber)) == length) then
        exit
    else
        strnumber = cat('0', trim(strnumber))
    end if
end do

END FUNCTION int2str

!-------------------------------------------------------------------------
 FUNCTION count_files(fileset,timex) result(numfiles)
 ! ===============================================================
 ! | This function counts the number of files with the root name |
 ! | "fileset" by checking their existence and returns that value |
 ! | as "numfiles". |
 ! ===============================================================
 character(len=*) :: fileset
 character(len=len(fileset)+txdigs+prdigs+1) :: filename
 character(len=prdigs) :: numstr
 integer(i4b) :: numfiles, timex
 logical :: ex

numfiles=0
do
    if (output == 'reg') then
        filename = filenamer(fileset, timex, numfiles)
    else
        numstr = int2str(numfiles,prdigs)
        filename = cat(cat(trim(fileset) , "." ), numstr)
    end if
  ! Does the next file exist? If not, you are done.
ex = .false.
inquire (file = filename, exist = ex)
if (ex .eqv. .true.) then
    print *, cat("I found ", filename)
    numfiles = numfiles+1
else
    exit

FUNCTION count_files(fileset,timex) result(numfiles)
 !-------------------------------------------------------------------------
end if
end do

END FUNCTION count_files

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------
SUBROUTINE fileout(filetimex)
! This subroutine determines when to call write_acsvars according to the method specified in the input file.
!-------------------------------------------------------------------------

#ifdef HAVE_MPI
USE mpi_fun
#endif
USE FDiff_vars
USE input_vars
USE analyt
USE source

integer, intent(in) :: filetimex
integer :: i,j

! Output for plotting:
select case (output)
  case ("reg") ! Write a file every N time steps specified by outarg1
    offset by outarg2.
    if ( (filetimex - lastout) == outarg1 + outarg2) then
      lastout = filetimex - outarg2
    ! Transform variables to physical domain.
    do i = 1,nACSvar
      acs(:,:,i) = Wb(:,:,i) * iJac
    end do
    if (analytic) call calc_anl(filetimex)

    call write_acsvars(filetimex)
  end if

  case ("per") ! Regular output specified by the drv_pdur
    ! "drive period duration" input variable
    if ( modulo(filetimex - txmin, subperx) == 0 ) then
      ! Transform variables to physical domain.
      do i = 1,nACSvar
        acs(:,:,i) = Wb(:,:,i) * iJac
      end do
      if (analytic) call calc_anl(filetimex)

      call write_acsvars(filetimex)
    end if

end select

#ifdef HAVE_MPI
USE mpi_fun
#endif
USE FDiff_vars
USE input_vars
USE analyt
USE source
call write_acsvars(filetimex)
end if

case ("pks") ! Write periodic maximum pressure values to a file
    call findmax(filetimex)
    if ( filetimex - timepoint%timex > nint(0.9 * periodx) ) then
        ! Transform variables to physical domain.
        do i = 1,nACSvar
            acs(:,:,i) = Wb(:,:,i) * iJac
        end do
        call write_timept() ! Write timepoint to file
        if (filetimex > outarg1) then
            call write_slice(filetimex)
        end if
        ! Reset timepoint
        timepoint%value = 0.0
        timepoint%timex = filetimex
    end if
end select

! Allow probe output concurrent with other file output.
if (probing) then
    ! Write data at every time step from a point on the grid.
    if (modulo(filetimex, prb_step) == 0 ) then
        i = prb_xi
        j = prb_eta
        ! Transform variables to physical domain.
        call write_prb(filetimex, Wb(i,j,RHOx)*iJac(i,j), &
                        Wb(i,j,Ux)*iJac(i,j), Wb(i,j,Vx)*iJac(i,j), &
                        Wb(i,j,Px)*iJac(i,j), shake, PA_mon)
    end if
end if

! Allow streaming output concurrent with other file output.
if (streamout .ne. 'non') then
    if (output == 'per') then
        call write_acsvars(filetimex, Wb(i,j,RHOx)*iJac(i,j), &
                            Wb(i,j,Ux)*iJac(i,j), Wb(i,j,Vx)*iJac(i,j), &
                            Wb(i,j,Px)*iJac(i,j), shake, PA_mon)
    end if
end if
if ( modulo(filetimex - txmin, subperx) == 0 ) then
    call write_mean(filetimex)
    sumacs = 0.0d0
    n_sum = 0
end if
else
    if (modulo(filetimex,outarg1) == 0) then
        call write_mean(filetimex)
        sumacs = 0.0d0
        n_sum = 0
    end if
end if

! Output for restart:
if ( filetimex > txmin .and. modulo(filetimex,restint) == 0 ) then
    do i = 1,nACSvar
        acs(:,:,i) = Wb(:,:,i) * iJac
    end do
    call write_acsvars(filetimex,"bin")
else if ( filetimex == txmax ) then
    do i = 1,nACSvar
        acs(:,:,i) = Wb(:,:,i) * iJac
    end do
    call write_acsvars(filetimex,"bin")
end if

#elseif HAVE_MPI
    call sync_mpi()
#endif

RETURN
END SUBROUTINE fileout

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE findmax(timex)
! ========================================================================
! | This subroutine compares the current value of some acoustics variable |
! | with a previous value of that variable and updates the stored value   |
! | if the new one is larger.                                           |
! ========================================================================

gt integer, intent(in) :: timex
real(DP) :: thisval

thisval = acs(datapos,int(nj_main/2),Px)
if ( thisval > timepoint%value ) then
    timepoint%value = thisval
end if
timepoint%timex = timex
end if

RETURN

END SUBROUTINE findmax

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE init_out()
!============================================================================
! | This subroutine initializes the various analysis files and constructs. |
! |============================================================================

#ifdef HAVE_MPI
USE mpi_fun
#endif
USE input_vars
implicit none

! Make sure that datapos is within i_range
if (datapos > ni_main) then
    print *, "Datapos is out of range. Adjusting to ", ni_main
datapos = ni_main
end if
if (datapos < 1) then
    print *, "Datapos is out of range. Adjusting to 1"
datapos = 1
end if

! Find how many digits in txmax for use in filenamer().
! Type conversion to integer should truncate fraction.
if (output == 'per') then
    drv_pmax(1) = int(real(txmax) / real(periodx))
    drv_pmax(3) = drv_pdur(3)
    drv_pmax(2) = modulo(txmax,periodx) * &
        int(real(drv_pmax(3))/real(periodx))
    p_digs = 3
    d_digs = 2
    txdigs = p_digs + 2*d_digs + 3 ! Number of chars in _ppp_nn:dd
else
    txdigs = log10(real(txmax)) + 1
end if

! Initialize analysis output files
if (output == 'pks') then
slicefile = cat(trim(setname), '_slice')
analoutfile = cat(trim(setname), '_timept')
call init_file(trim(analoutfile))
end if

if (probing) then
#ifdef HAVE_MPI
if (myid == MASTER) then
#endif
call init_probe()
#ifdef HAVE_MPI
endif
#endif
!
! Define timesteps between probe outputs.
if (prb_per_cyc == 0) then
    prb_step = 1
else
    prb_step = nint(real(periodx) / real(prb_per_cyc))
end if
end if

! Define restart interval for driven simulation.
if (output == 'per') then
restint = periodx * drv_rint(1) + &
         nint( periodx*real(drv_rint(2)) / drv_rint(3) )
end if
RETURN
END SUBROUTINE init_out

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE init_file(filename)
! =================================================================
! | This subroutine replaces any existing file with name filename |
! | by a new empty file of the same name. |
! =================================================================
character(len=*, intent(in)) :: filename
integer, parameter :: iounit = 40
integer :: ierr

! Open file in replacement mode
open (UNIT=iounit, FILE=filename, STATUS='REPLACE', ACCESS='SEQUENTIAL', &
      IOSTAT=ierr)
! Write header for the data columns in Gnuplot format ('#' = comment).
write (iounit, "(a)") &
"# Time phase z-velocity z-velocity z-velocity pressure" write (iounit, "(a)") &
"# seconds source at piston middle right end left end"

! Close the file. Now it is ready.
close (UNIT=iounit)

RETURN

END SUBROUTINE init_file

!=================================================================
!=================================================================
SUBROUTINE init_probe()
!=================================================================
! | This subroutine initializes a probe file which tracks acoustic |
! | variables at a particular point for every time step. The      |
! | initialization overwrites previous prb files, opening a new   |
! | file and leaving it open until the final probe time index.    |
! | Created 6mar06 btuttle                                     |
!=================================================================

USE input_vars

integer :: xidigs, etadigs, ierr
character(len=6) :: prb_xi_str, prb_eta_str
logical :: prb_file_exists

! Find number of digits required for the filename.
xidigs = log10(real(ni-1)) + 1
etadigs = log10(real(nj-1)) + 1

! Convert the numbers to strings having leading zeros.
prb_xi_str = int2str(prb_xi, xidigs)
prb_eta_str = int2str(prb_eta, etadigs)

! Create a filename.
prbfile = cat(cat(cat(cat('Xi',trim(prb_xi_str)),'Eta'), &
trim(prb_eta_str)),'.prb')
inquire(FILE=prbfile, EXIST=prb_file_exists)

if (ic_choice == 9 .and. prb_file_exists) then
! Open the file in append mode.
   open (UNIT=prbiou, FILE=prbfile, STATUS='OLD', ACCESS='SEQUENTIAL', &
       POSITION='APPEND', IOSTAT=ierr)
else
! Open the file in replacement mode.
   open (UNIT=prbiou, FILE=prbfile, STATUS='REPLACE', &
       ACCESS='SEQUENTIAL', IOSTAT=ierr)

! Write header for the data columns in Gnuplot format ('#' = comment).
write (prbiou,"(a)") &
"#Tx Time acs_rho u-vel v-vel acs_press accel_src P_ampl"
write (prbiou,"(a)") &
"#  (sec)  (kg/m^3)  (m/s)  (m/s)  (Pa)  (m/s/s)  (Pa)"

end if

RETURN

END SUBROUTINE init_probe

SUBROUTINE write_prb(ftimex, acsrho, u_vel, v_vel, acspress, src_accel, Pampl)
!
! This subroutine writes data to the probe output file and closes it if the time index equals the stopping point.
! Created 6mar06 btuttle
!
!==================================================================================================

#endif HAVE_MPI
USE mpi_fun
#endif

integer, intent(IN) :: ftimex
real(DP), intent(IN) :: acsrho, u_vel, v_vel, acspress, src_accel, Pampl
real(DP) :: secs

secs = ftimex*deltat

#endif HAVE_MPI
! ** N.B. Only data from CPU node 0 is available for probe output. **
if (myid == MASTER) then
#endif
!
! The file is already open. Write the data to it.
write (prbiou, "(i9,f9.6,6es12.4)"") ftimex, secs, acsrho, u_vel, v_vel, &
acspress, src_accel, Pampl

#endif HAVE_MPI
end if
#endif

RETURN

END SUBROUTINE write_prb

SUBROUTINE write_timept()
!
! This subroutine writes the time index and acoustic variable data stored in the structure timepoint to an existing file.
!
!==================================================================================================
character(len=len(trim(setname))+7) :: filename
integer, parameter :: iounit = 40
integer :: ierr
real :: secs

filename = trim(analoutfile)
secs = timepoint%timex*deltat

! Open existing file and append to it.
open (UNIT=iounit, FILE=filename, STATUS='OLD', ACCESS='SEQUENTIAL', &
     ACTION='WRITE', POSITION='APPEND', IOSTAT=ierr)

! Write formatted timepoint data.
write(iounit,"(f9.5,tr1,f12.9)") secs, timepoint%value

! Close file.
close (UNIT=iounit)

RETURN

END SUBROUTINE write_timept

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------
SUBROUTINE write_slice(filetimex)
! =====================================================================
! | This subroutine writes a slice of pressure and velocity data in a |
! | text file. N.B. filetimex is designed to be the actual time |
! | index, and not just an integer for a file name so that the |
! | phase of the source can be included in the header. |
! |====================================================================|
#ifdef HAVE_MPI
USE mpi_fun, only: myid
#endif
USE source
implicit none
integer, intent(in) :: filetimex
character(len=len(trim(slicefile))+txdigs+prdigs+1) :: filename
integer, parameter :: iounit = 60
integer :: ierr, i_pos, j_pos, i, j
real(DP) :: phase
! dim is the slice dimension 1=z-dir, 2=r-dir
integer, parameter :: dim = 2

! Name the file
#ifdef HAVE_MPI
   filename = filenamer(trim(slicefile), filetimex, myid)
#else
   filename = filenamer(trim(slicefile), filetimex)
#endif

! Choose the j or i positions to be the middle of the domain.
   j_pos = nj_main/2
   i_pos = ni_main/2

! Calculate wrapped phase of source velocity
   phase = phi + mod(filetimex*deltat*omega, 2.0*Pi)

! Open and replace the file.
   open (UNIT=iounit, FILE=filename, STATUS='REPLACE', ACCESS='SEQUENTIAL', &
       ACTION='WRITE', IOSTAT=ierr)

! Write header in Gnuplot format ('#' = comment)
   write(iounit,100) S_0, phase
100 format ('# Source amplitude and phase', ES16.6, f7.4)
   if ( dim == 1 ) then
      write(iounit,"(a)") &
      "# Z-pos pressure z-velocity temperature"
   else
      write(iounit,"(a)") &
      "# R-pos pressure r-velocity temperature"
   end if

! Write formatted slice data in z-direction.
   do i = 1, ni_main+2
      write(iounit,200) Xgrid(i,j_pos), acs(i,j_pos,Px), &
      acs(i,j_pos,Ux), &
      acs(i,j_pos,Tx)
   end do

! Write formatted slice data in r-direction.
   do j = 1, nj_main+2
      write(iounit,200) Ygrid(i_pos,j), acs(i_pos,j,Px), &
      acs(i_pos,j,Vx), &
      acs(i_pos,j,Tx)
   end do

! Close file
   close (UNIT=iounit)

RETURN

END SUBROUTINE write_slice

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------
SUBROUTINE read_slice()
! -----------------------------------------------------------------------
! This subroutine reads a one-dimensional slice of acoustic data       |
! from a formatted text file. Values read from the file are            |
! appropriate to initialize a linear, inviscid model.                 |
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
USE source

integer, parameter :: iounit = 70
integer :: i, ierr, rec_size
character(len=28) :: header1
character(len=100) :: header2
real :: z, p, u, t!, s

rec_size = 28

! Open the existing slice file.
open (UNIT=iounit, FILE=trim(IC_file), STATUS='OLD', &
     ACCESS='SEQUENTIAL', ACTION='READ', IOSTAT=ierr)

! Read source amplitude and phase from the first line of the header.
read (iounit, *(a, ES16.6, f7.4)) header1, S_0, phi

! Skip the second line
read (iounit, *(a))

! Read the rest of the file and assign initial values.
do i = 1, ni_main+2
   read(iounit,300) z, p, u, t!, s
   acs(i,j_main,RHOx) = p/C02
   acs(i,j_main,Ux) = u
   acs(i,j_main,Tx) = t
!    acs(i,j_main,Sx) = s
end do
300 format (f7.4, 4ES16.6)

! Close the file
close (UNIT=iounit)

RETURN

END SUBROUTINE read_slice

-------------------------------------------------------------------------

SUBROUTINE write_acsvars(filetimex,output_format)
! ================================================================
! | This subroutine decides whether to write Fortran unformatted   |
! | binary as acsdata type or as Plot3D. In each case it divides   |
! | all but the initial variables by the jacobian to prepare for   |
! | writing in the physical coordinates.                          |
! ===============================================================

USE plot3d
USE input_vars
USE analyt
#ifdef HAVE_MPI
  USE mpi_fun
#endif

integer, intent(IN) :: filetimex
character(len=3), OPTIONAL, intent(IN) :: output_format

type(acsdata_type), dimension(:,,:), allocatable :: acsdata
real(SP), dimension(:,,:), allocatable :: outarray
real(SP), dimension(:,,:), allocatable :: outgrid_x, outgrid_y
character(len=len(trim(setname))+txdigs+prdigs+1) :: filename
character(len=len(trim(setname))+txdigs+prdigs+3) :: anlfilename
character(len=3) :: file_format
integer :: i, j, alloc_stat

if (present(output_format)) then
  file_format = output_format ! the local argument
else
  file_format = outfmt ! the global variable
endif

select case (file_format)

  case ("bin") ! Fortran unformatted acsdata type -- used for restart files
    ifdef HAVE_MPI
      filename = filenamer(trim(setname), filetimex, myid)
    ifdef
    filename = filenamer(trim(setname), filetimex)
  endif
  alloc_stat = 0
  allocate(acsdata(ni,nj), stat=alloc_stat)

  ! Write [1:ni,1:nj] -- all boundary points
  do j = 1, nj
    do i = 1, ni
      acsdata(i,j)%Xi = i + i_offset
      acsdata(i,j)%Eta = j + j_offset
      acsdata(i,j)%grid_x1 = Xgrid(i,j)
      acsdata(i,j)%grid_x2 = Ygrid(i,j)
      acsdata(i,j)%vars%rho = acs(i,j,RHOx)
      acsdata(i,j)%vars%u = acs(i,j,Ux)
      acsdata(i,j)%vars%v = acs(i,j,Vx)
      acsdata(i,j)%vars%t = acs(i,j,Tx)
      acsdata(i,j)%vars%p = acs(i,j,Px)
    end do
  end do

call write_acsfile(filename, filetimex, acsdata)

  alloc_stat = 0
  deallocate(acsdata, stat=alloc_stat)
case ("p3d") ! Pplot3D format
    ! The only one that does the writing is the master node.
    ! If there are any other nodes, they send their data to the master
    ! first. We already know how many total grid points there are.
    ! Name the output file.
    filename = filenamer(trim(setname), filetimex)
    if (filetimex == 0) then
        ! Allocate the grid arrays -- single precision.
        alloc_stat = 0
        allocate(outgrid_x(ni_global,nj_global), stat=alloc_stat)
        allocate(outgrid_y(ni_global,nj_global), stat=alloc_stat)
    #ifdef HAVE_MPI
        ! Get grid from other processors.
        call collect_grid_mpi(Xgrid, Ygrid, outgrid_x, outgrid_y)
        if (myid == MASTER) then
            outgrid_x(1:ni,:) = real(Xgrid(:,:),SP)
            outgrid_y(1:ni,:) = real(Ygrid(:,:),SP)
        #endif
        ! Write the grid.
        call write_P3Dgrid(filename, outgrid_x, outgrid_y)
    #ifdef HAVE_MPI
        end if
        call sync_mpi()
    #endif
    endif
    ! Allocate the output array -- single precision.
    alloc_stat = 0
    allocate(outarray(ni_global,nj_global,nACSvar), stat=alloc_stat)
    ! Initialize output array.
    outarray = 0.0
  #ifdef HAVE_MPI
    ! Get output array from slave processors.
    do j = 1, nACSvar
        call collect_acs_mpi_SP(acs(:,:,j), outarray(:,:,j))
    end do
    if (myid == MASTER) then
        #endif
        outarray(1:ni,:,1:nACSvar) = real(acs(:,:,,:),SP)
        call write_P3D(filename, outarray)
    #ifdef HAVE_MPI
        end if
        call sync_mpi()
    #endif
#endif

if (analytic) then
    anlfilename = cat(trim(filename) , "an")
#endif

#ifdef HAVE_MPI
    ! Get output array from slave processors.
    call collect_acs_mpi_SP(T_anl, outarray(:,:,Tx))
    call collect_acs_mpi_SP(P_anl, outarray(:,:,Px))
    call collect_acs_mpi_SP(U_anl, outarray(:,:,Ux))
endif

if (myid == MASTER) then
    outarray(1:ni,:,Tx) = real(T_anl,SP)
    outarray(1:ni,:,Px) = real(P_anl,SP)
    outarray(1:ni,:,Ux) = real(U_anl,SP)
endif
#endif

endif

end if

! Deallocate the output array.
alloc_stat = 0
deallocate(outarray, stat=alloc_stat)

case ("ncf") ! netCDF format
#ifdef HAVE_MPI
    filename = filenamer(trim(setname), filetimex, myid)
#else
    filename = filenamer(trim(setname), filetimex)
#endif
end select

RETURN

END SUBROUTINE write_acsvars

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------

SUBROUTINE write_mean(filetimex)
! ==============================================================
! | This subroutine decides whether to write Fortran unformatted |
! | binary as acsdata type or as Plot3D. In each case it divides |
! | all but the initial variables by the jacobian to prepare for |
! | writing in the physical coordinates. |
! ==============================================================

SUBROUTINE write_mean(filetimex)
USE FDiff_vars
USE plot3d
USE input_vars
USE analyt
#ifdef HAVE_MPI
USE mpi_fun
#endif

integer, intent(IN) :: filetimex
real(SP), dimension(:, :, :), allocatable :: outarray
real(DP), dimension(:, :, :), allocatable :: outarrayDP
character(len=len(trim(setname)) + txdigs + prdigs + 1) :: filename
character(len=len(trim(setname)) + txdigs + prdigs + 3) :: strfilename
integer :: j, alloc_stat

select case (streamout)

case ("bin") ! Fortran double precision
  ! The only one that does the writing is the master node.
  ! If there are any other nodes, they send their data to the master
  ! first. We already know how many total grid points there are.

  ! Name the output file.
  filename = filenamer(trim(setname), filetimex)
  strfilename = cat(trim(filename) , ".mn")

  ! Allocate the output array -- single precision.
  alloc_stat = 0
  allocate(outarrayDP(ni_global, nj_global, nSACSvar), stat=alloc_stat)
  ! Initialize output array.
  outarrayDP = 0.0d0

#ifdef HAVE_MPI
  ! Get output array from slave processors.
  do j = 1, nSACSvar
    call collect_acs_mpi_DP(sumacs(:, :, j)/real(n_sum), outarrayDP(:, :, j))
  end do

  if (myid == MASTER) then
    endif
    outarrayDP(1:ni, :, :) = real(sumacs(:, :, :)/real(n_sum))

    call write_array(strfilename, outarrayDP)
#endif

! Deallocate the output array.
alloc_stat = 0
deallocate(outarrayDP, stat=alloc_stat)
case ("p3d") ! Plot3D format
  ! The only one that does the writing is the master node.
  ! If there are any other nodes, they send their data to the master
  ! first. We already know how many total grid points there are.

  ! Name the output file.
  filename = filenamer(trim(setname), filetimex)
  strfilename = cat(trim(filename) , "st")

  ! Allocate the output array -- single precision.
  alloc_stat = 0
  allocate(outarray(ni_global,nj_global,nSACSvar), stat=alloc_stat)

  ! Initialize output array.
  outarray = 0.0

#ifdef HAVE_MPI
  ! Get output array from slave processors.
  do j = 1, nSACSvar
    call collect_acs_mpi_SP(sumacs(:,:,j)/real(n_sum), outarray(:,:,j))
  end do

  if (myid == MASTER) then
    outarray(1:ni,:, :) = real(sumacs(:,:, :) / real(n_sum))
    call write_P3D(strfilename, outarray)
#endif

end if

#endif

! Deallocate the output array.
alloc_stat = 0
deallocate(outarray, stat=alloc_stat)

! end if

case ("ncf") ! netCDF format
#ifdef HAVE_MPI
  filename = filenamer(trim(setname), filetimex, myid)
#else
  filename = filenamer(trim(setname), filetimex)
#endif

strfilename = cat(trim(filename) , "st")

end select

RETURN
END SUBROUTINE write_mean

SUBROUTINE write_array(filename, outarr)

  ! Open file
  open (UNIT=iounit, FILE=trim(filename), STATUS='REPLACE', &
       ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=ierr)

  ! Write header
  write(iounit) i_size, j_size, vx_max

  ! Write data
  write(iounit) (((outarr(i,j,vx), i=1,i_size), j=1,j_size), vx=1,vx_max)

  ! Close file
  close (UNIT=iounit)
END SUBROUTINE write_array

SUBROUTINE write_acsfile(filename, timex, acdata)

  USE source

  ! Open file
  open (UNIT=iounit, FILE=trim(filename), STATUS='REPLACE', &
       ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=ierr)

  ! Write header
  write(iounit) i_size, j_size, vx_max

  ! Write data
  write(iounit) (((outarr(i,j,vx), i=1,i_size), j=1,j_size), vx=1,vx_max)

  ! Close file
  close (UNIT=iounit)
END SUBROUTINE write_acsfile
! Declare variables
character(len=*) :: filename
! type(acsdata_type), dimension(:,:), intent(in) :: acdata
type(acsdata_type), dimension(:), allocatable :: rowvector
integer, intent(in) :: timex
integer, parameter :: IOU = 30
integer, dimension(2) :: array_size
integer, dimension(4) :: comp_range
real(DP) :: time
integer :: i, ierr, astat, n_i, n_j
n_i = size(acdata,1)
n_j = size(acdata,2)
allocate(rowvector(n_j), STAT=astat)

! Create header information.
array_size = (/ n_i, n_j /)
comp_range = (/ acdata(1,1)%Xi, acdata(1,1)%Eta, &
                acdata(n_i,n_j)%Xi, acdata(n_i,n_j)%Eta /)
time = real(timex) * deltat ! in seconds

! Open file for binary output
open (UNIT=IOU, FILE=filename, STATUS='REPLACE', ACTION='WRITE', &
      FORM='UNFORMATTED', IOSTAT=ierr)
write(IOU) array_size
write(IOU) comp_range
write(IOU) timex
write(IOU) time
write(IOU) S_0
write(IOU) phi

! Write data to the file. One record is one row (j-direction) of data.
do i = 1, n_i
  rowvector = acdata(i,:) !
  do j = 1, n_j
    !
    write(IOU) rowvector
  end do
end do

! Close file
close (UNIT=IOU)
deallocate(rowvector, STAT=astat)

RETURN

END SUBROUTINE write_acsfile

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------
FUNCTION read_filesize(filename) result(filesize)
!  ========================================================================
! | This function reads the first record of a Fortran binary unformatted |
! | file and returns a 1x2 array of integers indicating the size of the |
! | data array stored in the rest of the file. |
!  ========================================================================
character(len=*) :: filename
integer, dimension(2) :: filesize
integer, parameter :: iounit = 30
integer :: ierr, stat

! Open the binary file.
open (UNIT=iounit, FILE=filename, STATUS='OLD', ACTION='READ', &
   FORM='UNFORMATTED', IOSTAT=ierr)

! Read data sizes from the file
read(iounit, IOSTAT=stat) filesize

! Close file
close (iounit)

END FUNCTION read_filesize

!-------------------------------------------------------------------------
!-------------------------------------------------------------------------
SUBROUTINE read_acsdata(filename, time, acsarr)
!  ====================================================================
! | This subroutine reads the data from a Fortran binary unformatted |
! | file. Data records are of acsdata_type. In order to know the |
! | size of the "data" array that is to receive the output of this |
! | subroutine, one should use the "read_filesize" function first. |
!  ====================================================================
USE source

character(len=*), intent(IN) :: filename
real(DP), intent(OUT) :: time
type(acsdata_type), dimension(:,,:), intent(OUT) :: acsarr
type(acsdata_type), dimension(:,), allocatable :: rowvector
integer, dimension(2) :: filesize
integer, dimension(4) :: comp_range
integer :: i, ierr, stat, timex, astat, j
integer, parameter :: iounit = 30

! Open the binary file.
open (UNIT=iounit, FILE=filename, STATUS='OLD', ACTION='READ', &
FORM='UNFORMATTED', IOSTAT(ierr))
if (ierr /= 0) then
    print *, cat(cat("Problem opening ", filename ), ".")
    STOP
end if

stat = 0
! Read filesize from the file
read(iounit, IOSTAT=stat) filesize
if (filesize(1) /= size(acsarr,1) .or. filesize(2) /= size(acsarr,2)) then
    write(*,"(a,i2,a,i2,a,i2,a,i2,a)") "You allocated a ",size(acsarr,1), &
    " by ",size(acsarr,2)," array, but want to read a ",filesize(1), &
    " by ",filesize(2)," data file. WTF!?!"
end if
allocate(rowvector(filesize(2)), STAT=astat)

! Read computational grid range.
read(iounit, IOSTAT=stat) comp_range
! Read time data from the file.
read(iounit, IOSTAT=stat) timex
read(iounit, IOSTAT=stat) time ! in seconds
! Read source amplitude and phase.
read(iounit, IOSTAT=stat) S_0
read(iounit, IOSTAT=stat) phi

! Read data from the file
do i = 1, filesize(1)
    ! do j = 1, filesize(2)
    ! read (iounit, IOSTAT=stat) acsarr(i,:
    ! read (iounit, IOSTAT=stat) rowvector
    if (stat /= 0) then
        print *, "Ran out of file records."
    end if
    end do
acsarr(i,:) = rowvector
end do

! Close file
close (iounit)
dallocate(rowvector, STAT=astat)
RETURN

END SUBROUTINE read_acsdata
SUBROUTINE fin_IO()
!
!跗 | Closes any files that were held open. |
!跗 | !

ifdef HAVE_MPI

USE mpi_fun
endif
implicit none

ifdef HAVE_MPI

! ** N.B. Only data from CPU node 0 is available for probe output. **
if (myid == MASTER) then
endif

close(prbiou)

endif

RETURN
END SUBROUTINE fin_IO
!

SUBROUTINE skipline(iounit, lines)
!
!跗 | Reads a number of lines in a formatted, sequentially accessed file |
!跗 | to position the file marker. |

integer, intent(IN) :: iounit
integer, intent(IN) :: lines
character(len=1) :: record
integer :: i, istat

do i = 1, lines
   read(UNIT=iounit, FMT="(a)", IOSTAT=istat) record
end do

RETURN
END SUBROUTINE skipline
SUBROUTINE skiptab(iounit, spaces)
!
! | Reads a number of characters in a formatted, sequentially accessed |
! | file to position the file marker. |
! |=======================================================================|
!
integer, intent(IN) :: iounit
integer, intent(IN) :: spaces
character(len=spaces) :: record
integer :: istat !, i

read(UNIT=iounit, FMT="(a)", ADVANCE='NO', IOSTAT=istat) record

RETURN

END SUBROUTINE skiptab
!
!=======================================================================
!
END MODULE IO_tools
MODULE maccormack_cyl
!===========================================================================!
! This module contains the subroutines for the MacCormack method of     
! finite difference approximation.                                      
! Table of contents:                                                     
! parameters A, B, switch                                               
! SUBROUTINE propagate_nl()                                             
! SUBROUTINE driver(timex)                                              
! SUBROUTINE FDiff(ver)                                                 
! SUBROUTINE FD_OPxi(ver)                                               
! SUBROUTINE FD_OPeta(ver)                                              
! SUBROUTINE mu_kappa()                                                 
!===========================================================================!
USE global
USE FDiff_vars
USE grid
implicit none
! -------------------------------------------------------------------
! Public routine
Public :: propagate_cyl
! -------------------------------------------------------------------
Private
integer, parameter :: A = 1, B = -1
integer, dimension(3), parameter :: switch = (/ B, 0, A /)
!
! Pointers to the generically-named metrics.
real(DP), dimension(:,:), pointer :: Xi_z, Eta_r
!
! Pointers to the generically-named grid arrays.
real(DP), dimension(:,,:), pointer :: tube_z, tube_r
!
! Pointer to gravity component.
real(DP), pointer :: g_z

#endif MAKE_LIN
! Space for the oft-repeated {v,u}_bar expression
real(DP), dimension(:,:), allocatable :: vel_bar
#endif

#endif MAKE_D1
! Memory for temperature gradients (ni,nj)
real(DP), dimension(:,:), allocatable :: t_xi
! Pointers to the generically-named metrics.
real(DP), dimension(:,,:), pointer :: Z_xi, R_eta
! Space for the oft-repeated xi_z*r_eta and eta_r*z_xi expressions
real(DP), dimension(:,,:), allocatable :: XIzReta, ETArZxi
! Ratios of gas parameters.
real(DP) :: Mu_Rho0, Kap_RhoT0
#endif

#ifdef MAKE_ND
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: rcp_cylR
! Memory for du/dz, dv/dz, du/dr, dv/dr, temperature gradient (ni,nj)
real(DP), dimension(:,,:), allocatable :: u_z, u_r, v_z, v_r
real(DP), dimension(:,,:), allocatable :: t_z, t_r, p_z, p_r
! Coefficients:
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
#endif

#ifdef MAKE_NDT
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: rcp_cylR
! Memory for du/dz, dv/dz, du/dr, dv/dr, temperature gradient (ni,nj)
real(DP), dimension(:,,:), allocatable :: u_z, u_r, v_z, v_r
real(DP), dimension(:,,:), allocatable :: t_z, t_r, p_z, p_r
! Coefficients:
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,,:), allocatable :: mu, kp
#endif

#ifdef MAKE_ND2
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: rcp_cylR
! Memory for du/dz, dv/dz, du/dr, dv/dr, temperature gradient (ni,nj)
real(DP), dimension(:,,:), allocatable :: u_z, u_r, v_z, v_r
real(DP), dimension(:,,:), allocatable :: t_z, t_r, p_z, p_r
! Coefficients:
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,,:), allocatable :: mu, kp
#endif

#ifdef MAKE_ND2T
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: rcp_cylR
! Memory for du/dz, dv/dz, du/dr, dv/dr, temperature gradient (ni,nj)
real(DP), dimension(:,,:), allocatable :: u_z, u_r, v_z, v_r
real(DP), dimension(:,,:), allocatable :: t_z, t_r, p_z, p_r
! Coefficients:
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,:), allocatable :: mu, kp
#endif
real(DP), dimension(:,:), allocatable :: OSD, BEQ
real(DP), dimension(:,:), allocatable :: L1, L2, L3, L4
#endif MAKE_LIN
#else
real(DP), dimension(:,:), allocatable :: L1p, L2p, L3p, L4p
#endif
CONTAINS
!---------------------------------------------------------------------------
SUBROUTINE propagate_cyl()
! =======================================================================
! | This subroutine manages the main time loop for the finite difference |
! | calculation. It keeps track of which version of the operator we're |
! | using. |
! =======================================================================
USE input_vars
USE IO_tools
USE boundary
USE source
USE analyt
implicit none
integer(i4b) :: ver, i
integer(i4b) :: timex
logical :: stopping
! Allocate variables in this module’s common block.
call alloc_FDvars()
call alloc_localvars()
! Declare pointers to the physical grid arrays tube_x1 and tube_x2.
tube_z => tube_x1
tube_r => tube_x2
! Declare pointer to gravity variable.
g_z => g_x1
! Declare pointers to Xi_x1, X1_xi, X2_eta, and Eta_x2
Xi_z => Xi_x1
Eta_r => Eta_x2
#endif MAKE_D1
Z_xi => X1_xi
R_eta => X2_eta
! Multiply metrics.
XIzReta = Xi_z * R_eta
ETArZxi = Eta_r * Z_xi
! Define ratios of gas parameters for dissipative terms.
Mu_Rho0 = mu0 / rho0
Kap_RhoT0 = k0 / (rho0 * T0)
#endif
#endif MAKE_ND
rcp_cylR = 1d0/tube_r
Fb2 = Jac * Xi_z / rho0
Fb3 = -Jac * Xi_z / rho0
Fb4 = Jac * Xi_z / (rho0 * Cv)
Gb2 = -Jac * Eta_r / rho0
Gb3 = Jac * Eta_r / rho0
Gb4 = Jac * Eta_r / (rho0 * Cv)
#endif MAKE_NDT
rcp_cylR = 1d0/tube_r
Fb2 = Jac * Xi_z / rho0
Fb3 = -Jac * Xi_z / rho0
Fb4 = Jac * Xi_z / (rho0 * Cv)
Gb2 = -Jac * Eta_r / rho0
Gb3 = Jac * Eta_r / rho0
Gb4 = Jac * Eta_r / (rho0 * Cv)
#endif MAKE_ND2
rcp_cylR = 1d0/tube_r
Fb2 = Jac * Xi_z / rho0
Fb3 = -Jac * Xi_z / rho0
Fb4 = Jac * Xi_z / (rho0 * Cv)
Gb2 = -Jac * Eta_r / rho0
Gb3 = Jac * Eta_r / rho0
Gb4 = Jac * Eta_r / (rho0 * Cv)
#endif MAKE_ND2T
rcp_cylR = 1d0/tube_r
Fb2 = Jac * Xi_z / rho0
Fb3 = -Jac * Xi_z / rho0
Fb4 = Jac * Xi_z / (rho0 * Cv)
Gb2 = -Jac * Eta_r / rho0
Gb3 = Jac * Eta_r / rho0
Gb4 = Jac * Eta_r / (rho0 * Cv)
#endif
! Define indices for central differencing.
i_p_one = i_main + 1
i_m_one = i_main - 1
j_p_one = j_main + 1
j_m_one = j_main - 1
! Transform variables to computational domain.
do i = 1,nACSvar
Wb(,:,i) = acs(,:,i) * Jac
end do

call BC_xi(Wb)
call BC_eta(Wb)

! Initialize the cumulative sum for streaming velocities.
if (streamout .ne. 'non') then
  sumacs = 0.0d0
  n_sum = 0
end if

stopping = .FALSE.

! Start time loop.
timex = txmin

! Set version to A (1) if timex is even, B (-1) if odd.
ver = -2*mod(timex,2) + 1

propagate: do

! This is the next time step.
timex = timex + 1

! Update source.
call ACSsrc(timex)

if (fdrv_type == 'PRS') then
  acs(,:,Px) = Wb(,:,Px) * iJac
  acs(,:,Tx) = Wb(,:,Tx) * iJac

  Wb(,:,Px) = Jac*(Psource * Gaussian_BL + acs(,:,Px) * iGaussian_BL)
  Wb(,:,Tx) = Jac*(Temperature_BL + acs(,:,Tx) * iGaussian_BL)

#ifdef MAKE_LIN
  Wb(,:,RHOx) = (Wb(,:,Px)/Rgas - Rho0*Wb(,:,Tx)) / T0
#else
  acs(,:,Tx) = Wb(,:,Tx) * iJac
  Wb(,:,RHOx) = (Wb(,:,Px)/Rgas - &
                  Rho0*Wb(,:,Tx)) / (T0 + acs(,:,Tx))
#endif

call BC_xi(Wb)
call BC_eta(Wb)
end if

! Do the finite difference operation for one time step.
call FDiff(ver)

! Contribute to cumulative sum for streaming calculation.
if (streamout .ne. 'non') then
  sumacs(:,:,1:nACSvar) = sumacs(:,:,1:nACSvar) + acs
sumacs(:,:,nACSvar+1) = sumacs(:,:,nACSvar+1) + &
    acs(:,:,RHOx)*acs(:,:,Ux)
sumacs(:,:,nACSvar+2) = sumacs(:,:,nACSvar+2) + &
    acs(:,:,RHOx)*acs(:,:,Vx)
    n_sum = n_sum + 1
end if

! Calculate maximum volume velocity and write out for each period.
call max_Uvol(timex)

! Write a file every so often.
call fileout(timex)

! End time loop.
call test_for_doneness(timex, stopping)
if (stopping) exit

!! Get ready for the next propagation step.
! Switch the version. (switch = [-1 0 1])
    ver = switch(2-ver) ! use the index of switch to find the next ver
end do propagate
    txmax = timex

! Clean up FDvars.
call dealloc_localvars()
call dealloc_FDvars()

RETURN
END SUBROUTINE propagate_cyl

!---------------------------------------------------------------------------

SUBROUTINE test_for_doneness(time_inx, stop_flag)
! =========================================================================
! | This subroutine decides when to stop by measuring the radial variation |
! | in the acoustic pressure variable at L/4 in the axial direction for a |
! | 1-D calculation of a standing wave. |
! =========================================================================

implicit none

integer, intent(IN) :: time_inx
logical, intent(OUT) :: stop_flag

if ( time_inx .ge. txmax ) stop_flag = .TRUE.

END SUBROUTINE test_for_doneness
!---------------------------------------------------------------------------
SUBROUTINE FDiff(ver)
!
! This subroutine organizes the alternate versions and ordering of the finite difference operators in the z and r directions.
!
! 28nov01 btuttle created
! 04nov03 btuttle changed to xi and eta directions
!
!=================================================================

implicit none

integer, intent(in) :: ver

if ( ver == A ) then
! Xi-direction
 call FD_OPxi(ver)
! Eta-direction
 call FD_OPeta(ver)
else if ( ver == B ) then
! Eta-direction
 call FD_OPeta(ver)
! Xi-direction
 call FD_OPxi(ver)
else
 print *, "Version is neither A nor B. Goodbye."
 STOP
end if

RETURN

END SUBROUTINE FDiff

!=================================================================

SUBROUTINE FD_OPxi(ver)
!
! This subroutine computes the finite difference in the xi-direction.
!
!=================================================================

USE boundary
USE source

implicit none

integer, intent(in) :: ver

integer, dimension(ni_main) :: i_p_ver, i_m_ver
integer :: q

#ifdef MAKE_D1
    integer, dimension(nj_main) :: j_p_ver, j_m_ver
#endif

real(DP), dimension(:,,:), pointer :: Fbar, SFbar

! Declare pointers to FGbar, SFGbar so they can be aptly named Fbar, SFbar.
Fbar => FGbar
SFbar => SFGbar

! Define shifted index vectors for the main and secondary acs vars arrays.
i_p_ver = i_main+ver ! The i_main index vector shifted one
i_m_ver = i_main-ver ! bin to the right or left.

#ifdef MAKE_D1
    j_p_ver = j_main+ver ! The j_main index vector shifted one
    j_m_ver = j_main-ver ! bin to the right or left.
#endif

! Eventually use the fourth order scheme over the inner domain.

! Use the second order scheme over the whole domain for now.

! Calculate these repeated expressions once.
#ifdef MAKE_LIN
    ! vel_bar is (ni,nj).
    vel_bar = Xi_z * Wb(:,:,Ux) ! + Xi_r * Wb(:,:,3)
#endif

#ifdef MAKE_ND
    VoR = acs(:,:,Vx)*rcp_cylR

    ! Z-derivatives are "rearward" diff wrt "forward" diff for predictor.
    u_z(i_main,:) = Xi_z(i_main,:) * &
                  ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
    v_z(i_main,:) = Xi_z(i_main,:) * &
                  ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
    t_z(i_main,:) = Xi_z(i_main,:) * &
                  ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))

    call BCgrad_xi(u_z, v_z, t_z)

    ! R-derivatives are center differenced.
    u_r(:,j_main) = Eta_r(:,j_main) * &
                  0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
    v_r(:,j_main) = Eta_r(:,j_main) * &
                  0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
    t_r(:,j_main) = Eta_r(:,j_main) * &
                  0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
#endif
p_r(:,j_main) = Eta_r(:,j_main) * &
0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))

call BCgrad_eta(u_r, v_r, t_r, p_r)
#endif
#endif MAKE_NDT
VoR = acs(:,Vx)*rcp_cylR

! Z-derivatives are "rearward" diff wrt "forward" diff for predictor.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                 0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                 0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
  t_r(:,j_main) = Eta_r(:,j_main) * &
                 0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))

call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
  call mu_kappa()
#endif
#endif MAKE_ND2
VoR = acs(:,Vx)*rcp_cylR

! Z-derivatives are "rearward" diff wrt "forward" diff for predictor.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
                 ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                 0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                 0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_r(:,j_main) = Eta_r(:,j_main) * 
               0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))

p_r(:,j_main) = Eta_r(:,j_main) * 
               0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()
#endif
#ifdef MAKE_ND2T
    VoR = acs(:,:,Vx)*rcp_cylR

! Z-derivatives are "rearward" diff wrt "forward" diff for predictor.
    u_z(i_main,:) = Xi_z(i_main,:) * &
                   ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
    v_z(i_main,:) = Xi_z(i_main,:) * &
                   ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
    t_z(i_main,:) = Xi_z(i_main,:) * &
                   ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))
call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
    u_r(:,j_main) = Eta_r(:,j_main) * &
                   0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
    v_r(:,j_main) = Eta_r(:,j_main) * &
                   0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
    t_r(:,j_main) = Eta_r(:,j_main) * &
                   0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
    p_r(:,j_main) = Eta_r(:,j_main) * &
                   0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()
#endif

! Calculate F1, F2, F3, and F4 for the first step of the MacCormack method.
#ifdef MAKE_LIN
    Fbar(:,:,1) = rho0 * vel_bar
    Fbar(:,:,2) = Xi_z * Wb(:,:,Px) / rho0
    Fbar(:,:,3) = 0.0d0
    Fbar(:,:,4) = PoRHOCv0 * Wb(:,:,Ux) * Xi_z
#endif
#ifdef MAKE_D1
    Fbar(:,:,1) = rho0 * vel_bar
    Fbar(:,:,2) = (Xi_z*Wb(:,:,5) - mu0*(2.0d0*Vgrad(:,:,Uxi)*XIzReta - &
Vgrad(:,:,Veta) - Xi_z*Wb(:,:,3)/tube_r)) / rho0
Fbar(:,:,3) = -Mu_Rho0*(Vgrad(:,:,Ueta) + Vgrad(:,:,Vxi)*XIzReta)
Fbar(:,:,4) = -Kap_RhoT0*t_xi*XIzReta
#endif

#ifdef MAKE_ND
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_z
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu0*(2d0*u_z - VoR - v_r))
Fbar(:,:,3) = Fb3*mu0*(v_z + u_r)
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - k0*t_z)
#endif

#ifdef MAKE_NDT
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_z
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
Fbar(:,:,3) = Fb3*mu*(v_z + u_r)
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_z)
#endif

#ifdef MAKE_ND2
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_z
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
Fbar(:,:,3) = Fb3*mu*(v_z + u_r)
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_z)
#endif

#ifdef MAKE_ND2T
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_z
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
Fbar(:,:,3) = Fb3*mu*(v_z + u_r)
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_z)
#endif

! Calculate SF term for first part of MacCormack method.
! SFbar is (ni_main,nj_main,4)
#ifdef MAKE_LIN
SFbar(:,:,1) = 0.0d0 + Density_src*Jac
SFbar(:,:,2) = Jac * (g_z + gdrv_x1)
SFbar(:,:,3) = 0.0d0
SFbar(:,:,4) = 0.0d0 + Temperature_src*Jac
#endif

#ifdef MAKE_D1
SFbar(:,:,1) = 0.0d0
SFbar(:,:,2) = Mu_Rho0 * Vgrad(i_main,j_main,Vxi) * R_eta(i_main,j_main) & + (Jac(i_main,j_main) + Wb(i_main,j_main,1)/rho0) * g_z
SFbar(:,:,3:4) = 0.0d0
#endif

#ifdef MAKE_ND
SFbar(:,:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,:,RHOx))*VoR)
SFbar(:,:,2) = Jac*((mu0*rcp_cylR - rho0*acs(:,:,Vx)) * & u_r*ONEoRHO0 + g_z + gdrv_x1)
SFbar(:,:,3) = Jac * (acs(:,:,RHOx)*p_r*ONEoRHO0 + &
(2d0*v_r - VoR)*mu0*rcp_cylR - &
rho0*acs(:,Vx)*v_r)*ONEoRHO0
SFbar(:,4) = Jac * (((P0*acs(:,RHOx)*ONEoRHO0 - acs(:,Px)) * &
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + &
(k0*rcp_cylR - Cv*rho0*acs(:,Vx))*t_r)*ONEoRHOCv0 + &
Temperature_src)
#endif
#endif MAKE_NDT
! Full nonlinear, dissipative equations minus O(\epsilon^3) and
! O(\mu\epsilon^2) terms
SFbar(:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,RHOx))*VoR)
SFbar(:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,Vx)) * &
 u_r*ONEoRHO0 + g_z + gdrv_x1)
SFbar(:,3) = Jac * (acs(:,RHOx)*p_r*ONEoRHO0 + &
(2d0*v_r - VoR)*mu*rcp_cylR - &
rho0*acs(:,Vx)*v_r)*ONEoRHO0
SFbar(:,4) = Jac * (((P0*acs(:,RHOx)*ONEoRHO0 - acs(:,Px)) * &
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + &
(kp*rcp_cylR - Cv*rho0*acs(:,Vx))*t_r + &
mu0*(twothds*((2.0d0*(v_r - VoR) - u_z)*v_r + VoR*VoR) + &
(v_z + u_r)*u_r))*ONEoRHOCv0 + Temperature_src)
#endif
#endif MAKE_ND2
SFbar(:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,RHOx))*VoR)
SFbar(:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,Vx)) * &
 u_r*ONEoRHO0 + g_z + gdrv_x1)
SFbar(:,3) = Jac * (acs(:,RHOx)*p_r*ONEoRHO0 + &
(2d0*v_r - VoR)*mu*rcp_cylR - &
rho0*acs(:,Vx)*v_r)*ONEoRHO0
SFbar(:,4) = Jac * (((P0*acs(:,RHOx)*ONEoRHO0 - acs(:,Px)) * &
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + &
(kp*rcp_cylR - Cv*rho0*acs(:,Vx))*t_r + &
mu0*(twothds*((2.0d0*(v_r - VoR) - u_z)*v_r + VoR*VoR) + &
(v_z + u_r)*u_r))*ONEoRHOCv0 + Temperature_src)
#endif
#endif MAKE_ND2T
SFbar(:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,RHOx))*VoR)
SFbar(:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,Vx)) * &
 u_r*ONEoRHO0 + g_z + gdrv_x1)
SFbar(:,3) = Jac * (acs(:,RHOx)*p_r*ONEoRHO0 + &
(2d0*v_r - VoR)*mu*rcp_cylR - &
rho0*acs(:,Vx)*v_r)*ONEoRHO0
SFbar(:,4) = Jac * (((P0*acs(:,RHOx)*ONEoRHO0 - acs(:,Px)) * &
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + &
(kp*rcp_cylR - Cv*rho0*acs(:,Vx))*t_r + &
mu0*(twothds*((2.0d0*(v_r - VoR) - u_z)*v_r + VoR*VoR) + &
(v_z + u_r)*u_r))*ONEoRHOCv0 + Temperature_src)
#endif

! Do the first step of MacCormack method.
Wbp(i_main,:,1:4) = Wb(i_main,:,1:4) - deltat * (ver * &
(Fbar(i_p_ver,:,1:4) - Fbar(i_main,:,1:4)) - &
SFbar(i_main,:,1:4))

#ifdef MAKE_LIN
    Wbp(:,Px) = Rgas * (rho0*Wbp(:,Tx) + T0*Wbp(:,RHOx))
#else
    Wbp(:,Px) = Rgas* (rho0 + Wbp(:,RHOx) * iJac) * Wbp(:,Tx) + & 
                  T0*Wbp(:,RHOx) )
#endif

! Use boundary condition to set left and right boundary points.
call BCpred_xi()
call BC_eta(Wbp)

#ifdef MAKE_ND
! Update acs variables before the next step
  do q = 1, nACSvar
    acs(:,q) = Wbp(:,q) * iJac
  end do
#endif
#ifdef MAKE_NDT
! Update acs variables before the next step
  do q = 1, nACSvar
    acs(:,q) = Wbp(:,q) * iJac
  end do
#endif
#ifdef MAKE_ND2
! Update acs variables before the next step
  do q = 1, nACSvar
    acs(:,q) = Wbp(:,q) * iJac
  end do
#endif
#ifdef MAKE_ND2T
! Update acs variables before the next step
  do q = 1, nACSvar
    acs(:,q) = Wbp(:,q) * iJac
  end do
#endif
#ifdef MAKE_LIN
! Precalculate expressions for second step.
    vel_bar = Xi_z * Wbp(:,Ux) !+ Xi_r * Wbp(:,3)
#endif
#ifdef MAKE_ND
    VoR = acs(:,Vx)*rcp_cylR
! Z-derivatives are "forward" diff wrt "rearward" diff for corrector.
    u_z(i_main,:) = Xi_z(i_main,:) * & 
                    ver * (acs(i_p_ver,:,Ux) - acs(i_main,:,Ux))
    v_z(i_main,:) = Xi_z(i_main,:) * & 
                    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
\[ t_z(i_{\text{main}},:) = \frac{\text{Xi}_z(i_{\text{main}},:) \cdot \text{ver} \cdot (\text{acs}(i_{\text{p\_ver}},:,\text{Tx}) - \text{acs}(i_{\text{main}},:,\text{Tx}))}{\text{acs}(i_{\text{main}},:,\text{Ux}) - \text{acs}(i_{\text{main}},:,\text{Vx})} \]

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
\[ u_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Ux}) - \text{acs}(:,j_{\text{m\_one}},\text{Ux}))}{\text{acs}(:,j_{\text{p\_one}},\text{Vx}) - \text{acs}(:,j_{\text{m\_one}},\text{Vx})} \]
\[ v_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Vx}) - \text{acs}(:,j_{\text{m\_one}},\text{Vx}))}{\text{acs}(:,j_{\text{p\_one}},\text{Tx}) - \text{acs}(:,j_{\text{m\_one}},\text{Tx})} \]
\[ t_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Tx}) - \text{acs}(:,j_{\text{m\_one}},\text{Tx}))}{\text{acs}(:,j_{\text{p\_one}},\text{Px}) - \text{acs}(:,j_{\text{m\_one}},\text{Px})} \]

call BCgrad_eta(u_r, v_r, t_r, p_r)

#ifdef MAKE_NDT
VoR = \text{acs}(:,:,\text{Vx})\text{rcp\_cylR}

! Z-derivatives are "forward" diff wrt "rearward" diff for corrector.
\[ u_z(i_{\text{main}},:) = \frac{\text{Xi}_z(i_{\text{main}},:) \cdot \text{ver} \cdot (\text{acs}(i_{\text{p\_ver}},:,\text{Ux}) - \text{acs}(i_{\text{main}},:,\text{Ux}))}{\text{acs}(i_{\text{main}},:,\text{Ux}) - \text{acs}(i_{\text{main}},:,\text{Vx})} \]
\[ v_z(i_{\text{main}},:) = \frac{\text{Xi}_z(i_{\text{main}},:) \cdot \text{ver} \cdot (\text{acs}(i_{\text{p\_ver}},:,\text{Vx}) - \text{acs}(i_{\text{main}},:,\text{Vx}))}{\text{acs}(i_{\text{main}},:,\text{Vx}) - \text{acs}(i_{\text{main}},:,\text{Tx})} \]
\[ t_z(i_{\text{main}},:) = \frac{\text{Xi}_z(i_{\text{main}},:) \cdot \text{ver} \cdot (\text{acs}(i_{\text{p\_ver}},:,\text{Tx}) - \text{acs}(i_{\text{main}},:,\text{Tx}))}{\text{acs}(i_{\text{main}},:,\text{Tx}) - \text{acs}(i_{\text{main}},:,\text{Px})} \]

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
\[ u_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Ux}) - \text{acs}(:,j_{\text{m\_one}},\text{Ux}))}{\text{acs}(:,j_{\text{p\_one}},\text{Vx}) - \text{acs}(:,j_{\text{m\_one}},\text{Vx})} \]
\[ v_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Vx}) - \text{acs}(:,j_{\text{m\_one}},\text{Vx}))}{\text{acs}(:,j_{\text{p\_one}},\text{Tx}) - \text{acs}(:,j_{\text{m\_one}},\text{Tx})} \]
\[ t_r(:,j_{\text{main}}) = \frac{\text{Eta}_r(:,j_{\text{main}}) \cdot 0.5d0 \cdot (\text{acs}(:,j_{\text{p\_one}},\text{Tx}) - \text{acs}(:,j_{\text{m\_one}},\text{Tx}))}{\text{acs}(:,j_{\text{p\_one}},\text{Px}) - \text{acs}(:,j_{\text{m\_one}},\text{Px})} \]

call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()
#endif

#ifdef MAKE_KAPPA
VoR = \text{acs}(:,:,\text{Vx})\text{rcp\_cylR}

! Z-derivatives are "forward" diff wrt "rearward" diff for corrector.
\[ u_z(i_{\text{main}},:) = \frac{\text{Xi}_z(i_{\text{main}},:) \cdot \text{ver} \cdot (\text{acs}(i_{\text{p\_ver}},:,\text{Ux}) - \text{acs}(i_{\text{main}},:,\text{Ux}))}{\text{acs}(i_{\text{main}},:,\text{Ux}) - \text{acs}(i_{\text{main}},:,\text{Vx})} \]
v_z(i_main,:) = Xi_z(i_main,:) * &
    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
t_z(i_main,:) = Xi_z(i_main,:) * &
    ver * (acs(i_p_ver,:,Tx) - acs(i_main,:,Tx))

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
u_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))

call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
    call mu_kappa()
#endif
#ifdef MAKE_ND2T
VoR = acs(:,:,Vx)*rcp_cylR

! Z-derivatives are "forward" diff wrt "rearward" diff for corrector.
v_z(i_main,:) = Xi_z(i_main,:) * &
    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
t_z(i_main,:) = Xi_z(i_main,:) * &
    ver * (acs(i_p_ver,:,Tx) - acs(i_main,:,Tx))

call BCgrad_xi(u_z, v_z, t_z)

! R-derivatives are center differenced.
u_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_r(:,j_main) = Eta_r(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))

call BCgrad_eta(u_r, v_r, t_r, p_r)

! Calculate temperature dependent viscosity and thermal conductivity
    call mu_kappa()
#endif
! Calculate Fbar for the second step of the MacCormack method.
#endif MAKE_LIN
    Fbar(:,1) = rho0 * vel_bar
    Fbar(:,2) = Xi_z * Wbp(:,Px) / rho0
    Fbar(:,3) = 0.0d0
    Fbar(:,4) = PoRHOCv0 * Wbp(:,Ux) * Xi_z
#endif

#ifdef MAKE_D1
    Fbar(:,1) = rho0 * vel_bar
    Fbar(:,2) = (Xi_z*Wbp(:,5) - mu0*(2.0d0*Vgrad(:,Uxi)*XIzReta - &
                                Vgrad(:,Veta) - Xi_z*Wbp(:,3)/tube_r)) / rho0
    Fbar(:,3) = -Mu_Rho0*(Vgrad(:,Ueta) + Vgrad(:,Vxi)*XIzReta)
    Fbar(:,4) = -Kap_RhoT0*t_xi*XIzReta
#endif

#ifdef MAKE_ND
    Fbar(:,1) = (rho0 + acs(:,RHOx)) * Wbp(:,Ux) * Xi_z
    Fbar(:,2) = Fb2*(acs(:,Px) - twothds*mu0*(2d0*u_z - VoR - v_r))
    Fbar(:,3) = Fb3*mu0*(v_z + u_r)
    Fbar(:,4) = Fb4 * (P0*acs(:,Ux) - k0*t_z)
#endif

#ifdef MAKE_NDT
    Fbar(:,1) = (rho0 + acs(:,RHOx)) * Wbp(:,Ux) * Xi_z
    Fbar(:,2) = Fb2*(acs(:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
    Fbar(:,3) = Fb3*mu*(v_z + u_r)
    Fbar(:,4) = Fb4 * (P0*acs(:,Ux) - kp*t_z)
#endif

#ifdef MAKE_ND2
    Fbar(:,1) = (rho0 + acs(:,RHOx)) * Wbp(:,Ux) * Xi_z
    Fbar(:,2) = Fb2*(acs(:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
    Fbar(:,3) = Fb3*mu*(v_z + u_r)
    Fbar(:,4) = Fb4 * (P0*acs(:,Ux) - kp*t_z)
#endif

#ifdef MAKE_ND2T
    Fbar(:,1) = (rho0 + acs(:,RHOx)) * Wbp(:,Ux) * Xi_z
    Fbar(:,2) = Fb2*(acs(:,Px) - twothds*mu*(2d0*u_z - VoR - v_r))
    Fbar(:,3) = Fb3*mu*(v_z + u_r)
    Fbar(:,4) = Fb4 * (P0*acs(:,Ux) - kp*t_z)
#endif

! Calculate SFbar term for second part of MacCormack method.
#endif MAKE_LIN
    SFbar(:,1) = 0.0d0 + Density_src*Jac
    SFbar(:,2) = Jac * (g_z + gdrv_x1)
    SFbar(:,3) = 0.0d0
    SFbar(:,4) = 0.0d0 + Temperature_src*Jac
#endif

#ifdef MAKE_D1
    SFbar(:,1) = 0.0d0
SFbar(:,:,2) = Mu_Rho0 * Vgrad(i_main,j_main,Vxi) * R_eta(i_main,j_main) & 
+ (Jac(i_main,j_main) + Wbp(i_main,j_main,1)/rho0) * g_z
SFbar(:,:,3:4) = 0.0d0
#endif

#endif MAKE_ND
SFbar(:,:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,:,RHOx))*VoR)
SFbar(:,:,2) = Jac*((mu0*rcp_cylR - rho0*acs(:,:,Vx)) * & 
u_r*ONEoRHO0 + g.z + gdrv.x1)
SFbar(:,:,3) = Jac * (acs(:,:,RHOx)*p_r*ONEoRHO0 + & 
(2d0*v.r - VoR)*mu0*rcp_cylR - & 
rho0*acs(:,:,Vx)*v_r)*ONEoRHO0
SFbar(:,:,4) = Jac * (((P0*acs(:,:,RHOx)*ONEoRHO0 - acs(:,:,Px)) * & 
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + & 
(k0*rcp_cylR - Cv*rho0*acs(:,:,Vx))*t_r)*ONEoRHO0 + 
Temperature_src)
#endif

#endif MAKE_NDT
! Full nonlinear, dissipative equations minus O(\epsilon^3) and ! O(\mu\epsilon^2) terms
SFbar(:,:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,:,RHOx))*VoR)
SFbar(:,:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,:,Vx)) * & 
u_r*ONEoRHO0 + g.z + gdrv.x1)
SFbar(:,:,3) = Jac * (acs(:,:,RHOx)*p_r*ONEoRHO0 + & 
(2d0*v.r - VoR)*mu*rcp_cylR - & 
rho0*acs(:,:,Vx)*v_r)*ONEoRHO0
SFbar(:,:,4) = Jac * (((P0*acs(:,:,RHOx)*ONEoRHO0 - acs(:,:,Px)) * & 
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + & 
(kp*rcp_cylR - Cv*rho0*acs(:,:,Vx))*t_r)*ONEoRHO0 + & 
Temperature_src)
#endif

#endif MAKE_ND2
SFbar(:,:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,:,RHOx))*VoR)
SFbar(:,:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,:,Vx)) * & 
u_r*ONEoRHO0 + g.z + gdrv.x1)
SFbar(:,:,3) = Jac * (acs(:,:,RHOx)*p_r*ONEoRHO0 + & 
(2d0*v.r - VoR)*mu*rcp_cylR - & 
rho0*acs(:,:,Vx)*v_r)*ONEoRHO0
SFbar(:,:,4) = Jac * (((P0*acs(:,:,RHOx)*ONEoRHO0 - acs(:,:,Px)) * & 
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + & 
(kp*rcp_cylR - Cv*rho0*acs(:,:,Vx))*t_r + & 
mu0*(twothds*((2.0d0*(v.r - VoR) - u.z)*v_r + VoR*VoR) + & 
(v.z + u.r)*u.r))*ONEoRHO0 + Temperature_src)
#endif

#endif MAKE_ND2T
SFbar(:,:,1) = Jac*(Density_src - 0.5d0*(rho0+acs(:,:,RHOx))*VoR)
SFbar(:,:,2) = Jac*((mu*rcp_cylR - rho0*acs(:,:,Vx)) * & 
u_r*ONEoRHO0 + g.z + gdrv.x1)
SFbar(:,:,3) = Jac * (acs(:,:,RHOx)*p_r*ONEoRHO0 + & 
(2d0*v.r - VoR)*mu*rcp_cylR - & 
rho0*acs(:,:,Vx)*v_r)*ONEoRHO0
SFbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - acs(:,:,Px)) * &
(0.5d0*VoR + v_r) - 0.5d0*P0*VoR + &
(kp*rcp_cylR - Cv*rho0*acs(:,:,Vx))*t_r + &
mu*(twothds*((2.0d0*(v_r - VoR) - u_z)*v_r + VoR*VoR) + &
(v_z + u_r)*u_r))*ONEoRHOCv0 + Temperature_src)
#endif

! Do the second step of MacCormack method.
Wb(i_main,:,1:4) = 0.5d0 * (Wb(i_main,:,1:4) + &
Wbp(i_main,:,1:4) - deltat*(ver * (Fbar(i_main,:,1:4) - &
Fbar(i_m_ver,:,1:4)) - SFbar(i_main,:,1:4)))
#ifdef MAKE_LIN
Wb(:,:,Px) = Rgas * (rho0*Wb(:,:,Tx) + T0*Wb(:,:,RHOx))
#else
Wb(:,:,Px) = Rgas* (rho0 + Wb(:,:,RHOx) * iJac) * Wb(:,:,Tx) + &
T0*Wb(:,:,RHOx) )
#endif

! Use boundary condition to set left and right boundary points.
call BCcorr_xi()
call BC_eta(Wb)
#ifdef MAKE_ND
    do q = 1, nACSvar
        acs(:,q) = Wb(:,q) * iJac
    end do
#endif
#ifdef MAKE_NDT
    do q = 1, nACSvar
        acs(:,q) = Wb(:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2
    do q = 1, nACSvar
        acs(:,q) = Wb(:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2T
    do q = 1, nACSvar
        acs(:,q) = Wb(:,q) * iJac
    end do
#endif
RETURN

END SUBROUTINE FD_OPxi

---------------------------------------------------------------------------
---------------------------------------------------------------------------
SUBROUTINE FD_OPeta(ver)
USE boundary
USE source

implicit none

integer, intent(in) :: ver
integer, dimension(nj_main) :: j_p_ver, j_m_ver
integer :: q

#ifdef MAKE_D1
integer, dimension(ni_main) :: i_p_ver, i_m_ver
#endif

real(DP), dimension(:,:,:), pointer :: Gbar, SGbar

! Point Gbar and SGbar at FGbar and SFGbar so their names refer to functions.
Gbar => FGbar
SGbar => SFGbar

! Define shifted index vectors for the main and secondary acs vars arrays.
j_p_ver = j_main+ver ! The j_main index vector shifted one
j_m_ver = j_main-ver ! bin up or down.

#ifdef MAKE_D1
i_p_ver = i_main+ver ! The i_main index vector shifted one
i_m_ver = i_main-ver ! bin to the right or left.
#endif

! Eventually use the fourth order scheme over the inner domain.
! Use the second order scheme over the whole domain for now.

#ifdef MAKE_LIN
vel_bar is (ni,nj).
vel_bar = Eta_r * Wb(,:,:)*Vx + Eta_z * Wb(,:,:)*2
#endif

#ifdef MAKE_ND
VoR = acs(,:,:)*rcp_cylR

! R-derivatives are "rearward" diff wrt "forward" diff for predictor.
u_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_main,Ux) - acs(:,j_m_ver,Ux))
v_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_main,Vx) - acs(:,j_m_ver,Vx))
t_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_main,Tx) - acs(:,j_m_ver,Tx))
#endif
call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
  p_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_z, v_z, t_z, p_z)
#endif
#endif MAKE_NDT
  VoR = acs(:,;Vx)*rcp_cylR

! R-derivatives are "rearward" diff wrt "forward" diff for predictor.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,;j_main,Ux) - acs(:,;j_m_ver,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,;j_main,Vx) - acs(:,;j_m_ver,Vx))
  t_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,;j_main,Tx) - acs(:,;j_m_ver,Tx))

call BCgrad_eta(u_r, v_r, t_r)
#endif
#endif MAKE_ND2
  VoR = acs(:,;Vx)*rcp_cylR

! Z-derivatives are center differenced.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
  p_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_z, v_z, t_z, p_z)

! Calculate temperature dependent viscosity and thermal conductivity
  call mu_kappa()
#endif
#endif MAKE_ND2
  VoR = acs(:,;Vx)*rcp_cylR

! R-derivatives are "rearward" diff wrt "forward" diff for predictor.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,;j_main,Ux) - acs(:,;j_m_ver,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,;j_main,Vx) - acs(:,;j_m_ver,Vx))
\[ t_r(:,j_{main}) = \text{Eta}_r(:,j_{main}) \times \text{ver} \times (\text{acs}(:,j_{main},Tx) - \text{acs}(:,j_{m\_ver},Tx)) \]

call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.
\[ u_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Ux) - \text{acs}(i_{m\_one},:,Ux)) \]
\[ v_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Vx) - \text{acs}(i_{m\_one},:,Vx)) \]
\[ t_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Tx) - \text{acs}(i_{m\_one},:,Tx)) \]
\[ p_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Px) - \text{acs}(i_{m\_one},:,Px)) \]

call BCgrad_xi(u_z, v_z, t_z, p_z)

! Calculate temperature dependent viscosity and thermal conductivity
\[ \text{call mu\_kappa()} \]

#ifdef MAKE_ND2T
VoR = \text{acs}(::,Vx)*rcp\_cylR

! R-derivatives are "rearward" diff wrt "forward" diff for predictor.
\[ u_r(:,j_{main}) = \text{Eta}_r(:,j_{main}) \times \text{ver} \times (\text{acs}(:,j_{main},Ux) - \text{acs}(:,j_{m\_ver},Ux)) \]
\[ v_r(:,j_{main}) = \text{Eta}_r(:,j_{main}) \times \text{ver} \times (\text{acs}(:,j_{main},Vx) - \text{acs}(:,j_{m\_ver},Vx)) \]
\[ t_r(:,j_{main}) = \text{Eta}_r(:,j_{main}) \times \text{ver} \times (\text{acs}(:,j_{main},Tx) - \text{acs}(:,j_{m\_ver},Tx)) \]

call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.
\[ u_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Ux) - \text{acs}(i_{m\_one},:,Ux)) \]
\[ v_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Vx) - \text{acs}(i_{m\_one},:,Vx)) \]
\[ t_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Tx) - \text{acs}(i_{m\_one},:,Tx)) \]
\[ p_z(i_{main}, :) = \text{Xi}_z(i_{main}, :) \times 0.5d0 \times (\text{acs}(i_{p\_one},:,Px) - \text{acs}(i_{m\_one},:,Px)) \]

call BCgrad_xi(u_z, v_z, t_z, p_z)

! Calculate temperature dependent viscosity and thermal conductivity
\[ \text{call mu\_kappa()} \]

#endif

! Calculate Gbar for the first step of the MacCormack method.
#ifdef MAKE_LIN
Gbar(:,:,1) = rho0 * vel_bar
Gbar(:,:,2) = 0.0d0
Gbar(:,:,3) = Eta_r * Wb(:,:,Px) / rho0
Gbar(:,:,4) = PoRHOCv0 * Wb(:,:,Vx) * Eta_r
#endif

#ifdef MAKE_D1
Gbar(:,:,1) = rho0 * vel_bar
Gbar(:,:,2) = -Mu_Rho0*(Vgrad(:,:,Ueta)*ETArZxi + &
                 Vgrad(:,:,Vxi))
Gbar(:,:,3) = ( Wb(:,:,5)*Eta_r &
             - twothds*mu0*(2.0d0*Vgrad(:,:,Veta)*ETArZxi - Vgrad(:,:,Uxi) &
                             - Eta_r*Wb(:,:,3)/tube_r) ) / rho0
Gbar(:,:,4) = -Kap_RhoT0*t_eta*ETArZxi
#endif

#ifdef MAKE_ND
Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Vx) * Eta_r
Gbar(:,:,2) = Gb2*mu0*(v_z + u_r)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu0*(2.0d0*v_r - VoR - u_z))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - k0*t_r)
#endif

#ifdef MAKE_NDT
Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Vx) * Eta_r
Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*t_r)
#endif

#ifdef MAKE_ND2
Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Vx) * Eta_r
Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*t_r)
#endif

#ifdef MAKE_NDT2
Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Vx) * Eta_r
Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*t_r)
#endif

! Calculate SGbar term for first part of MacCormack method.
#ifdef MAKE_LIN
SGbar(:,:,1) = -rho0 * Wb(:,:,Vx) / tube_r
SGbar(:,:,2:3) = 0.0d0
SGbar(:,:,4) = -PoRHOCv0 * Wb(:,:,Vx) / tube_r
#endif

#ifdef MAKE_D1
SGbar(:,:,1) = -rho0 * Wb(i_main,j_main,3) / tube_r(i_main,j_main)
SGbar(:,:,2) = Mu_Rho0 * Vgrad(i_main,j_main,Ueta) * Z_xi(i_main,j_main)
SGbar(:,:,3) = 2.0d0 * Mu_Rho0 * (Vgrad(i_main,j_main,Veta)*Z_xi &
   - Wb(i_main,j_main,3)/tube_r(i_main,j_main)) &
   / tube_r(i_main,j_main)
SGbar(:,:,4) = Kap_RhoT0 * t_\eta(i_main,j_main) * Z_xi(i_main,j_main) &
   / tube_r(i_main,j_main)
#endif
#endif
if defined MAKE_ND
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
   mu0*v_z*rcp_cylR - &
   rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu0*VoR*rcp_cylR - &
   rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
   acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
   Cv*rho0*acs(:,:,Ux)*t_z)*ONEoRHO0
#endif
#endif
if defined MAKE_NDT
! Full nonlinear, dissipative equations minus O(\epsilon^3) terms
! 0(\mu'\epsilon^2) terms
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
   mu*v_z*rcp_cylR - &
   rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu*VoR*rcp_cylR - &
   rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
   acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
   Cv*rho0*acs(:,:,Ux)*t_z)*ONEoRHO0
#endif
#endif
if defined MAKE_ND2
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
   mu*v_z*rcp_cylR - &
   rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu*VoR*rcp_cylR - &
   rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
   acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
   Cv*rho0*acs(:,:,Ux)*t_z + &
   mu0*((twothds*((2.0d0*(u_z - VoR) - v_r)*u_z + VoR*VoR) + &
   (v_z + u_r)*v_z))*ONEoRHO0
#endif
#endif
if defined MAKE_ND2T
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
   mu*v_z*rcp_cylR - &
   rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu*VoR*rcp_cylR - &
   rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
   acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
   Cv*rho0*acs(:,:,Ux)*t_z + &
   mu0*((twdths*(2.0d0*(u_z - VoR) - v_r)*u_z + VoR*VoR) + &
   (v_z + u_r)*v_z))*ONEoRHO0
\[ \rho_0 \cdot \acs(:,U_x) \cdot v_z \cdot \text{ONEoRHO0} \]

\[ \text{SGbar}(:,4) = \text{Jac} \cdot \left( \left( P_0 \cdot \acs(:,RHOx) \cdot \text{ONEoRHO0} - \acs(:,Px) \right) \cdot (u_z + 0.5d0 \cdot V_oR) - 0.5d0 \cdot P_0 \cdot V_oR - \mu \cdot \left( \text{twothds} \cdot (2.0d0 \cdot (u_z - V_oR) - v_r) \cdot u_z + V_oR \cdot V_oR \right) + (v_z + u_r) \cdot v_z \right) \cdot \text{ONEoRHOCv0} \]

\#endif

! Do the first step of MacCormack method.
\[ W_{bp}(::,j_{main},1:4) = W_{b}(::,j_{main},1:4) - \text{deltat} \cdot (\text{ver} \cdot \left( G_{bar}(::,j_{p\_ver},1:4) - G_{bar}(::,j_{main},1:4) \right) - \text{SGbar}(::,j_{main},1:4)) \]

\#ifdef MAKE_LIN
\[ W_{bp}(::,Px) = R_{gas} \cdot \left( \rho_0 \cdot W_{bp}(::,Tx) + T_0 \cdot W_{bp}(::,RHOx) \right) \]
\#else
\[ W_{bp}(::,Px) = R_{gas} \cdot \left( \rho_0 + W_{bp}(::,RHOx) \cdot i\text{Jac} \right) \cdot W_{bp}(::,Tx) + T_0 \cdot W_{bp}(::,RHOx) \]
\#endif

! Use boundary condition to set left and right boundary points.
call BC_eta(W_{bp})

\#ifdef MAKE_ND
! Update acs variables before the next step
\[ \text{do q = 1, nACSvar} \]
\[ \acs(:,:,q) = W_{bp}(::,q) \cdot i\text{Jac} \]
\[ \text{end do} \]
\#endif

\#ifdef MAKE_NDT
! Update acs variables before the next step
\[ \text{do q = 1, nACSvar} \]
\[ \acs(:,:,q) = W_{bp}(::,q) \cdot i\text{Jac} \]
\[ \text{end do} \]
\#endif

\#ifdef MAKE_ND2
! Update acs variables before the next step
\[ \text{do q = 1, nACSvar} \]
\[ \acs(:,:,q) = W_{bp}(::,q) \cdot i\text{Jac} \]
\[ \text{end do} \]
\#endif

\#ifdef MAKE_ND2T
! Update acs variables before the next step
\[ \text{do q = 1, nACSvar} \]
\[ \acs(:,:,q) = W_{bp}(::,q) \cdot i\text{Jac} \]
\[ \text{end do} \]
\#endif

\#ifdef MAKE_LIN
! Precalculate expressions for second step.
\[ \text{vel\_bar} = Eta_r \cdot W_{bp}(::,Vx) + Eta_z \cdot W_{bp}(::,2) \]
#endif

#ifdef MAKE_ND

VoR = acs(:,::,Vx)*rcp_cylR

! R-derivatives are "forward" diff wrt "rearward" diff for corrector.

u_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))

v_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))

t_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.

u_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))

v_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))

t_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))

p_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_z, v_z, t_z, p_z)
#endif

#ifdef MAKE_NDT

VoR = acs(:,::,Vx)*rcp_cylR

! R-derivatives are "forward" diff wrt "rearward" diff for corrector.

u_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))

v_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))

t_r(:,j_main) = Eta_r(:,j_main) * &
ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.

u_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))

v_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))

t_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))

p_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_z, v_z, t_z, p_z)
! Calculate temperature dependent viscosity and thermal conductivity
! call mu_kappa()
#endif
#ifdef MAKE_ND2

VoR = acs(:,:,Vx)*rcp_cylR

! R-derivatives are "forward" diff wrt "rearward" diff for corrector.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
  t_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

  call BCgrad_eta(u_r, v_r, t_r)

! R-derivatives are "forward" diff wrt "rearward" diff for corrector.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
  t_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

  call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
  p_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

  call BCgrad_xi(u_z, v_z, t_z, p_z)

! Calculate temperature dependent viscosity and thermal conductivity
! call mu_kappa()
#endif
#ifdef MAKE_ND2T

VoR = acs(:,:,Vx)*rcp_cylR

! R-derivatives are "forward" diff wrt "rearward" diff for corrector.
  u_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
  v_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
  t_r(:,j_main) = Eta_r(:,j_main) * &
                  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

  call BCgrad_eta(u_r, v_r, t_r)

! Z-derivatives are center differenced.
  u_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_z(i_main,:) = Xi_z(i_main,:) * &
                  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
  t_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
p_z(i_main,:) = Xi_z(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))
call BCgrad_xi(u_z, v_z, t_z, p_z)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()
#endif

! Calculate the main part of G1, G2, G3, and G4 for the second step of
! the MacCormack method.
! G* need to be (ni_main+2, nj_main+2) for finite differencing.
#ifdef MAKE_LIN
  Gbar(:,:,1) = rho0 * vel_bar
  Gbar(:,:,2) = 0.0d0
  Gbar(:,:,3) = Eta_r * Wbp(:,Px) / rho0
  Gbar(:,:,4) = PoRHOCv0 * Wbp(:,Vx) * Eta_r
#endif

#ifdef MAKE_D1
  Gbar(:,:,1) = rho0 * vel_bar
  Gbar(:,:,2) = -Mu_Rho0*(Vgrad(:,Ueta)*ETArZxi + &
                    Vgrad(:,Vxi))
  Gbar(:,:,3) = ( Wbp(:,5)*Eta_r &
                  - twothds*mu0*(2.0d0*Vgrad(:,Veta)*ETArZxi - Vgrad(:,Uxi) &
                    - Eta_r*Wbp(:,3)/tube_r) ) / rho0
  Gbar(:,:,4) = -Kap_RhoT0*t_eta*ETArZxi
#endif

#ifdef MAKE_ND
  Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Vx) * Eta_r
  Gbar(:,:,2) = Gb2*mu0*(v_z + u_r)
  Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu0*(2.0d0*v_r - VoR - u_z))
  Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - k0*t_r)
#endif

#ifdef MAKE_NDT
  Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Vx) * Eta_r
  Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
  Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
  Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*t_r)
#endif

#ifdef MAKE_ND2
  Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Vx) * Eta_r
  Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
  Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
  Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*t_r)
#endif

#ifdef MAKE_ND2T
  Gbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Vx) * Eta_r
  Gbar(:,:,2) = Gb2*mu*(v_z + u_r)
Gbar(:,:,3) = Gb3*(acs(:,Px) - twothds*mu*(2.0d0*v_r - VoR - u_z))
Gbar(:,:,4) = Gb4*(P0*acs(:,Vx) - kp*t_r)
#endif

! Calculate SGbar term for second part of MacCormack method.
! SGbar is (ni_main,nj_main,4).
#endif MAKE_LIN
SGbar(:,:,1) = -rho0 * Wbp(:,:,Vx) / tube_r
SGbar(:,:,2:3) = 0.0d0
SGbar(:,:,4) = -PoRHOCv0 * Wbp(:,:,Vx) / tube_r
#endif

#ifdef MAKE_D1
SGbar(:,:,1) = -rho0 * Wbp(i_main,j_main,3) / tube_r(i_main,j_main)
SGbar(:,:,2) = Mu_Rho0 * Vgrad(i_main,j_main,Ueta) * Z_xi(i_main,j_main)
SGbar(:,:,3) = 2.0d0 * Mu_Rho0 * (Vgrad(i_main,j_main,Veta)*Z_xi &
    - Wbp(i_main,j_main,3)/tube_r(i_main,j_main)) &
    / tube_r(i_main,j_main)
SGbar(:,:,4) = Kap_RhoTO * t_eta(i_main,j_main) * Z_xi(i_main,j_main) &
    / tube_r(i_main,j_main)
#endif

#ifdef MAKE_ND
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
    mu*v_z*rcp_cylR - &
    rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu*VoR*rcp_cylR - &
    rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
    acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
    Cv*rho0*acs(:,:,Ux)*t_z)*ONEoRHOCv0
#endif

#ifdef MAKE_NDT
! Full nonlinear, dissipative equations minus O(epsilon^3) and
! O(mu*epsilon^2) terms
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
    mu*v_z*rcp_cylR - &
    rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * ( -mu*VoR*rcp_cylR - &
    rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - &
    acs(:,:,Px))*(u_z + 0.5d0*VoR) - 0.5d0*P0*VoR - &
    Cv*rho0*acs(:,:,Ux)*t_z)*ONEoRHOCv0
#endif

#ifdef MAKE_ND2
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + &
    mu*v_z*rcp_cylR - &
    rho0*acs(:,:,Ux)*u_z)*ONEoRHO0
SGbar(:,:,3) = Jac * (-mu*VoR*rcp_cylR - 
    rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - 
    rho0*acs(:,:,Ux)*acs(:,:,Ux)*v_z)*ONEoRHO0
    Cv*rho0*acs(:,:,Ux)*t_z + 
    mu0*(twothds*((2.0d0*(u_z - VoR) - v_r)*u_z + VoR*VoR) + 
    (v_z + u_r)*v_z))*ONEoRHOCv0
#endif
#ifdef MAKE_ND2T
SGbar(:,:,1) = -0.5d0*(rho0+acs(:,:,RHOx))*VoR*Jac
SGbar(:,:,2) = Jac * (ONEoRHO0*acs(:,:,RHOx)*p_z + 
    mu*v_z*rcp_cylR - 
    rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,3) = Jac * (-mu*VoR*rcp_cylR - 
    rho0*acs(:,:,Ux)*v_z)*ONEoRHO0
SGbar(:,:,4) = Jac * ((P0*acs(:,:,RHOx)*ONEoRHO0 - 
    rho0*acs(:,:,Ux)*acs(:,:,Ux)*v_z)*ONEoRHO0
    Cv*rho0*acs(:,:,Ux)*t_z + 
    mu0*(twothds*((2.0d0*(u_z - VoR) - v_r)*u_z + VoR*VoR) + 
    (v_z + u_r)*v_z))*ONEoRHOCv0
#endif
! Do the second step of MacCormack method.
Wb(:,j_main,1:4) = 0.5d0 * (Wb(:,j_main,1:4) + 
    Wbp(:,j_main,1:4) - deltat*(ver * (Gbar(:,j_main,1:4) - 
    Gbar(:,j_m_ver,1:4)) - SGbar(:,j_main,1:4)))
#ifdef MAKE_LIN
Wb(:,:,Px) = Rgas * (rho0*Wb(:,:,Tx) + T0*Wb(:,:,RHOx))
#else
Wb(:,:,Px) = Rgas* ( (rho0 + Wb(:,:,RHOx) * iJac) * Wb(:,:,Tx) + 
    T0*Wb(:,:,RHOx) )
#endif
! Use boundary condition to set left and right boundary points.
call BC_eta(Wb)
#ifdef MAKE_ND
! Update acs variables before the next step
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_NDT
! Update acs variables before the next step
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2
! Update acs variables before the next step
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2T
! Update acs variables before the next step
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2T
! Update acs variables before the next step
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
SUBROUTINE FD_OPeta

!---------------------------------------------------------------------------
! ---------------------------------------------------------------------------
SUBROUTINE BCpred_xi()

! Accomodate MOC BC, which is particular to each coordinate system.
! Predictor step, Xi-direction.
!---------------------------------------------------------------------------

USE input_vars
USE boundary
implicit none

if (end1_BC == 'moc') then

! ifdef MAKE_LIN
OSD(:,RHOx) = 0.5d0*(-3d0*Wb(:,RHOx)+4d0*Wb(:,RHOx)-Wb(3,:,RHOx)) * &
Xi_x1(:,1)
OSD(:,Ux) = 0.5d0*(-3d0*Wb(:,Ux)+4d0*Wb(:,Ux)-Wb(3,:,Ux))*Xi_x1(:,1)
OSD(:,Tx) = 0.5d0*(-3d0*Wb(:,Tx)+4d0*Wb(:,Tx)-Wb(3,:,Tx))*Xi_x1(:,1)
L1 = -C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - C0*rho0*OSD(:,Tx)
L2 = 0.0d0
L3 = 0.0d0
L4 = 0.0d0
BEQ(:,RHOx) = (L2/Cp - 0.5d0*P0*(L1+L4)/C02)/(rho0*T0)
BEQ(:,Ux) = (P0*(L1-L4))/(2d0*C0*rho02*T0) ! + g_z terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = (-0.5d0*P0*(L1+L4) - T0*L2) / (Cp*rho02*T0)
Wbp(1,:,1:4) = Wb(1,:,1:4) + deltat * BEQ
Wbp(1,:,Px) = Rgas * (rho0*Wbp(1,:,Tx) + T0*Wbp(1,:,RHOx))

#else
OSD(:,RHOx) = 0.5d0*(-3d0*Wb(:,RHOx)+4d0*Wb(:,RHOx)-Wb(3,:,RHOx)) * &
Xi_x1(:,1)
OSD(:,Ux) = 0.5d0*(-3d0*Wb(:,Ux)+4d0*Wb(:,Ux)-Wb(3,:,Ux))*Xi_x1(:,1)
OSD(:,Tx) = 0.5d0*(-3d0*Wb(:,Tx)+4d0*Wb(:,Tx)-Wb(3,:,Tx))*Xi_x1(:,1)
L1 = -C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - C0*rho0*OSD(:,Tx)
L2 = 0.0d0

RETURN

END SUBROUTINE FD_OPeta

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
L3 = 0.0d0
L4 = 0.0d0
L1p = -C0*(T0+acs(1,:,Tx))*OSD(:,RHOx) + &
    gamma_ratio*(rho0*T0*rho0*acs(1,:,Tx)+acs(1,:,RHOx)*T0)*OSD(:,Ux) - &
    C0*(rho0*acs(1,:,RHOx))*OSD(:,Tx)
L2p = 0.0d0
L3p = 0.0d0
L4p = 0.0d0
BEQ(:,RHOx) = (C02*T0*rho0*L2p - 0.5d0*Cp*rho0*T0*P0*(L1p+L4p) - &
    0.5d0*Cp*rho0*T0*acs(1,:,Px)*(L1+L4) - &
    (rho0*acs(1,:,Tx) + T0*acs(1,:,RHOx))*(C02*L2 - &
    0.5d0*P0*Cp*(L1+L4)))/(rho02*T0*T0*P0*CO2)
BEQ(:,Ux) = ((T0*rho0*acs(1,:,Px) - P0*rho0*acs(1,:,Tx) - &
    2d0*T0*P0*acs(1,:,RHOx))*(L1-L4) + &
    P0*rho0*T0*(L1p-L4p))/(2d0*CO*rho02*T0*rho0*T0) ! + g_z terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = ((P0*rho0*acs(1,:,Tx) + 2d0*P0*T0*acs(1,:,RHOx) - &
    T0*rho0*acs(1,:,Px))*(L1+L4) + 4d0*T0*P0*acs(1,:,RHOx)*L2 - &
    T0*rho0*(P0*(L1p+L4p) + 2d0*T0*L2p))/(2d0*CO*rho02*T0*rho0*T0)
Wbp(1,:,1:4) = Wb(1,:,1:4) + deltat * BEQ
Wbp(1,:,Px) = Rgas * ( (rho0 + Wbp(1,:,RHOx)*iJac(1,:)))*Wbp(1,:,Tx) + &
    T0*Wbp(1,:,RHOx))

#ifdef MAKE_LIN
OSD(:,RHOx) = -0.5d0*(-3d0*Wb(ni,:,RHOx) + 4d0*Wb(ni-1,:,RHOx) - &
    Wb(ni-2,:,RHOx)) * Xi_x1(ni,:)
OSD(:,Ux) = -0.5d0*(-3d0*Wb(ni,:,Ux)+4d0*Wb(ni-1,:,Ux)-Wb(ni-2,:,Ux)) * &
    Xi_x1(ni,:)
OSD(:,Tx) = -0.5d0*(-3d0*Wb(ni,:,Tx)+4d0*Wb(ni-1,:,Tx)-Wb(ni-2,:,Tx)) * &
    Xi_x1(ni,:)
L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = CO*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + CO*rho0*OSD(:,Tx)
BEQ(:,RHOx) = (L2/Cp - 0.5d0*P0*(L1+L4)/CO2)/(rho0*T0)
BEQ(:,Ux) = (P0*(L1-L4))/(2d0*CO*rho02*T0) ! + g_z terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = (-0.5d0*P0*(L1+L4) - T0*L2) / (Cp*rho02*T0)
Wbp(ni,:,1:4) = Wb(ni,:,1:4) + deltat * BEQ
Wbp(ni,:,Px) = Rgas * (rho0*Wbp(ni,:,Tx) + T0*Wbp(ni,:,RHOx))
#endif
L3 = 0.0d0
L4 = CO*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + CO*rho0*OSD(:,Tx)
L1p = 0.0d0
L2p = 0.0d0
L3p = 0.0d0
L4p = CO*(T0+acs(ni,:,Tx))*OSD(:,RHOx) + &
gamma_ratio*(rho0*T0+rho0*acs(ni,:,Tx)+acs(ni,:,RHOx)*T0)*OSD(:,Ux) + &
CO*(rho0+acs(ni,:,RHOx))*OSD(:,Tx)

BEQ(:,RHOx) = (C02*T0*rho0*L2p - 0.5d0*Cp*rho0*T0*P0*(L1p+L4p) - &
0.5d0*Cp*rho0*T0*acs(ni,:,Px)*(L1+L4) - &
(rho0*acs(ni,:,Tx) + T0*acs(ni,:,RHOx))*(CO2*L2 - &
0.5d0*P0*Cp*(L1+L4)))/(rho02*T0*T0*Cp*C02)

BEQ(:,Ux) = (((T0*rho0*acs(ni,:,Px) - P0*rho0*acs(ni,:,Tx) - &
2d0*T0*P0*acs(ni,:,RHOx))*(L1-L4) + &
P0*rho0*T0*(L1-L4))/((2d0*C0*rho02*T0*rho0*T0) ! + g_z terms

BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = ((P0*rho0*acs(ni,:,Tx) + 2d0*P0*T0*acs(ni,:,RHOx) - &
T0*rho0*acs(ni,:,Px))*(L1+L4) + 4d0*T0*T0*acs(ni,:,RHOx)*L2 - &
T0*rho0*(P0*(L1+L4) + 2d0*T0*L2p)/(2d0*Cp*rho02*T0*rho0*T0)

Wbp(ni,:,1:4) = Wb(ni,:,1:4) + deltat * BEQ

#endif
end if

if (.not.(end1_BC == 'moc' .and. end2_BC == 'moc')) then
call BC_xi(Wbp)
end if
RETURN

END SUBROUTINE BCpred_xi

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BCcorr_xi()
! =======================================================================
! | Accomodate MOC BC, which is particular to each coordinate system. |
! | Corrector step, Xi-direction. |
! =======================================================================

USE input_vars
USE boundary
implicit none

if (end1_BC == 'moc') then
  #ifdef MAKE_LIN
  OSD(:,RHOx) = 0.5d0*(-3d0*Wbp(1,:,RHOx) + 4d0*Wbp(2,:,RHOx) - &
         Wbp(3,:,RHOx))*Xi_x1(1,:)
  OSD(:,Ux) = 0.5d0*(-3d0*Wbp(1,:,Ux)+4d0*Wbp(2,:,Ux)-Wbp(3,:,Ux))*Xi_x1(1,:)
\begin{align*}
\text{OSD}(\cdot, Tx) &= 0.5d0 \ast (-3d0 \ast \text{Wbp}(1, :, Tx) + 4d0 \ast \text{Wbp}(2, :, Tx) - \text{Wbp}(3, :, Tx)) \ast \text{Xi}_x1(1, :)
\text{L1} &= -C0 \ast T0 \ast \text{OSD}(\cdot, \text{RHOx}) + \rho0 \ast \gamma_{ratio} \ast T0 \ast \text{OSD}(\cdot, \text{Ux}) - C0 \ast \rho0 \ast \text{OSD}(\cdot, \text{Tx})
\text{L2} &= 0.0d0
\text{L3} &= 0.0d0
\text{L4} &= 0.0d0
\text{BEQ}(\cdot, \text{RHOx}) &= (L2/Cp - 0.5d0 \ast P0 \ast (L1+L4)/C02)/(\rho0 \ast T0)
\text{BEQ}(\cdot, \text{Ux}) &= (P0 \ast (L1-L4))/(2d0 \ast C0 \ast \rho02 \ast T0) + g_z \text{ terms}
\text{BEQ}(\cdot, \text{Vx}) &= 0.0d0
\text{BEQ}(\cdot, \text{Tx}) &= (-0.5d0 \ast P0 \ast (L1+L4) - T0 \ast L2) / (Cp \ast \rho02 \ast T0)
\text{Wb}(1, :, 1:4) &= 0.5d0 \ast (\text{Wb}(1, :, 1:4) + \text{Wb}(1, :, 1:4) + \text{deltat} \ast \text{BEQ})
\text{Wb}(1, :, Px) &= \text{Rgas} \ast (\rho0 \ast \text{Wb}(1, :, Tx) + T0 \ast \text{Wb}(1, :, \text{RHOx}))
\end{align*}

\begin{align*}
\text{else}
\text{OSD}(\cdot, \text{RHOx}) &= 0.5d0 \ast (-3d0 \ast \text{Wbp}(1, :, \text{RHOx}) + 4d0 \ast \text{Wbp}(2, :, \text{RHOx}) - \text{Wbp}(3, :, \text{RHOx})) \ast \text{Xi}_x1(1, :)
\text{OSD}(\cdot, \text{Ux}) &= 0.5d0 \ast (-3d0 \ast \text{Wbp}(1, :, \text{Ux}) + 4d0 \ast \text{Wbp}(2, :, \text{Ux}) - \text{Wbp}(3, :, \text{Ux})) \ast \text{Xi}_x1(1, :)
\text{OSD}(\cdot, \text{Tx}) &= 0.5d0 \ast (-3d0 \ast \text{Wbp}(1, :, \text{Tx}) + 4d0 \ast \text{Wbp}(2, :, \text{Tx}) - \text{Wbp}(3, :, \text{Tx})) \ast \text{Xi}_x1(1, :)
\text{L1} &= -C0 \ast T0 \ast \text{OSD}(\cdot, \text{RHOx}) + \rho0 \ast \gamma_{ratio} \ast T0 \ast \text{OSD}(\cdot, \text{Ux}) - C0 \ast \rho0 \ast \text{OSD}(\cdot, \text{Tx})
\text{L2} &= 0.0d0
\text{L3} &= 0.0d0
\text{L4} &= 0.0d0
\text{L1p} &= -C0 \ast (T0 \ast \text{acs}(1, :, Tx)) \ast \text{OSD}(\cdot, \text{RHOx}) + \&
\gamma_{ratio} \ast (\rho0 \ast T0 \ast \rho0 \ast \text{acs}(1, :, Tx) + \text{acs}(1, :, \text{RHOx}) \ast T0) \ast \text{OSD}(\cdot, \text{Ux}) + \&
C0 \ast (\rho0 \ast \text{acs}(1, :, \text{RHOx}) \ast \text{OSD}(\cdot, \text{Tx})
\text{L2p} &= 0.0d0
\text{L3p} &= 0.0d0
\text{L4p} &= 0.0d0
\text{BEQ}(\cdot, \text{RHOx}) &= (C02 \ast T0 \ast \text{rho0} \ast L2p - 0.5d0 \ast C0 \ast \rho0 \ast T0 \ast P0 \ast (L1p+L4p) - \&
0.5d0 \ast C0 \ast \rho0 \ast T0 \ast \text{acs}(1, :, Px) \ast (L1+L4) - \&
(\rho0 \ast \text{acs}(1, :, Tx) + T0 \ast \text{acs}(1, :, \text{RHOx})) \ast (C02 \ast L2 - \&
0.5d0 \ast P0 \ast \text{acs}(1, :, Px) \ast (L1+L4))/(\rho02 \ast T0 \ast T0 \ast \text{C02})
\text{BEQ}(\cdot, \text{Ux}) &= ((T0 \ast \text{rho0} \ast \text{acs}(1, :, Px) - P0 \ast \text{rho0} \ast \text{acs}(1, :, Tx) - \&
2d0 \ast T0 \ast \text{acs}(1, :, \text{RHOx}) \ast (L1+L4) + \&
P0 \ast \text{rho0} \ast T0 \ast (L1p+L4p))/(2d0 \ast C0 \ast \rho02 \ast T0 \ast \rho0 \ast T0) + g_z \text{ terms}
\text{BEQ}(\cdot, \text{Vx}) &= 0.0d0
\text{BEQ}(\cdot, \text{Tx}) &= ((P0 \ast \text{rho0} \ast \text{acs}(1, :, Tx) + 2d0 \ast P0 \ast T0 \ast \text{acs}(1, :, \text{RHOx}) - \&
T0 \ast \text{rho0} \ast \text{acs}(1, :, Px)) \ast (L1+L4) + 4d0 \ast T0 \ast \text{acs}(1, :, \text{RHOx}) \ast L2 - \&
T0 \ast \text{rho0} \ast (P0 \ast (L1p+L4p) + 2d0 \ast T0 \ast L2p))/(2d0 \ast C0 \ast \rho02 \ast T0 \ast \rho0 \ast T0)
\text{Wb}(1, :, 1:4) &= 0.5d0 \ast (\text{Wb}(1, :, 1:4) + \text{Wb}(1, :, 1:4) + \text{deltat} \ast \text{BEQ})
\text{Wb}(1, :, Px) &= \text{Rgas} \ast ((\rho0 + \text{Wb}(1, :, \text{RHOx}) \ast iJac(1, :)) \ast \text{Wb}(1, :, Tx) + \&
T0 \ast \text{Wb}(1, :, \text{RHOx}))
\text{endif}
\text{end if}
\text{if (end2_BC == 'moc')} then
\text{ifdef MAKE_LIN}
\text{OSD}(\cdot, \text{RHOx}) &= -0.5d0 \ast (-3d0 \ast \text{Wbp}(ni, :, \text{RHOx}) + 4d0 \ast \text{Wbp}(ni-1, :, \text{RHOx}) - \&
\text{Wbp}(ni-2, :, \text{RHOx})) \ast \text{Xi}_x1(ni, :)
\text{OSD}(\cdot, \text{Ux}) &= -0.5d0 \ast (-3d0 \ast \text{Wbp}(ni, :, \text{Ux}) + 4d0 \ast \text{Wbp}(ni-1, :, \text{Ux}) - \&
\text{Wbp}(ni-2, :, \text{Ux})) \ast \text{Xi}_x1(ni, :)
\text{OSD}(\cdot, \text{Tx}) &= -0.5d0 \ast (-3d0 \ast \text{Wbp}(ni, :, \text{Tx}) + 4d0 \ast \text{Wbp}(ni-1, :, \text{Tx}) - \&
\text{Wbp}(ni-2, :, \text{Tx})) \ast \text{Xi}_x1(ni, :)
\text{L1} &= 0.0d0
\text{endif}
\end{align*}
L2 = 0.0d0
L3 = 0.0d0
L4 = C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + C0*rho0*OSD(:,Tx)
BEQ(:,RHOx) = (L2/Cp - 0.5d0*P0*(L1+L4)/C02)/(rho0*T0)
BEQ(:,Ux) = (P0*(L1-L4))/(2d0*C0*rho02*T0) + g_z terms
BEQ(:,Tx) = (-0.5d0*P0*(L1+L4) - T0*L2) / (Cp*rho02*T0)
WB(ni,:,1:4) = 0.5d0*(WB(ni,:,1:4) + WBP(ni,:,1:4) + deltat * BEQ)
WB(ni,:,Px) = Rgas * (rho0*WB(ni,:,Tx) + T0*WB(ni,:,RHOx))

#else
OSD(:,RHOx) = -0.5d0*(-3d0*WBP(ni,:,RHOx) + 4d0*WBP(ni-1,:,RHOx) - &
WBP(ni-2,:,RHOx)) * Xi_x1(ni,:)
OSD(:,Ux) = -0.5d0*(-3d0*WBP(ni,:,Ux) + 4d0*WBP(ni-1,:,Ux) - &
WBP(ni-2,:,Ux)) * Xi_x1(ni,:)
OSD(:,Tx) = -0.5d0*(-3d0*WBP(ni,:,Tx) + 4d0*WBP(ni-1,:,Tx) - &
WBP(ni-2,:,Tx)) * Xi_x1(ni,:)

L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + C0*rho0*OSD(:,Tx)
L1p = 0.0d0
L2p = 0.0d0
L3p = 0.0d0
L4p = C0*(T0+ACS(ni,:,Tx))*OSD(:,RHOx) + &
gamma_ratio*(rho0*T0*rho0*ACS(ni,:,Tx)+ACS(ni,:,RHOx)*T0)*OSD(:,Ux) + &
C0*(rho0*ACS(ni,:,RHOx))*OSD(:,Tx)
BEQ(:,RHOx) = (C02*T0*rho0*L2p - 0.5d0*Cp*rho0*T0*P0*(L1p+L4p) - &
0.5d0*Cp*rho0*T0*ACS(ni,:,Px)*(L1+L4) - &
(rho0*ACS(ni,:,Tx) + T0*ACS(ni,:,RHOx))*(C02*L2 - &
0.5d0*P0*Cp*(L1+L4)))/(rho02*T0*P0*Cp*C02)
BEQ(:,Ux) = ((T0*rho0*ACS(ni,:,Px) - P0*rho0*ACS(ni,:,Tx) - &
2d0*T0*P0*ACS(ni,:,RHOx))*(L1-L4) + &
P0*rho0*T0*(L1p-L4))/2d0*C0*rho02*T0*rho0*T0) + g_z terms
BEQ(:,Tx) = (P0*rho0*ACS(ni,:,Tx) + 2d0*P0*T0*ACS(ni,:,RHOx) - &
T0*rho0*ACS(ni,:,Px))*(L1+L4) + 4d0*T0*T0*ACS(ni,:,RHOx)*L2 - &
T0*rho0*(P0*(L1p+L4) + 2d0*T0*L2p)/(2d0*Cp*rho02*T0*rho0*T0)
WB(ni,:,1:4) = 0.5d0*(WB(ni,:,1:4) + WB(ni,:,1:4) + deltat * BEQ)
WB(ni,:,Px) = Rgas * (rho0 + WB(ni,:,RHOx)*iJac(ni,:))*WB(ni,:,Tx) + &
T0*WB(ni,:,RHOx))

#endif
endif

if (.not.(end1_BC == 'moc' .and. end2_BC == 'moc')) then
    call BC_xi(WB)
endif

if (wall_TBC == 'iso') then
    WB(:,nj,Tx) = 0.0d0
endif
SUBROUTINE mu_kappa()
!
! | Calculate temperature dependent viscosity and thermal conductivity. |
! | .............................................................................
!
implicit none

real(DP) :: C1 = 70.0d0 ! Kelvin
real(DP) :: C2 = 33.0d0 ! Kelvin

#ifdef MAKE_NDT
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif
#ifdef MAKE_ND2
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif
#ifdef MAKE_ND2T
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif

kp = k0*(T0+C2)/(T0+acs(:,:,Tx)+C2) * (T0+acs(:,:,Tx)/T0)**1.5d0

RETURN

END SUBROUTINE mu_kappa

SUBROUTINE alloc_localvars()
!
! | Allocate memory for the arrays local to module mod_maccormack_lin. |
! | .............................................................................
!
integer :: astat

#ifdef MAKE_LIN
astat = 0
allocate(vel_bar(ni,nj), STAT=astat)
#endif
#ifdef MAKE_D1
astat = 0
allocate(t_xi(ni,nj), STAT=astat)
#endif
allocate(t_eta(ni,nj), STAT=astat)
allocate(XIzReta(ni,nj), STAT=astat)
allocate(ETArZxi(ni,nj), STAT=astat)
#endif
#ifdef MAKE_ND
astat = 0
allocate(VoR(ni,nj), STAT=astat)
allocate(rcp_cylR(ni,nj), STAT=astat)
allocate(u_z(ni,nj), STAT=astat)
allocate(u_r(ni,nj), STAT=astat)
allocate(v_z(ni,nj), STAT=astat)
allocate(v_r(ni,nj), STAT=astat)
allocate(t_z(ni,nj), STAT=astat)
allocate(t_r(ni,nj), STAT=astat)
allocate(p_z(ni,nj), STAT=astat)
allocate(p_r(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
#endif
#ifdef MAKE_ND2
astat = 0
allocate(VoR(ni,nj), STAT=astat)
allocate(rcp_cylR(ni,nj), STAT=astat)
allocate(u_z(ni,nj), STAT=astat)
allocate(u_r(ni,nj), STAT=astat)
allocate(v_z(ni,nj), STAT=astat)
allocate(v_r(ni,nj), STAT=astat)
allocate(t_z(ni,nj), STAT=astat)
allocate(t_r(ni,nj), STAT=astat)
allocate(p_z(ni,nj), STAT=astat)
allocate(p_r(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif
#ifdef MAKE_NDT
astat = 0
allocate(VoR(ni,nj), STAT=astat)
allocate(rcp_cylR(ni,nj), STAT=astat)
allocate(u_z(ni,nj), STAT=astat)
allocate(u_r(ni,nj), STAT=astat)
allocate(v_z(ni,nj), STAT=astat)
allocate(v_r(ni,nj), STAT=astat)
allocate(v_r(ni,nj), STAT=astat)
allocate(t_z(ni,nj), STAT=astat)
allocate(t_r(ni,nj), STAT=astat)
allocate(p_z(ni,nj), STAT=astat)
allocate(p_r(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif
#ifdef MAKE_ND2T
astat = 0
allocate(VoR(ni,nj), STAT=astat)
allocate(rcp_cylR(ni,nj), STAT=astat)
allocate(u_z(ni,nj), STAT=astat)
allocate(u_r(ni,nj), STAT=astat)
allocate(v_z(ni,nj), STAT=astat)
allocate(v_r(ni,nj), STAT=astat)
allocate(t_z(ni,nj), STAT=astat)
allocate(t_r(ni,nj), STAT=astat)
allocate(p_z(ni,nj), STAT=astat)
allocate(p_r(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif
if (.not. (end1_BC /= 'moc' .and. end2_BC /= 'moc')) then
astat = 0
allocate(OSD(nj,4), STAT=astat)
allocate(L1(nj), STAT=astat)
allocate(L2(nj), STAT=astat)
allocate(L3(nj), STAT=astat)
allocate(L4(nj), STAT=astat)
allocate(BEQ(nj,4), STAT=astat)
#endif
#ifdef MAKE_LIN
#else
allocate(L1p(nj), STAT=astat)
allocate(L2p(nj), STAT=astat)
allocate(L3p(nj), STAT=astat)
allocate(L4p(nj), STAT=astat)
#endif
if (astat > 0) print *, "Problem allocating memory for MOC BC."
end if

RETURN

END SUBROUTINE alloc_localvars

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE dealloc_localvars()
! | Deallocate memory for the arrays local to module mod_maccormack_lin. |
! | ----------------------------------------------------------------------------|
!
integer :: astat

#ifdef MAKE_LIN
  astat = 0
  deallocate(vel_bar, STAT=astat)
#endif
#ifdef MAKE_D1
  astat = 0
  deallocate(t_xi, STAT=astat)
  deallocate(t_eta, STAT=astat)
  deallocate(XIzReta, STAT=astat)
  deallocate(ETArZxi, STAT=astat)
#endif
#ifdef MAKE_ND
  astat = 0
  deallocate(VoR, STAT=astat)
  deallocate(rcp_cylR, STAT=astat)
  deallocate(u_z, STAT=astat)
  deallocate(u_r, STAT=astat)
  deallocate(v_z, STAT=astat)
  deallocate(v_r, STAT=astat)
  deallocate(t_z, STAT=astat)
  deallocate(t_r, STAT=astat)
  deallocate(p_z, STAT=astat)
  deallocate(p_r, STAT=astat)
  deallocate(Fb2, STAT=astat)
  deallocate(Fb3, STAT=astat)
  deallocate(Fb4, STAT=astat)
  deallocate(Gb2, STAT=astat)
  deallocate(Gb3, STAT=astat)
  deallocate(Gb4, STAT=astat)
#endif
#ifdef MAKE_ND2
  astat = 0
  deallocate(VoR, STAT=astat)
  deallocate(rcp_cylR, STAT=astat)
  deallocate(u_z, STAT=astat)
#endif
deallocate(u_r, STAT=astat)
deleallocate(v_z, STAT=astat)
deleallocate(v_r, STAT=astat)
deleallocate(t_z, STAT=astat)
deleallocate(t_r, STAT=astat)
deleallocate(p_z, STAT=astat)
deleallocate(p_r, STAT=astat)
deleallocate(Fb2, STAT=astat)
deleallocate(Fb3, STAT=astat)
deleallocate(Fb4, STAT=astat)
deleallocate(Gb2, STAT=astat)
deleallocate(Gb3, STAT=astat)
deleallocate(Gb4, STAT=astat)
deleallocate(mu, STAT=astat)
deleallocate(kp, STAT=astat)
#endif
#ifdef MAKE_NDT
astat = 0
  deleallocate(VoR, STAT=astat)
  deleallocate(rcp_cylR, STAT=astat)
  deleallocate(u_z, STAT=astat)
  deleallocate(u_r, STAT=astat)
  deleallocate(v_z, STAT=astat)
  deleallocate(v_r, STAT=astat)
  deleallocate(t_z, STAT=astat)
  deleallocate(t_r, STAT=astat)
  deleallocate(p_z, STAT=astat)
  deleallocate(p_r, STAT=astat)
  deleallocate(Fb2, STAT=astat)
  deleallocate(Fb3, STAT=astat)
  deleallocate(Fb4, STAT=astat)
  deleallocate(Gb2, STAT=astat)
  deleallocate(Gb3, STAT=astat)
  deleallocate(Gb4, STAT=astat)
  deleallocate(mu, STAT=astat)
  deleallocate(kp, STAT=astat)
#endif
#ifdef MAKE_ND2T
astat = 0
  deleallocate(VoR, STAT=astat)
  deleallocate(rcp_cylR, STAT=astat)
  deleallocate(u_z, STAT=astat)
  deleallocate(u_r, STAT=astat)
  deleallocate(v_z, STAT=astat)
  deleallocate(v_r, STAT=astat)
  deleallocate(t_z, STAT=astat)
  deleallocate(t_r, STAT=astat)
  deleallocate(p_z, STAT=astat)
  deleallocate(p_r, STAT=astat)
  deleallocate(Fb2, STAT=astat)
  deleallocate(Fb3, STAT=astat)
deallocate(Fb4, STAT=astat)
dallocate(Gb2, STAT=astat)
dallocate(Gb3, STAT=astat)
dallocate(Gb4, STAT=astat)
dallocate(mu, STAT=astat)
dallocate(kp, STAT=astat)
#endif

if (.not. (end1_BC /= 'moc' .and. end2_BC /= 'moc')) then
  astat = 0
  deallocate(OSD, STAT=astat)
dallocate(L1, STAT=astat)
dallocate(L2, STAT=astat)
dallocate(L3, STAT=astat)
dallocate(L4, STAT=astat)
dallocate(BEQ, STAT=astat)
#endif MAKE_LIN
#else
  deallocate(L1p, STAT=astat)
dallocate(L2p, STAT=astat)
dallocate(L3p, STAT=astat)
dallocate(L4p, STAT=astat)
#endif

  if (astat > 0) print *, "Problem deallocating memory for MOC BC."
endif

RETURN

END SUBROUTINE dealloc_localvars

!---------------------------------------------------------------------------------

END MODULE maccormack_cyl
D.4.13 maccormack_sph.fpp

MODULE maccormack_sph
!===========================================================================
!
! This module contains the subroutines for the MacCormack method of
! finite difference approximation in spherical coordinates.
!
! Table of contents:
! parameters A, B, switch
! SUBROUTINE propagate_sph()
! SUBROUTINE driver(timex)
! SUBROUTINE FDiff(ver)
! SUBROUTINE FD_OPxi(ver)
! SUBROUTINE FD_OPeta(ver)
! SUBROUTINE pandt_sphlin()
! SUBROUTINE MU_art()
!===========================================================================

USE global
USE FDiff_vars
USE grid

implicit none

! -------------------------------------------------------------------
! Public routine
Public :: propagate_sph
! -------------------------------------------------------------------

Private
integer, parameter :: A = 1, B = -1
integer, dimension(3), parameter :: switch = (/ B, 0, A /)

! Pointers to the generically named metrics.
real(DP), dimension(:,,:), pointer :: Xi_rh, Eta_th

! Pointers to the generically named grid arrays.
real(DP), dimension(:,,:), pointer :: tube_rh, tube_th

! Pointer to gravity components.
real(DP), pointer :: g_rh, g_th

#ifdef MAKE_LIN
real(DP), dimension(:,,:), allocatable :: COT
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: Vsph
#endif

#ifdef MAKE_ND
real(DP), dimension(:,,:), allocatable :: COT
#endif
real(DP), dimension(:,,:), allocatable :: Vsph
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: u_rh, u_th, v_rh, v_th
real(DP), dimension(:,,:), allocatable :: t_rh, t_th, p_rh, p_th
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
#endif

#ifdef MAKE_NDT

real(DP), dimension(:,,:), allocatable :: COT
real(DP), dimension(:,,:), allocatable :: Vsph
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: u_rh, u_th, v_rh, v_th
real(DP), dimension(:,,:), allocatable :: t_rh, t_th, p_rh, p_th
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,,:), allocatable :: mu, kp
#endif

#ifdef MAKE_ND2

real(DP), dimension(:,,:), allocatable :: COT
real(DP), dimension(:,,:), allocatable :: Vsph
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: u_rh, u_th, v_rh, v_th
real(DP), dimension(:,,:), allocatable :: t_rh, t_th, p_rh, p_th
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,,:), allocatable :: mu, kp
#endif

#ifdef MAKE_ND2T

real(DP), dimension(:,,:), allocatable :: COT
real(DP), dimension(:,,:), allocatable :: Vsph
real(DP), dimension(:,,:), allocatable :: VoR
real(DP), dimension(:,,:), allocatable :: u_rh, u_th, v_rh, v_th
real(DP), dimension(:,,:), allocatable :: t_rh, t_th, p_rh, p_th
real(DP), dimension(:,,:), allocatable :: Fb2, Fb3, Fb4
real(DP), dimension(:,,:), allocatable :: Gb2, Gb3, Gb4
! Temperature dependent viscosity and thermal conductivity:
real(DP), dimension(:,,:), allocatable :: mu, kp
#endif

real(DP), dimension(:,,:), allocatable :: OSD, BEQ
real(DP), dimension(:,), allocatable :: L1, L2, L3, L4
#endif

#ifdef MAKE_LIN
#else

real(DP), dimension(:,), allocatable :: L1p, L2p, L3p, L4p
#endif

CONTAINS
SUBROUTINE propagate_sph()

! This subroutine manages the main time loop for the finite difference calculation. It keeps track of which version of the operator we’re using.

USE input_vars
USE IO_tools
USE boundary
USE source
USE analyt
!
USE timer

implicit none

integer(i4b) :: ver, i
integer(i4b) :: timex
logical :: stopping
!

! Allocate variables in this module’s common block.
call alloc_FDvars()
call alloc_localvars()
!

! Declare pointers to the physical grid arrays tube_x1 and tube_x2.
tube_rh => tube_x1
tube_th => tube_x2
!

! Declare pointers to gravity variables.
g_rh => g_x1
g_th => g_x2
!

! Declare pointers to Xi_x1, X1_xi, X2_eta and Eta_x2.
Xi_rh => Xi_x1
Eta_th => Eta_x2

#ifdef MAKE_LIN
    COT = 1.0d0 / dtan(tube_th)
#endif

#ifdef MAKE_ND
    COT = 1.0d0 / dtan(tube_th)
    Fb2 = Jac * Xi_rh / rho0
    Fb3 = -Jac * Xi_rh / rho0
    Fb4 = Jac * Xi_rh / (rho0 * Cv)
    Gb2 = -Jac * Eta_th * rcp_tubeR / rho0
    Gb3 = Jac * Eta_th * rcp_tubeR / rho0
    Gb4 = Jac * Eta_th * rcp_tubeR / (rho0*Cv)
#endif

#ifdef MAKE_NDT
    COT = 1.0d0 / dtan(tube_th)
    Fb2 = Jac * Xi_rh / rho0
\[ F_{b3} = -\text{Jac} \times \frac{\Phi_{r}}{\rho_0} \]
\[ F_{b4} = \text{Jac} \times \frac{\Phi_{r}}{(\rho_0 \times \text{Cv})} \]
\[ G_{b2} = -\text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b3} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b4} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{(\rho_0 \times \text{Cv})} \]

```c
#endif
#endif MAKE_ND2
COT = 1.0d0 / \text{dtan(tube}\_th) \]
\[ F_{b2} = \text{Jac} \times \frac{\Phi_{r}}{\rho_0} \]
\[ F_{b3} = -\text{Jac} \times \frac{\Phi_{r}}{\rho_0} \]
\[ F_{b4} = \text{Jac} \times \frac{\Phi_{r}}{(\rho_0 \times \text{Cv})} \]
\[ G_{b2} = -\text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b3} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b4} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{(\rho_0 \times \text{Cv})} \]
```

```c
#endif MAKE_ND2T
COT = 1.0d0 / \text{dtan(tube}\_th) \]
\[ F_{b2} = \text{Jac} \times \frac{\Phi_{r}}{\rho_0} \]
\[ F_{b3} = -\text{Jac} \times \frac{\Phi_{r}}{\rho_0} \]
\[ F_{b4} = \text{Jac} \times \frac{\Phi_{r}}{(\rho_0 \times \text{Cv})} \]
\[ G_{b2} = -\text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b3} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{\rho_0} \]
\[ G_{b4} = \text{Jac} \times \frac{\Phi_{r} \times \text{Cv}}{(\rho_0 \times \text{Cv})} \]
```

```
!
Define indices for central differencing.
\[ i_{p\_one} = i_{\text{main}} + 1 \]
\[ i_{m\_one} = i_{\text{main}} - 1 \]
\[ j_{p\_one} = j_{\text{main}} + 1 \]
\[ j_{m\_one} = j_{\text{main}} - 1 \]
!
Transform variables to computational domain.
\[
\text{do} \ i = 1, n\text{ACSvar} \\
\quad \text{Wb}(\ldots,i) = \text{acs}(\ldots,i) \times \text{Jac} \\
\text{end do} \\
\text{call BC}_\text{xi}(\text{Wb}) \\
\text{call BC}_\text{eta}(\text{Wb})
! Initialize the cumulative sum for streaming velocities.
\[
\text{if} \ (\text{streamout} \neq \text{non}) \ \text{then} \\
\quad \text{sumacs} = 0.0d0 \\
\quad n\_\text{sum} = 0 \\
\text{end if}
\]
\[
\text{stopping} = \text{.FALSE.}
!
Start time loop.
\[ \text{timex} = \text{txmin} \]
!
Set version to A (1) if \text{timex} is even, B (-1) if odd.
\[ \text{ver} = -2\times\text{mod(timex,2)} + 1 \]
propagate: do

! This is the next time step.

timex = timex + 1

! Update source.
call ACSsrc(timex)

if (fdrv_type == 'PRS') then
  acs(:,,:Px) = Wb(:,,:Px) * iJac
  acs(:,,:Tx) = Wb(:,,:Tx) * iJac
  Wb(:,,:Px) = Jac*(Psource * Gaussian_BL + acs(:,,:Px) * iGaussian_BL)
  Wb(:,,:Tx) = Jac*(Temperature_BL + acs(:,,:Tx) * iGaussian_BL)
#endif MAKE_LIN
#else
  acs(:,,:Tx) = Wb(:,,:Tx) * iJac
  Wb(:,,:RHOx) = (Wb(:,,:Px)/Rgas - &
                   Rho0*Wb(:,,:Tx)) / (T0 + acs(:,,:Tx))
#endif

call BC_xi(Wb)
call BC_eta(Wb)
end if

! Do the finite difference operation for one time step.
call FDiff(ver)

! Contribute to cumulative sum for streaming calculation.

if (streamout .ne. 'non') then
  sumacs(:,1:nACSvar) = sumacs(:,1:nACSvar) + acs
  sumacs(:,nACSvar+1) = sumacs(:,nACSvar+1) + &
                       acs(:,RHOx)*acs(:,Ux)
  sumacs(:,nACSvar+2) = sumacs(:,nACSvar+2) + &
                       acs(:,RHOx)*acs(:,Vx)
  n_sum = n_sum + 1
end if

! Calculate maximum volume velocity and write out for each period.
call max_Uvol(timex)

! Write a file every so often.
call fileout(timex)

! Decide whether to stop the time loop.
call test_for_doneness(timex, stopping)
if (stopping) exit

!! Get ready for the next propagation step.
! Switch the version. (switch = [-1 0 1])
   ver = switch(2-ver) ! use the index of switch to find the next ver

   end do propagate
   txmax = timex

! Clean up FDvars.
   call dealloc_localvars()
   call dealloc_FDvars()

RETURN

END SUBROUTINE propagate_sph

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE test_for_doneness(time_inx, stop_flag)
! =========================================================================
! | This subroutine decides when to stop the time loop by the criterion |
! | outlined below. | |
! =========================================================================

implicit none

integer, intent(IN) :: time_inx
logical, intent(OUT) :: stop_flag

if ( time_inx .ge. txmax ) stop_flag = .TRUE.

RETURN

END SUBROUTINE test_for_doneness

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE FDiff(ver)
! =======================================================================
! | This subroutine organizes the alternate versions and ordering of the |
! | finite difference operators in the z and r directions. |
! =======================================================================

implicit none

integer, intent(in) :: ver

if ( ver == A ) then
   ! Xi-direction
   call FD_OPxi(ver)
   ! Eta_direction
   call FD_OPeta(ver)
else if ( ver == B ) then
    ! Eta_direction
    call FD_OPeta(ver)
    ! Xi-direction
    call FD_OPxi(ver)
else
    print *, "Version is neither A nor B. Goodbye."
    STOP
end if

RETURN

END SUBROUTINE FDiff

SUBROUTINE FD_OPxi(ver)

USE boundary
USE source

implicit none

integer, intent(in) :: ver
integer, dimension(ni_main) :: i_p_ver, i_m_ver
real(DP), dimension(:,,:), pointer :: Fbar, SFbar
integer :: q

! Declare pointers to FGbar, SFGbar so they can be aptly named Fbar, SFbar.
Fbar => FGbar
SFbar => SFGbar

! Define shifted index vectors for the main and secondary acs vars arrays.
i_p_ver = i_main+ver  ! The i_main index vector shifted one
i_m_ver = i_main-ver  ! bin to the right or left.

! Eventually use the fourth order scheme over the inner domain.

! Use the second order scheme over the whole domain for now.

! Calculate these repeated expressions once.
#ifdef MAKE_LIN
    VoR = Wb(:,:,Ux) * rcp_tubeR
    Vsph = 2.0d0*Wb(:,:,Ux) !+ 0.5d0*Wb(:,:,Vx)*COT
#endif
#ifdef MAKE_ND
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Rh-derivatives are "rearward" diff wrt "forward" diff for predictor.
u_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))
call BCgrad_xi(u_rh, v_rh, t_rh)

! Th-derivatives are center differenced.
u_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_th, v_th, t_th, p_th)
#endif

#ifdef MAKE_ND
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Rh-derivatives are "rearward" diff wrt "forward" diff for predictor.
u_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * &
   ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))
call BCgrad_xi(u_rh, v_rh, t_rh)

! Th-derivatives are center differenced.
u_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_th(:,j_main) = Eta_th(:,j_main) * &
   0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_th, v_th, t_th, p_th)
#endif
! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()

#endif
#endif MAKE_ND2

VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Rh-derivatives are "rearward" diff wrt "forward" diff for predictor.

u_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))

call BCgrad_xi(u_rh, v_rh, t_rh)

! Th-derivatives are center differenced.

u_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))

call BCgrad_eta(u_th, v_th, t_th, p_th)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()

#endif
#endif MAKE_ND2T

VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Rh-derivatives are "rearward" diff wrt "forward" diff for predictor.

u_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Ux) - acs(i_m_ver,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Vx) - acs(i_m_ver,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_main,:,Tx) - acs(i_m_ver,:,Tx))

call BCgrad_xi(u_rh, v_rh, t_rh)

! Th-derivatives are center differenced.

u_th(:,j_main) = Eta_th(:,j_main) * &
v_th(:,j_main) = Eta_th(:,j_main) * &
0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))

t_th(:,j_main) = Eta_th(:,j_main) * &
0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
p_th(:,j_main) = Eta_th(:,j_main) * &
0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))

call BCgrad_eta(u_th, v_th, t_th, p_th)

! Calculate temperature dependent viscosity and thermal conductivity
call mu_kappa()

#endif

! Calculate F1, F2, F3, and F4 for the first step of the MacCormack method.
! Fbar needs to be (ni_main+2*BP, nj_main+2*BP,4) for finite differencing.
#ifdef MAKE_LIN
Fbar(:,:,1) = Xi_rh * rho0 * Wb(:,:,Ux)
Fbar(:,:,2) = Xi_rh * Wb(:,:,Px) / rho0
Fbar(:,:,3) = 0.0d0
Fbar(:,:,4) = Xi_rh * PorH0Cv0 * Wb(:,:,Ux)
#endif
#ifdef MAKE_ND
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu0*(2d0*(u_rh - Vsph) - &
v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu0*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - k0*t_rh)
#endif
#ifdef MAKE_NDT
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - &
v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
#ifdef MAKE_ND2
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - &
v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
#ifdef MAKE_ND2T
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wb(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - &
v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
! Calculate SF term for first part of MacCormack method.

#ifdef MAKE_LIN
SFbar(:,:,1) = -rho0 * rcp_tubeR * Vsp + Density_src*Jac
SFbar(:,:,2) = 0.0d0 + (g_rh + Vr_src)*Jac
SFbar(:,:,3) = 0.0d0
SFbar(:,:,4) = -PoRHO0c0 * rcp_tubeR * Vsp + Temperature_src*Jac
#endif

#ifdef MAKE_ND
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx)))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*(((rho0 + 0d0)*acs(:,:,Vx)) * &
(0.5d0*acs(:,:,Vx) - u_th) + mu0*(rcp_tubeR*((u_th - &
0.5d0*acs(:,:,Vx)))*COT - 2d0*v_th - 2d0*Vsph)) + g_rh)*Jac
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - &
(rho0+0d0)*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + &
m0*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx)*COT) + &
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsp + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_th*rcp_tubeR/rho0 + &
(k0*COT*rcp_tubeR - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx))) + &
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src)*Jac
#endif

#ifdef MAKE_NDT
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx)))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*(((rho0 + 0d0)*acs(:,:,Vx)) * &
(0.5d0*acs(:,:,Vx) - u_th) + mu0*(rcp_tubeR*((u_th - &
0.5d0*acs(:,:,Vx)))*COT - 2d0*v_th - 2d0*Vsph)) + g_rh)*Jac
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - &
(rho0+0d0)*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + &
m0*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx)*COT) + &
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsp + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_th*rcp_tubeR/rho0 + &
(kp*COT*rcp_tubeR - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx))) + &
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src)*Jac
#endif

#ifdef MAKE_ND2
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx)))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*(((rho0 + 0d0)*acs(:,:,Vx)) * &
(0.5d0*acs(:,:,Vx) - u_th) + mu0*(rcp_tubeR*((u_th - &
0.5d0*acs(:,:,Vx)))*COT - 2d0*v_th - 2d0*Vsph)) + g_rh)*Jac
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - &
(rho0+0d0)*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + &
m0*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx)*COT) + &
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsp + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_th*rcp_tubeR/rho0 + &
(kp*COT*rcp_tubeR - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx))) + &
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src)*Jac
#endif
\[
\begin{align*}
&\frac{u_{th}(v_{rh}+rcp_{tubeR}u_{th}))}{rcp_{tubeR}/(rho0*Cv)} + \frac{Temperature_src*Jac}{\text{#endif}} \\
&\text{#ifdef MAKE_ND2T} \\
&\text{SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx))*Vsph) * Jac} \\
&\text{SFbar(:,:,2) = (rcp_{tubeR}/rho0)((rho0 + 0d0)*acs(:,:,Vx) + (0.5d0*acs(:,:,Vx) - u_{th}) + mu*(rcp_{tubeR}((u_{th} - & \text{0.5d0*acs(:,:,Vx))*COT - 2d0*v_{th}) - 2d0*Vsph)) + g_{rh})*Jac} \\
&\text{SFbar(:,:,3) = rcp_{tubeR}/rho0*(acs(:,:,RHOx)*p_{th}/rho0 - (rho0 +0d0)*acs(:,:,Vx)*(v_{th} + 0.5d0*acs(:,:,Ux)) + & mu*rcp_{tubeR}(2d0*COT*(v_{th} - 0.5d0*acs(:,:,Vx)*COT) + & 3d0*(u_{th} - 0.5d0*acs(:,:,Vx))))*Jac} \\
&\text{SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + & (P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_{th}*rcp_{tubeR}/rho0 + & (kp*COT*rcp_{tubeR} - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx)) + & rcp_{tubeR}*t_{th})/(rho0*Cv) + 0.5d0*Temperature_src)*Jac} \\
&\text{#endif}
\end{align*}
\]

! Do the first step of MacCormack method.
Wbp(i_main,:,1:4) = Wb(i_main,:,1:4) + deltat * (ver * & 
(Fbar(i_main,:,1:4) - Fbar(i_p_ver,:,1:4)) + 
SFbar(i_main,:,1:4))

! Pressure (p)
#ifdef MAKE_LIN
Wbp(:,:,Px) = Rgas*( rho0 * Wbp(:,:,Tx) + T0*Wbp(:,:,RHOx) )
#else
Wbp(:,:,Px) = Rgas*( (rho0 + Wbp(:,:,RHOx) * iJac) * Wbp(:,:,Tx) + & 
T0*Wbp(:,:,RHOx) )
#endif

! Use boundary condition to set left and right boundary points.
call BCpred_xi()
call BC_eta(Wbp)

#ifdef MAKE_ND
do q = 1, nACSvar
acs(:,:,q) = Wbp(:,:,q) * iJac
done
#endif

#ifdef MAKE_NDT
do q = 1, nACSvar
acs(:,:,q) = Wbp(:,:,q) * iJac
done
#endif

#ifdef MAKE_ND2
do q = 1, nACSvar
acs(:,:,q) = Wbp(:,:,q) * iJac
done
#endif

#ifdef MAKE_ND2T
do q = 1, nACSvar
    acs(:,:,q) = Wbp(:,:,q) * iJac
end do
#endif
#endif MAKE_LIN
! Precalculate expressions for second step.
VoR = Wbp(:,:,Ux) * rcp_tubeR
Vsph = 2.0d0*Wbp(:,:,Ux) !+ 0.5d0*Wbp(:,:,Vx)*COT
#endif
#endif MAKE_ND
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR
! Rh-derivatives are "forward" diff wrt "rearward" diff for corrector.
    u_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Ux) - acs(i_main,:,Ux))
    v_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
    t_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Tx) - acs(i_main,:,Tx))
call BCgrad_xi(u_rh, v_rh, t_rh)
! Th-derivatives are center differenced.
    u_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
    v_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
    t_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
    p_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_th, v_th, t_th, p_th)
#endif MAKE_NDT
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR
! Rh-derivatives are "forward" diff wrt "rearward" diff for corrector.
    u_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Ux) - acs(i_main,:,Ux))
    v_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
    t_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Tx) - acs(i_main,:,Tx))
call BCgrad_xi(u_rh, v_rh, t_rh)
! Th-derivatives are center differenced.

\[
\begin{align*}
    u_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Ux) - acs(j_{m\_one},Ux)) \\
v_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Vx) - acs(j_{m\_one},Vx)) \\
t_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Tx) - acs(j_{m\_one},Tx)) \\
p_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Px) - acs(j_{m\_one},Px))
\end{align*}
\]

\text{call \text{BCgrad_eta}(u_{th}, v_{th}, t_{th}, p_{th})}

! Calculate temperature dependent viscosity and thermal conductivity
\text{call \text{mu_kappa}()}

\text{#endif}
\text{#ifdef MAKE_ND2}

VoR = acs(:,Vx)\times rcp\_tubeR \\
Vsph = (acs(:,Ux) + 0.5d0*acs(:,Vx)\times COT) \times rcp\_tubeR

! Rh-derivatives are "forward" diff wrt "rearward" diff for corrector.

\[
\begin{align*}
    u_{rh}(i_{main}) &= \text{Xi}_{rh}(i_{main}) \times \text{ver} \times (acs(i_{p\_ver},Ux) - acs(i_{main},Ux)) \\
v_{rh}(i_{main}) &= \text{Xi}_{rh}(i_{main}) \times \text{ver} \times (acs(i_{p\_ver},Vx) - acs(i_{main},Vx)) \\
t_{rh}(i_{main}) &= \text{Xi}_{rh}(i_{main}) \times \text{ver} \times (acs(i_{p\_ver},Tx) - acs(i_{main},Tx))
\end{align*}
\]

\text{call \text{BCgrad_xi}(u_{rh}, v_{rh}, t_{rh})}

! Th-derivatives are center differenced.

\[
\begin{align*}
    u_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Ux) - acs(j_{m\_one},Ux)) \\
v_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Vx) - acs(j_{m\_one},Vx)) \\
t_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Tx) - acs(j_{m\_one},Tx)) \\
p_{th}(j_{main}) &= \text{Eta}_{th}(j_{main}) \times \frac{1}{2} (acs(j_{p\_one},Px) - acs(j_{m\_one},Px))
\end{align*}
\]

\text{call \text{BCgrad_eta}(u_{th}, v_{th}, t_{th}, p_{th})}

! Calculate temperature dependent viscosity and thermal conductivity
\text{call \text{mu_kappa}()}

\text{#endif}
\text{#ifdef MAKE_ND2T}

VoR = acs(:,Vx)\times rcp\_tubeR \\
Vsph = (acs(:,Ux) + 0.5d0*acs(:,Vx)\times COT) \times rcp\_tubeR

! Rh-derivatives are "forward" diff wrt "rearward" diff for corrector.
\begin{verbatim}
u_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Ux) - acs(i_main,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Vx) - acs(i_main,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * &
    ver * (acs(i_p_ver,:,Tx) - acs(i_main,:,Tx))
call BCgrad_xi(u_rh, v_rh, t_rh)

! Th-derivatives are center differenced.
u_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Ux) - acs(:,j_m_one,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Vx) - acs(:,j_m_one,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Tx) - acs(:,j_m_one,Tx))
p_th(:,j_main) = Eta_th(:,j_main) * &
    0.5d0 * (acs(:,j_p_one,Px) - acs(:,j_m_one,Px))
call BCgrad_eta(u_th, v_th, t_th, p_th)

! Calculate temperature dependent viscosity and thermal conductivity
    call mu_kappa()

#ifdef MAKE_LIN
Fbar(:,:,1) = Xi_rh * rho0 * Wbp(:,:,Ux)
Fbar(:,:,2) = Xi_rh * Wbp(:,:,Px) / rho0
Fbar(:,:,3) = 0.0d0
Fbar(:,:,4) = Xi_rh * PoRHOCv0 * Wbp(:,:,Ux)
#endif
#ifdef MAKE_ND
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu0*(2d0*(u_rh - Vsph) - &
    v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu0*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - k0*t_rh)
#endif
#ifdef MAKE_NDT
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - &
    v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
#ifdef MAKE_ND2
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Ux) * Xi_rh
\end{verbatim}
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
#endif MAKE_ND2T
Fbar(:,:,1) = (rho0 + acs(:,:,RHOx)) * Wbp(:,:,Ux) * Xi_rh
Fbar(:,:,2) = Fb2*(acs(:,:,Px) - twothds*mu*(2d0*(u_rh - Vsph) - v_th * rcp_tubeR))
Fbar(:,:,3) = Fb3*mu*(v_rh - rcp_tubeR*(acs(:,:,Vx) + u_rh))
Fbar(:,:,4) = Fb4 * (P0*acs(:,:,Ux) - kp*t_rh)
#endif
#endif MAKE_LIN
SFbar(:,:,1) = -rho0 * rcp_tubeR * Vsph + Density_src*Jac
SFbar(:,:,2) = 0.0d0 + (g_rh + Vr_src)*Jac
SFbar(:,:,3) = 0.0d0
SFbar(:,:,4) = -PoRHOCv0 * rcp_tubeR * Vsph + Temperature_src*Jac
#endif
#endif MAKE_ND
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*(rho0*acs(:,:,Vx)) * &
(0.5d0*acs(:,:,Vx) - u_th) + mu0*(rcp_tubeR*((u_th - &
0.5d0*acs(:,:,Vx))*COT - 2d0*v_th) - 2d0*Vsph)) + g_rh)*Jac
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - &
rho0*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + &
mu0*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx))*COT) + &
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_th*rcp_tubeR)/rho0 + &
(k0*COT*rcp_tubeR - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx)))) * &
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src*Jac
#endif
#endif MAKE_NDT
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*((rho0 + 0d0)*acs(:,:,Vx)) * &
(0.5d0*acs(:,:,Vx) - u_th) + mu*(rcp_tubeR*((u_th - &
0.5d0*acs(:,:,Vx))*COT - 2d0*v_th) - 2d0*Vsph)) + g_rh)*Jac
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - &
(rho0+0d0)*acs(:,:,Vx)*((v_th + 0.5d0*acs(:,:,Ux)) + &
mu*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx))*COT) + &
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*v_th*rcp_tubeR)/rho0 + &
(kp*COT*rcp_tubeR - Cv*acs(:,:,Vx)*(rho0 + acs(:,:,RHOx)))) * &
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src*Jac
#endif
#endif MAKE_ND2
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx))*Vsph) * Jac
SFbar(:,:,2) = (rcp_tubeR/rho0*(rho0 + 0d0)*acs(:,:,Vx)) * &
\[(0.5d0*acs(:,:,Vx) - u_th) + \mu*(rcp_tubeR*((u_th - & \\
0.5d0*acs(:,:,Vx))*COT - 2d0*v_th) - 2d0*Vsph)) + g_rh)*Jac\]

\[
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - & \\
(rho0+0d0)*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + & \\
\mu*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx)*COT) + & \\
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac\]

\[
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + & \\
(P0*acs(:,:,RHOx) - rho0)*v_th*rcp_tubeR)/rho0 + & \\
((kp*COT*rcp_tubeR - Cv*acs(:,:,Vx)*rho0)*acs(:,:,RHOx))/t_th + & \\
mu0*(fourthds*(v_th*(rcp_tubeR*(v_th-acs(:,:,Vx)*COT) - & \\
0.5d0*u_rh) + 0.5d0*rcp_tubeR*acs(:,:,Ux)*acs(:,:,Ux)) + & \\
 VoR*(twothds*COT*(COT+acs(:,:,Ux))+acs(:,:,Vx)-2.0d0*u_th) + & \\
u_th*(v_rh+rcp_tubeR*u_th)))*rcp_tubeR)/(rho0*Cv) + & \\
Temperature_src)*Jac\]

#endif

#ifdef MAKE_ND2T

\[
SFbar(:,:,1) = (Density_src - (rho0 + acs(:,:,RHOx))*Vsph) * Jac\]

\[
SFbar(:,:,2) = (rcp_tubeR/rho0*((rho0 + 0d0)*acs(:,:,Vx) + & \\
0.5d0*acs(:,:,Vx) - u_th) + \mu*(rcp_tubeR*((u_th - & \\
0.5d0*acs(:,:,Vx))*COT - 2d0*v_th) - 2d0*Vsph)) + g_rh)*Jac\]

\[
SFbar(:,:,3) = rcp_tubeR/rho0*(acs(:,:,RHOx)*p_th/rho0 - & \\
(rho0+0d0)*acs(:,:,Vx)*(v_th + 0.5d0*acs(:,:,Ux)) + & \\
\mu*rcp_tubeR*(2d0*COT*(v_th - 0.5d0*acs(:,:,Vx)*COT) + & \\
3d0*(u_th - 0.5d0*acs(:,:,Vx)))*Jac\]

\[
SFbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + & \\
(P0*acs(:,:,RHOx) - rho0)*v_th*rcp_tubeR)/rho0 + & \\
(kp*COT*rcp_tubeR - Cv*acs(:,:,Vx)*rho0)*acs(:,:,RHOx))/t_th + & \\
rcp_tubeR*t_th)/(rho0*Cv) + 0.5d0*Temperature_src)*Jac\]

#endif

! Do the second step of MacCormack method.

\[
Wb(i_main,:,1:4) = 0.5d0 * (Wb(i_main,:,1:4) + & \\
Wbp(i_main,:,1:4) - deltat*(ver * (Fbar(i_main,:,1:4) - & \\
Fbar(i_m_ver,:,1:4)) - SFbar(i_main,:,1:4)))\]

#ifdef MAKE_LIN

\[
Wb(:,:,Px) = Rgas*( rho0 * Wb(:,:,Tx) + T0*Wb(:,:,RHOx) )\]

#else

\[
Wb(:,:,Px) = Rgas*( (rho0 + Wb(:,:,RHOx) * iJac) * Wb(:,:,Tx) + & \\
T0*Wb(:,:,RHOx) )\]

#endif

! Use boundary condition to set left and right boundary points.

call BCcorr_xi()
call BC_eta(Wb)

#ifdef MAKE_ND

do q = 1, nACSvar
    acs(:,:,q) = Wb(:,:,q) * iJac
end do

#endif
```fortran
#ifdef MAKE_NDT
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif
#ifdef MAKE_ND2T
    do q = 1, nACSvar
        acs(:,:,q) = Wb(:,:,q) * iJac
    end do
#endif

RETURN

END SUBROUTINE FD_OPxi

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE FD_OPeta(ver)
! =======================================================================
! | This subroutine computes the finite difference in the eta-direction. |
! =======================================================================

USE boundary

implicit none

integer, intent(in) :: ver
integer, dimension(nj_main) :: j_p_ver, j_m_ver
real(DP), dimension(:,,:), pointer :: Gbar, SGbar
integer :: q

! Point Gbar and SGbar at FGbar and SFGbar so their names refer to functions.
    Gbar => FGbar
    SGbar => SFGbar

! Define shifted index vectors for the main and secondary acs vars arrays.
    j_p_ver = j_main+ver  ! The j_main index vector shifted one
    j_m_ver = j_main-ver  ! bin up or down.

! Eventually use the fourth order scheme over the inner domain.

! Use the second order scheme over the whole domain for now.

! Calculate these repeated expressions once.
#ifdef MAKE_LIN
    VoR = Wb(:,Vx) * rcp_tubeR
#endif
```
Vsph = Wb(:,:,Vx)*COT
#endif

#ifdef MAKE_ND
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Th-derivatives are "rearward" diff wrt "forward" diff for predictor.
   u_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Ux) - acs(:,j_m_ver,Ux))
   v_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Vx) - acs(:,j_m_ver,Vx))
   t_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Tx) - acs(:,j_m_ver,Tx))

call BCgrad_eta(u_th, v_th, t_th)

! Rh-derivatives are center differenced.
   u_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
   v_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
   t_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
   p_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)
#endif

#ifdef MAKE_NDT
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Th-derivatives are "rearward" diff wrt "forward" diff for predictor.
   u_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Ux) - acs(:,j_m_ver,Ux))
   v_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Vx) - acs(:,j_m_ver,Vx))
   t_th(:,j_main) = Eta_th(:,j_main) * &
                   ver * (acs(:,j_main,Tx) - acs(:,j_m_ver,Tx))

call BCgrad_eta(u_th, v_th, t_th)

! Rh-derivatives are center differenced.
   u_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
   v_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
   t_rh(i_main,:) = Xi_rh(i_main,:) * &
                   0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
   p_rh(i_main,:) = Xi_rh(i_main,:) * &
$0.5d0 \times (\text{acs}(\text{i}_p, :, \text{Px}) - \text{acs}(\text{i}_m, :, \text{Px}))$

call BCgrad_xi(u\_rh, v\_rh, t\_rh, p\_rh)

! Calculate temperature dependent viscosity and thermal conductivity
call mu\_kappa()
#endif
#endif MAKE\_KAPPA

VoR = acs(:, :, Vx) * rcp\_tubeR
Vsph = (acs(:, :, Ux) + 0.5d0 * acs(:, :, Vx) * COT) * rcp\_tubeR

! Th-derivatives are "rearward" diff wrt "forward" diff for predictor.
$u\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Ux) - acs(:, j\_m\_ver, Ux))
$v\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Vx) - acs(:, j\_m\_ver, Vx))
$t\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Tx) - acs(:, j\_m\_ver, Tx))

call BCgrad_eta(u\_th, v\_th, t\_th)

! Rh-derivatives are center differenced.
$u\_rh(\text{i}_\text{main}, :) = Xi\_rh(\text{i}_\text{main}, :) \times &$
0.5d0 * (acs(\text{i}_p, :, Ux) - acs(\text{i}_m, :, Ux))
$v\_rh(\text{i}_\text{main}, :) = Xi\_rh(\text{i}_\text{main}, :) \times &$
0.5d0 * (acs(\text{i}_p, :, Vx) - acs(\text{i}_m, :, Vx))
$t\_rh(\text{i}_\text{main}, :) = Xi\_rh(\text{i}_\text{main}, :) \times &$
0.5d0 * (acs(\text{i}_p, :, Tx) - acs(\text{i}_m, :, Tx))
$p\_rh(\text{i}_\text{main}, :) = Xi\_rh(\text{i}_\text{main}, :) \times$
0.5d0 * (acs(\text{i}_p, :, Px) - acs(\text{i}_m, :, Px))

call BCgrad_xi(u\_rh, v\_rh, t\_rh, p\_rh)

! Calculate temperature dependent viscosity and thermal conductivity
call mu\_kappa()
#endif
#endif MAKE\_ND2T

VoR = acs(:, :, Vx) * rcp\_tubeR
Vsph = (acs(:, :, Ux) + 0.5d0 * acs(:, :, Vx) * COT) * rcp\_tubeR

! Th-derivatives are "rearward" diff wrt "forward" diff for predictor.
$u\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Ux) - acs(:, j\_m\_ver, Ux))
$v\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Vx) - acs(:, j\_m\_ver, Vx))
$t\_th(:, j\_main) = Eta\_th(:, j\_main) \times &$
ver * (acs(:, j\_main, Tx) - acs(:, j\_m\_ver, Tx))

call BCgrad_eta(u\_th, v\_th, t\_th)

! Rh-derivatives are center differenced.
u_rh(i_main,:) = Xi_rh(i_main,:) * & 0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))

v_rh(i_main,:) = Xi_rh(i_main,:) * & 0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))

t_rh(i_main,:) = Xi_rh(i_main,:) * & 0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))

p_rh(i_main,:) = Xi_rh(i_main,:) * & 0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)

! Calculate temperature dependent viscosity and thermal conductivity
   call mu_kappa()
#endif

! Calculate Gbar for the first step of the MacCormack method.
#endif MAKE_LIN
Gbar(:,:,1) = Eta_th * rho0 * VoR
Gbar(:,:,2) = 0.0d0
Gbar(:,:,3) = 0.0d0 !Eta_th * rcp_tubeR * Wb(:,:,Px) / rho0
Gbar(:,:,4) = Eta_th * PoRHOCv0 * VoR
#endif
#endif MAKE_ND
Gbar(:,:,1) = Jac * Eta_th * (rho0 + acs(:,:,RHOx)) * VoR
Gbar(:,:,2) = Gb2*mu0*(v_rh - VoR + u_th*rcp_tubeR)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu0*(rcp_tubeR*(2d0*v_th + & acs(:,:,Ux) - acs(:,:,Vx)*COT) - u_rh))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - k0*rcp_tubeR*t_th)
#endif
#endif MAKE_NDT
Gbar(:,:,1) = Jac * Eta_th * (rho0 + acs(:,:,RHOx)) * VoR
Gbar(:,:,2) = Gb2*mu*(v_rh - VoR + u_th*rcp_tubeR)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(rcp_tubeR*(2d0*v_th + & acs(:,:,Ux) - acs(:,:,Vx)*COT) - u_rh))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*rcp_tubeR*t_th)
#endif
#endif MAKE_ND2
Gbar(:,:,1) = Jac * Eta_th * (rho0 + acs(:,:,RHOx)) * VoR
Gbar(:,:,2) = Gb2*mu*(v_rh - VoR + u_th*rcp_tubeR)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(rcp_tubeR*(2d0*v_th + & acs(:,:,Ux) - acs(:,:,Vx)*COT) - u_rh))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*rcp_tubeR*t_th)
#endif
#endif MAKE_ND2T
Gbar(:,:,1) = Jac * Eta_th * (rho0 + acs(:,:,RHOx)) * VoR
Gbar(:,:,2) = Gb2*mu*(v_rh - VoR + u_th*rcp_tubeR)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*(rcp_tubeR*(2d0*v_th + & acs(:,:,Ux) - acs(:,:,Vx)*COT) - u_rh))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*rcp_tubeR*t_th)
! Calculate SGbar term for first part of MacCormack method.

#ifdef MAKE_LIN
SGbar(:,:,1) = -rho0 * rcp_tubeR * Vsph !+ Density_src*Jac*0.5d0
SGbar(:,:,2) = 0.0d0
SGbar(:,:,3) = 0.0d0 !g_th
SGbar(:,:,4) = -PoRHOCv0 * rcp_tubeR * Vsph !+ Temperature_src*Jac*0.5d0
#endif

#ifdef MAKE_ND
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
(acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
(v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
(rho0+acs(:,:,RHOx))*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
(k*p*rcp_tubeR - CV*acs(:,:,Ux)*(rho0 + acs(:,:,RHOx)))* &
t_rh)/(rho0*CV) + Temperature_src*0.5d0) * Jac
#endif

#ifdef MAKE_NDT
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
(acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
(v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
(rho0+acs(:,:,RHOx))*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
(k*p*rcp_tubeR - CV*acs(:,:,Ux)*(rho0 + acs(:,:,RHOx)))* &
t_rh)/(rho0*CV) + Temperature_src*0.5d0) * Jac
#endif

#ifdef MAKE_ND2
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
(acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
(v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
(rho0+0.0d0)*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
(P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
(2.0d0*k*p*rcp_tubeR - CV*acs(:,:,Ux)*(rho0+acs(:,:,RHOx)))*t_rh + &
mu0*(fourthds*(u_rh*(u_rh - 2.0d0*Vsph - 0.5d0*v_th)) + &
rcp_tubeR*(acs(:,:,Vx)*(rcp_tubeR*(twothds*COT*(acs(:,:,Vx)*COT + &
acs(:,:,Ux)) + acs(:,:,Vx))) - 4.0d0*v_rh) + &
twothds*rcp_tubeR*acs(:,:,Ux)*acs(:,:,Ux)) + &
v_rh*(rcp_tubeR*u_th + 2.0d0*v_rh))/(rho0*Cv) * Jac

#define MAKE_ND2T
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
    (acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
    mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
    (v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
    (rho0+acs(:,:,RHOx))*g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
    (P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
    (kp*rcp_tubeR - Cv*acs(:,:,Ux)))/rho0 + &
    Temperature_src*0.5d0) * Jac

#define MAKE_ND2T
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
    (acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
    mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
    (v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
    (rho0+acs(:,:,RHOx))*g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
    (P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
    (kp*rcp_tubeR - Cv*acs(:,:,Ux)))/rho0 + &
    Temperature_src*0.5d0) * Jac

! Do the first step of MacCormack method.
Wbp(:,j_main,1:4) = Wb(:,j_main,1:4) + deltat * (ver * &
    (Gbar(:,j_main,1:4) - Gbar(:,j_p_ver,1:4)) + &
    SGbar(:,j_main,1:4))

! Pressure (p)
#ifdef MAKE_LIN
Wbp(:,:,Px) = Rgas*( rho0 * Wbp(:,:,Tx) + T0*Wbp(:,:,RHOx) )
#else
Wbp(:,:,Px) = Rgas*( (rho0 + Wbp(:,:,RHOx) * iJac) * Wbp(:,:,Tx) + &
    T0*Wbp(:,:,RHOx) )
#endif

! Use boundary condition to set left and right boundary points.
call BC_eta(Wbp)
#endif MAKE_LIN

#ifdef MAKE_ND
do q = 1, nACSvar
    acs(:,:,q) = Wbp(:,:,q) * iJac
end do
#endif MAKE_ND

#ifdef MAKE_NDT
do q = 1, nACSvar
    acs(:,:,q) = Wbp(:,:,q) * iJac
end do
#endif MAKE_NDT

#ifdef MAKE_ND2
do q = 1, nACSvar
    acs(:,:,q) = Wbp(:,:,q) * iJac
end do
#endif MAKE_ND2

#ifdef MAKE_ND2T
do q = 1, nACSvar
    acs(:,:,q) = Wbp(:,:,q) * iJac
end do
#endif MAKE_ND2T
ifdef MAKE_LIN
  ! Precalculate expressions for second step.
  VoR = Wbp(:,:,Vx) * rcp_tubeR
  Vsph = Wbp(:,:,Vx)*COT
endif

ifdef MAKE_ND
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR
!
! Th-derivatives are "forward" diff wrt "rearward" diff for corrector.
  u_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
  v_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
  t_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))
  call BCgrad_eta(u_th, v_th, t_th)
!
! Rh-derivatives are center differenced.
  u_rh(i_main,:) = Xi_rh(i_main,:) * &
  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_rh(i_main,:) = Xi_rh(i_main,:) * &
  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
  t_rh(i_main,:) = Xi_rh(i_main,:) * &
  0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
  p_rh(i_main,:) = Xi_rh(i_main,:) * &
  0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))
  call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)
endif

ifdef MAKE_NDT
VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR
!
! Th-derivatives are "forward" diff wrt "rearward" diff for corrector.
  u_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
  v_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
  t_th(:,j_main) = Eta_th(:,j_main) * &
  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))
  call BCgrad_eta(u_th, v_th, t_th)
!
! Rh-derivatives are center differenced.
  u_rh(i_main,:) = Xi_rh(i_main,:) * &
  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
  v_rh(i_main,:) = Xi_rh(i_main,:) * &
0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))

\[ t_{rh}(i_{main,:}) = \Xi_{rh}(i_{main,:}) * \]
0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))

\[ p_{rh}(i_{main,:}) = \Xi_{rh}(i_{main,:}) * \]
0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)

! Calculate temperature dependent viscosity and thermal conductivity
  call mu_kappa()

#endif
#ifdef MAKE_ND2

VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Th-derivatives are "forward" diff wrt "rearward" diff for corrector.
u_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))

call BCgrad_eta(u_th, v_th, t_th)

! Rh-derivatives are center differenced.
u_rh(i_main,:) = Xi_rh(i_main,:) * 
  0.5d0 * (acs(i_p_one,:,Ux) - acs(i_m_one,:,Ux))
v_rh(i_main,:) = Xi_rh(i_main,:) * 
  0.5d0 * (acs(i_p_one,:,Vx) - acs(i_m_one,:,Vx))
t_rh(i_main,:) = Xi_rh(i_main,:) * 
  0.5d0 * (acs(i_p_one,:,Tx) - acs(i_m_one,:,Tx))
p_rh(i_main,:) = Xi_rh(i_main,:) * 
  0.5d0 * (acs(i_p_one,:,Px) - acs(i_m_one,:,Px))

call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)

! Calculate temperature dependent viscosity and thermal conductivity
  call mu_kappa()

#endif MAKE_ND2T

VoR = acs(:,:,Vx)*rcp_tubeR
Vsph = (acs(:,:,Ux) + 0.5d0*acs(:,:,Vx)*COT) * rcp_tubeR

! Th-derivatives are "forward" diff wrt "rearward" diff for corrector.
u_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Ux) - acs(:,j_main,Ux))
v_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Vx) - acs(:,j_main,Vx))
t_th(:,j_main) = Eta_th(:,j_main) * 
  ver * (acs(:,j_p_ver,Tx) - acs(:,j_main,Tx))
call BCgrad_eta(u_th, v_th, t_th)

! Rh-derivatives are center differenced.

\[ u_{rh}(i_{main},:) = \Xi_{rh}(i_{main},:) \times 0.5d0 \times (acs(i_{p\_one},:,Ux) - acs(i_{m\_one},:,Ux)) \]

\[ v_{rh}(i_{main},:) = \Xi_{rh}(i_{main},:) \times 0.5d0 \times (acs(i_{p\_one},:,Vx) - acs(i_{m\_one},:,Vx)) \]

\[ t_{rh}(i_{main},:) = \Xi_{rh}(i_{main},:) \times 0.5d0 \times (acs(i_{p\_one},:,Tx) - acs(i_{m\_one},:,Tx)) \]

\[ p_{rh}(i_{main},:) = \Xi_{rh}(i_{main},:) \times 0.5d0 \times (acs(i_{p\_one},:,Px) - acs(i_{m\_one},:,Px)) \]

call BCgrad_xi(u_rh, v_rh, t_rh, p_rh)

! Calculate temperature dependent viscosity and thermal conductivity

call mu_kappa()

#endif

! Calculate the main part of G1, G2, G3, and G4 for the second step of the MacCormack method.
! G* need to be (ni_main+2, nj_main+2) for finite differencing.

#ifdef MAKE_LIN

\[ G_{bar}(i,1) = Eta_{th} \times rho0 \times VoR \]
\[ G_{bar}(i,2) = 0.0d0 \]
\[ G_{bar}(i,3) = 0.0d0 \times Eta_{th} \times rcp_{tubeR} \times Wbp(:,Px) / rho0 \]
\[ G_{bar}(i,4) = Eta_{th} \times Po\text{RHOCv0} \times VoR \]
#endif

#ifdef MAKE_ND

\[ G_{bar}(i,1) = Jac \times Eta_{th} \times (rho0 + acs(:,RHOx)) \times VoR \]
\[ G_{bar}(i,2) = Gb2*mu0*(v_{rh} - VoR + u_{th}*rcp_{tubeR}) \]
\[ G_{bar}(i,3) = Gb3*(acs(:,Px) - twoths*mu0*(rcp_{tubeR}*(2d0*v_{th} + \]
\[ \text{acs(:,Ux)} - \text{acs(:,Vx)*COT}) - u_{rh}) \]
\[ G_{bar}(i,4) = Gb4*(P0*acs(:,Vx) - k0*rcp_{tubeR}*t_{th}) \]
#endif

#ifdef MAKE_NDT

\[ G_{bar}(i,1) = Jac \times Eta_{th} \times (rho0 + acs(:,RHOx)) \times VoR \]
\[ G_{bar}(i,2) = Gb2*mu*(v_{rh} - VoR + u_{th}*rcp_{tubeR}) \]
\[ G_{bar}(i,3) = Gb3*(acs(:,Px) - twoths*mu*(rcp_{tubeR}*(2d0*v_{th} + \]
\[ \text{acs(:,Ux)} - \text{acs(:,Vx)*COT}) - u_{rh}) \]
\[ G_{bar}(i,4) = Gb4*(P0*acs(:,Vx) - k0*rcp_{tubeR}*t_{th}) \]
#endif

#ifdef MAKE_ND2

\[ G_{bar}(i,1) = Jac \times Eta_{th} \times (rho0 + acs(:,RHOx)) \times VoR \]
\[ G_{bar}(i,2) = Gb2*mu*(v_{rh} - VoR + u_{th}*rcp_{tubeR}) \]
\[ G_{bar}(i,3) = Gb3*(acs(:,Px) - twoths*mu*(rcp_{tubeR}*(2d0*v_{th} + \]
\[ \text{acs(:,Ux)} - \text{acs(:,Vx)*COT}) - u_{rh}) \]
\[ G_{bar}(i,4) = Gb4*(P0*acs(:,Vx) - kp*rcp_{tubeR}*t_{th}) \]
#endif

#ifdef MAKE_ND2T

\[ G_{bar}(i,1) = Jac \times Eta_{th} \times (rho0 + acs(:,RHOx)) \times VoR \]
Gbar(:,:,2) = Gb2*mu*(v_rh - VoR + u_th*rcp_tubeR)
Gbar(:,:,3) = Gb3*(acs(:,:,Px) - twothds*mu*rcp_tubeR*(2d0*v_th + &
         acs(:,:,Ux) - acs(:,:,Vx)*COT - u_rh))
Gbar(:,:,4) = Gb4*(P0*acs(:,:,Vx) - kp*rcp_tubeR*t_th)
#endif

! Calculate SGbar term for second part of MacCormack method.
#endif MAKE_LIN
SGbar(:,:,1) = -rho0 * rcp_tubeR *Vsph !+ Density_src*Jac*0.5d0
SGbar(:,:,2) = 0.0d0
SGbar(:,:,3) = 0.0d0 !g_th
SGbar(:,:,4) = -PoRHOCv0 * rcp_tubeR * Vsph !+ Temperature_src*Jac*0.5d0
#endif
#endif MAKE_ND
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
         (acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
         mu0*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
         (v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu0*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
         (rho0+acs(:,:,RHOx))*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
         g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
         (P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
         (k0*rcp_tubeR - Cv*acs(:,:,Ux)*(rho0 + acs(:,:,RHOx)))) * &
         t_rh)/(rho0*Cv) + Temperature_src*0.5d0) * Jac
#endif
#endif MAKE_NDT
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+acs(:,:,RHOx)) * &
         (acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
         mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
         (v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
         (rho0+0.0d0)*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
         g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
         (P0*acs(:,:,RHOx) - acs(:,:,Px)*rho0)*u_rh)/rho0 + &
         (kp*rcp_tubeR - Cv*acs(:,:,Ux)*(rho0 + acs(:,:,RHOx)))) * &
         t_rh)/(rho0*Cv) + Temperature_src*0.5d0) * Jac
#endif
#endif MAKE_ND2
SGbar(:,:,1) = -(rho0 + acs(:,:,RHOx))*Vsph * Jac
SGbar(:,:,2) = (acs(:,:,RHOx)*p_rh/rho0 - (rho0+0.0d0) * &
         (acs(:,:,Ux)*u_rh - 0.5d0*VoR*acs(:,:,Vx)) + &
         mu*rcp_tubeR*(2d0*(2d0*u_rh - Vsph) + &
         (v_rh - 0.5d0*VoR)*COT))/rho0 * Jac
SGbar(:,:,3) = ((mu*rcp_tubeR*(3d0*(v_rh - 0.5d0*VoR) - VoR*COT*COT) - &
         (rho0+0.0d0)*(v_rh+0.5d0*VoR)*acs(:,:,Ux))/rho0 + &
         g_th) * Jac
SGbar(:,:,4) = (((P0*(acs(:,:,RHOx) - rho0) - rho0*acs(:,:,Px))*Vsph + &
\[
(\rho_0 \ast \text{acs}(\cdot, \cdot, \cdot, \cdot) - \text{acs}(\cdot, \cdot, \cdot, \cdot) \ast \rho_0) \ast u_rh)/\rho_0 + (2.0d0 \ast k_p \ast \text{rcp_tubeR} - C_v \ast \text{acs}(\cdot, \cdot, \cdot, \cdot) \ast (\rho_0 + \text{acs}(\cdot, \cdot, \cdot, \cdot))) \ast t_rh + \mu_0 \ast (\text{fourthds} \ast (u_rh \ast (u_rh - 2.0d0 \ast V_sph - 0.5d0 \ast v_th)) + \text{rcp_tubeR} \ast (\text{acs}(\cdot, \cdot, \cdot, \cdot) \ast twothds \ast COT \ast (\text{acs}(\cdot, \cdot, \cdot, \cdot) \ast COT + \text{acs}(\cdot, \cdot, \cdot, \cdot)) - 4.0d0 \ast v_rh) + \text{twothds} \ast \text{rcp_tubeR} \ast \text{acs}(\cdot, \cdot, \cdot, \cdot) \ast \text{acs}(\cdot, \cdot, \cdot, \cdot) + v_rh \ast (\text{rcp_tubeR} \ast u_th + 2.0d0 \ast v_rh))/(\rho_0 \ast C_v) \ast \text{Jac}
\]

#endif

ifdef MAKE_ND2T
SGbar(:,1,1) = -(\rho_0 + \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast V_sph \ast \text{Jac}
SGbar(:,2,1) = (\text{acs}(\cdot, \cdot, \cdot, \cdot) \ast p_rh/\rho_0 - (\rho_0 \ast \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast (\text{acs}(\cdot, \cdot, \cdot, \cdot) \ast u_rh - 0.5d0 \ast V_oR \ast \text{acs}(\cdot, \cdot, \cdot, \cdot)) + \mu_0 \ast \text{rcp_tubeR} \ast (2d0 \ast (2d0 \ast u_rh - V_sph) + \text{v_rh} - 0.5d0 \ast V_oR) \ast \text{COT})/\rho_0 \ast \text{Jac}
SGbar(:,3,1) = ((\mu_0 \ast \text{rcp_tubeR} \ast (3d0 \ast (\text{v_rh} - 0.5d0 \ast V_oR) - V_oR \ast \text{COT} \ast \text{COT}) - (\rho_0 + \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast (\text{v_rh} + 0.5d0 \ast V_oR) \ast \text{acs}(\cdot, \cdot, \cdot, \cdot))/\rho_0 + \text{g_th}) \ast \text{Jac}
SGbar(:,4,1) = (((P_0 \ast (\text{acs}(\cdot, \cdot, \cdot, \cdot) - \rho_0) - \rho_0 \ast \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast V_sph + (P_0 \ast \text{acs}(\cdot, \cdot, \cdot, \cdot) - \text{acs}(\cdot, \cdot, \cdot, \cdot) \ast \rho_0) \ast u_rh)/\rho_0 + (k_p \ast \text{rcp_tubeR} - C_v \ast \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast (\rho_0 + \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast t_rh)/(\rho_0 \ast C_v) + \text{Temperature_src} \ast 0.5d0) \ast \text{Jac}
#endif

! Do the second step of MacCormack method.
Wb(:,j_main,1:4) = 0.5d0 \ast (Wb(:,j_main,1:4) + \text{Wbp}(,j_main,1:4) - \text{deltat} \ast (\text{ver} \ast (\text{Gbar}(,j_main,1:4) - \text{Gbar}(,j_m_ver,1:4)) - \text{SGbar}(,j_main,1:4)))
ifdef MAKE_LIN
Wb(:,Px) = \text{Rgas} \ast (\rho_0 \ast \text{Wb}(,j_main,1:4) + \text{T0} \ast \text{Wb}(,j_main,1:4))
#else
Wb(:,Px) = \text{Rgas} \ast (\rho_0 + \text{acs}(\cdot, \cdot, \cdot, \cdot)) \ast (\text{acs}(\cdot, \cdot, \cdot, \cdot) \ast \text{IJac}) \ast \text{Wb}(,j_main,1:4) \ast \text{T0} + \text{Wb}(,j_main,1:4))
#endif

! Use boundary condition to set left and right boundary points.
call BC_eta(Wb)
ifdef MAKE_IND
do q = 1, nACSvar
acs(:,q) = Wb(:,q) \ast \text{IJac}
end do
#endif
ifdef MAKE_NDT
do q = 1, nACSvar
acs(:,q) = Wb(:,q) \ast \text{IJac}
end do
#endif
ifdef MAKE_ND2
do q = 1, nACSvar
acs(:,q) = Wb(:,q) \ast \text{IJac}
end do
#endif
end do
#endif
#endif

RETURN
END SUBROUTINE FD_OPeta

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BCpred_xi()
!----------------------------------------------------------------------------
! | MOC BC, which is particular to each coordinate system.            | |
! | Predictor step, Xi-direction.                                    | |
!----------------------------------------------------------------------------
USE input_vars
USE boundary
!
implicit none
!
if (end1_BC == 'moc') then
#endif
OSD(:,RHOx) = 0.5d0*(-3d0*Wb(1,:,RHOx)+4d0*Wb(2,:,RHOx)-Wb(3,:,RHOx)) * &
X_i_x1(1,:)
OSD(:,Ux) = 0.5d0*(-3d0*Wb(1,:,Ux)+4d0*Wb(2,:,Ux)-Wb(3,:,Ux))*Xi_x1(1,:)
OSD(:,Tx) = 0.5d0*(-3d0*Wb(1,:,Tx)+4d0*Wb(2,:,Tx)-Wb(3,:,Tx))*Xi_x1(1,:)
L1 = -CO*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - CO*rho0*OSD(:,Tx)
L2 = 0.0d0
L3 = 0.0d0
L4 = 0.0d0
BEQ(:,RHOx) = L2/(rho0*T0*Cp) - (L1+L4)/(2d0*gamma_ratio*T0) - &
rho0*Vsph(1,:)*rcp_tubeR(1,:)
BEQ(:,Ux) = P0*(L1-L4)/(2d0*rho02*T0*C0) ! + g_rh terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = -(0.5d0*P0*(L1+L4) + T0*L2)/(rho02*T0*Cp) - &
P0*Vsph(1,:)*rcp_tubeR(1,:) / (rho0*Cv)
Wbp(1,:,1:4) = Wb(1,:,1:4) + deltat * BEQ
Wb(1,:,Px) = Rgas * (rho0*Wbp(1,:,Tx) + T0*Wbp(1,:,RHOx))
#else
OSD(:,RHOx) = 0.5d0*(-3d0*Wb(1,:,RHOx)+4d0*Wb(2,:,RHOx)-Wb(3,:,RHOx)) * &
X_i_x1(1,:)
OSD(:,Ux) = 0.5d0*(-3d0*Wb(1,:,Ux)+4d0*Wb(2,:,Ux)-Wb(3,:,Ux))*Xi_x1(1,:)
OSD(:,Tx) = 0.5d0*(-3d0*Wb(1,:,Tx)+4d0*Wb(2,:,Tx)-Wb(3,:,Tx))*Xi_x1(1,:)
L1 = -CO*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - CO*rho0*OSD(:,Tx)
L2 = 0.0d0
L3 = 0.0d0
endif
L4 = 0.0d0
L1p = -(C0*T0+acs(1,:,Tx))*OSD(:,RHOx) + & gamma_ratio*(rho0*T0*Wb(ni,:,RHOx)*T0)*OSD(:,Ux) - & C0*(rho0*acs(1,:,RHOx))*OSD(:,Tx)

L2p = 0.0d0
L3p = 0.0d0
L4p = 0.0d0

BEQ(:,RHOx) = (C02*T0*rho0*L2p - 0.5d0*Cp*rho0*T0*P0*(L1p+L4p) - & 0.5d0*Cp*rho0*T0*acs(1,:,Px)*(L1+L4) - & (rho0*acs(1,:,Tx) + T0*acs(1,:,RHOx))*(C02*L2 - & 0.5d0*P0*Cp*(L1+L4)))/(rho0*T0*P0*Cp*C02) - & (rho0*acs(1,:,RHOx))*Vsp(1,:) * Jac(1,:) * Vsph(1,:)/rho02*T0

BEQ(:,Ux) = ((T0*rho0*acs(1,:,P)-rho0*acs(1,:,T)) - & 2d0*P0*acs(1,:,RHOx))*Vsp(1,:)*Jac(1,:)

BEQ(:,Tx) = ((P0*rho0*acs(1,:,P)-rho0*acs(1,:,T)) - & T0*acs(1,:,Px))/2d0*rho02*T0

# ifdef MAKE_LIN

OSD(:,RHOx) = -0.5d0*(-3d0*Wb(ni,:,RHOx) + 4d0*Wb(ni-1,:,RHOx) - & Wb(ni-2,:,RHOx) + Xi_x1(ni,:))

OSD(:,Ux) = -0.5d0*(-3d0*Wb(ni,:,Ux)+4d0*Wb(ni-1,:,Ux)-Wb(ni-2,:,Ux)) * & Xi_x1(ni,:)

OSD(:,Tx) = -0.5d0*(-3d0*Wb(ni,:,T)+4d0*Wb(ni-1,:,T)-Wb(ni-2,:,T)) * & Xi_x1(ni,:)

L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = (C02*T0)*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + C0*rho0*OSD(:,Tx)

BEQ(:,RHOx) = L2/(rho0*T0*Cp) - (L1+L4)/(2d0*gamma_ratio*T0) - & rho0*Vsp(1,:)*rcp_tubeR(ni,:) / (rho0*Cv)

BEQ(:,Ux) = P0*(L1+L4)/(2d0*rho02*T0*C0) ! + g_rh terms

BEQ(:,Vx) = 0.0d0

BEQ(:,Tx) = -(0.5d0*P0*(L1+L4) + T0*L2)/(rho02*T0*Cp) - & P0*Vsp(1,:)*rcp_tubeR(ni,:) / (rho0*Cv)

Wbp(ni,:,1:4) = Wb(ni,:,1:4) + deltat * BEQ

Wbp(ni,:,P) = Rgas * (rho0*Wbp(ni,:,T) + T0*Wbp(ni,:,RHOx))

# else

OSD(:,RHOx) = -0.5d0*(-3d0*Wb(ni,:,RHOx) + 4d0*Wb(ni-1,:,RHOx) - & Wb(ni-2,:,RHOx) + Xi_x1(ni,:))

OSD(:,Ux) = -0.5d0*(-3d0*Wb(ni,:,Ux)+4d0*Wb(ni-1,:,Ux)-Wb(ni-2,:,Ux)) * & Xi_x1(ni,:)

OSD(:,Tx) = -0.5d0*(-3d0*Wb(ni,:,T)+4d0*Wb(ni-1,:,T)-Wb(ni-2,:,T)) * & Xi_x1(ni,:)

L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = (C02*T0)*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + C0*rho0*OSD(:,Tx)

BEQ(:,RHOx) = L2/(rho0*T0*Cp) - (L1+L4)/(2d0*gamma_ratio*T0) - & rho0*Vsp(1,:)*rcp_tubeR(ni,:) / (rho0*Cv)

BEQ(:,Ux) = P0*(L1+L4)/(2d0*rho02*T0*C0) ! + g_rh terms

BEQ(:,Vx) = 0.0d0

BEQ(:,Tx) = -(0.5d0*P0*(L1+L4) + T0*L2)/(rho02*T0*Cp) - & P0*Vsp(1,:)*rcp_tubeR(ni,:) / (rho0*Cv)

Wbp(ni,:,1:4) = Wb(ni,:,1:4) + deltat * BEQ

Wbp(ni,:,P) = Rgas * (rho0*Wbp(ni,:,T) + T0*Wbp(ni,:,RHOx))
\[ \Xi_{x1}(n_i, :) \]

\[
\text{OSD}(:, Tx) = -0.5d0*(-3d0*Wb(ni,:,Tx)+4d0*Wb(ni-1,:,Tx)-Wb(ni-2,:,Tx)) * & \Xi_{x1}(n_i,:) 
\]

\[ L1 = 0.0d0 \]
\[ L2 = 0.0d0 \]
\[ L3 = 0.0d0 \]
\[ L4 = C0*T0*\text{OSD}(:, \text{RHOx}) + \rho0*\text{gamma_ratio}*T0*\text{OSD}(:, Ux) + C0*\rho0*\text{OSD}(:, Tx) \]
\[ L1p = 0.0d0 \]
\[ L2p = 0.0d0 \]
\[ L3p = 0.0d0 \]
\[ L4p = C0*(T0+\text{acs}(n_i,:,Tx))*\text{OSD}(:, \text{RHOx}) + & \text{gamma_ratio}*(\rho0*T0*\rho0*\text{acs}(n_i,:,Tx)+\rho0*, \text{RHOx})*T0)*\text{OSD}(:, Ux) + & C0*(\rho0*\text{acs}(n_i,:,\text{RHOx}))*\text{OSD}(:, Tx) \]
\[ \text{BEQ}(:, \text{RHOx}) = (\rho0*T0*C02*L2p - 0.5d0*\rho0*T0*Cp*(P0*(L1p+L4p) + & \text{acs}(n_i,:,Px)*(L1+L4)) - (\rho0*\text{acs}(n_i,:,Tx) + T0*\text{acs}(n_i,:,\text{RHOx}))* & (C02*L2 - 0.5d0*P0*Cp*(L1+L4)))/(\rho02*T0*T0*C02) - & (\rho0*\text{acs}(n_i,:,\text{RHOx}))*Vsph(ni,:)*Jac(ni,:) \]
\[ \text{BEQ}(:, Ux) = ((T0*\rho0*\text{acs}(n_i,:,Px) - P0*\rho0*\text{acs}(n_i,:,Tx) - & 2d0*T0*P0*\text{acs}(n_i,:,\text{RHOx}))*L1p + & P0*\rho0*T0*(L1p-L4))/(2d0*C0*\rho02*T0*\rho0*T0) \] \[ + \] g_rh terms
\[ \text{BEQ}(:, Vx) = 0.0d0 \]
\[ \text{BEQ}(:, Tx) = ((P0*\rho0*\text{acs}(n_i,:,Tx) + 2d0*P0*T0*\text{acs}(n_i,:,\text{RHOx}) - & T0*\rho0*\text{acs}(n_i,:,Px))*(L1+L4) + 4d0*T0*P0*\text{acs}(n_i,:,\text{RHOx})*L2 - & T0*\rho0*(P0*(L1p+L4) + 2d0*T0*L2p))/(2d0*P0*\rho02*T0*\rho0*T0) - & (P0*\rho0*T0 + P0*\rho0*\text{acs}(n_i,:,Tx) + P0*T0*\text{acs}(n_i,:,\text{RHOx}) + & \rho0*T0*\text{acs}(n_i,:,Px))*Vsph(ni,:)*Jac(ni,:) / (\rho02*T0*Cv) \]
\[ Wbp(ni,:,1:4) = Wb(ni,:,1:4) + \text{deltat} * \text{BEQ} \]
\[ Wbp(ni,:,Px) = Rgas*(\rho0 + Wbp(ni,:,\text{RHOx})*1Jac(ni,:))*Wbp(ni,:,Tx) + & T0*Wbp(ni,:,\text{RHOx}) \]

#endif
end if

if (.not. (end1_BC == 'moc' .and. end2_BC == 'moc')) then
    call BC_xi(Wbp)
end if

RETURN

END SUBROUTINE BCpred_xi

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE BCCorr_xi()
!==========================================================================
!==========================================================================

USE input_vars
USE boundary

implicit none
if (end1_BC == 'moc') then
  #ifdef MAKE_LIN
    OSD(:,RHOx) = 0.5d0*(-3d0*Wbp(1,:,RHOx) + 4d0*Wbp(2,:,RHOx) - & Wbp(3,:,RHOx)) * Xi_x1(1,:)
    OSD(:,Ux) = 0.5d0*(-3d0*Wbp(1,:,Ux)+4d0*Wbp(2,:,Ux)-Wbp(3,:,Ux))*Xi_x1(1,:)
    OSD(:,Tx) = 0.5d0*(-3d0*Wbp(1,:,Tx)+4d0*Wbp(2,:,Tx)-Wbp(3,:,Tx))*Xi_x1(1,:)
    L1 = -C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - C0*rho0*OSD(:,Tx)
    L2 = 0.0d0
    L3 = 0.0d0
    L4 = 0.0d0
    BEQ(:,RHOx) = L2/(rho0*T0*Cp) - (L1+L4)/(2d0*gamma_ratio*T0) - & rho0*Vsph(1,:)*rcp_tubeR(1,:)
    BEQ(:,Ux) = P0*(L1-L4)/(2d0*rho02*T0*C0) ! + g_rh terms
    BEQ(:,Tx) = 0.0d0
    WB(1,:,1:4) = 0.5d0*(WB(1,:,1:4) + WBp(1,:,1:4) + deltat * BEQ)
    WB(1,:,Px) = Rgas * (rho0*WB(1,:,Tx) + T0*WB(1,:,RHOx))
  #else
    OSD(:,RHOx) = 0.5d0*(-3d0*Wbp(1,:,RHOx) + 4d0*Wbp(2,:,RHOx) - & Wbp(3,:,RHOx)) * Xi_x1(1,:)
    OSD(:,Ux) = 0.5d0*(-3d0*Wbp(1,:,Ux)+4d0*Wbp(2,:,Ux)-Wbp(3,:,Ux))*Xi_x1(1,:)
    OSD(:,Tx) = 0.5d0*(-3d0*Wbp(1,:,Tx)+4d0*Wbp(2,:,Tx)-Wbp(3,:,Tx))*Xi_x1(1,:)
    L1 = -C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) - C0*rho0*OSD(:,Tx)
    L2 = 0.0d0
    L3 = 0.0d0
    L4 = 0.0d0
    L1p = -C0*(T0+acs(1,:,Tx))*OSD(:,RHOx) + & gamma_ratio*(rho0*T0*rho0*acs(1,:,Tx)+acs(1,:,RHOx)*T0)*OSD(:,Ux) - & C0*(rho0*rho0*acs(1,:,RHOx))*OSD(:,Tx)
    L2p = 0.0d0
    L3p = 0.0d0
    L4p = 0.0d0
    BEQ(:,RHOx) = (C02*T0*rho0*L2p - 0.5d0*Cp*rho0*T0*P0*(L1p+L4p) - & 0.5d0*Cp*rho0*T0*acs(1,:,Px)*(L1+L4) - & (rho0*acs(1,:,Tx) + T0*acs(1,:,RHOx))*(C02*L2 - & 0.5d0*P0*Cp*(L1+L4)))/(rho02*T0*Cp*C02) - & (rho0*acs(1,:,RHOx))*Vsph(1,:)*Jac(1,:)
    BEQ(:,Ux) = ((T0*rho0*acs(1,:,Px) - P0*rho0*acs(1,:,Tx) - & 2d0*T0*P0*acs(1,:,RHOx))*(L1-L4) + & P0*rho0*T0*(L1-L4))/(2d0*C0*rho02*T0*rho0*T0) ! + g_z terms
    BEQ(:,Tx) = (P0*rho0*acs(1,:,Tx) + 2d0*P0*T0*acs(1,:,RHOx)) - & T0*rho0*acs(1,:,P0)*(L1-L4) + 4d0*T0*T0*acs(1,:,RHOx)*L2 - & (P0*acs(1,:,P0)*acs(1,:,Tx) + P0*acs(1,:,RHOx) + & rho0*T0*acs(1,:,Px))*Vsph(1,:)*Jac(1,:) / (rho02*T0*Cp)
    WB(1,:,1:4) = 0.5d0*(WB(1,:,1:4) + WBp(1,:,1:4) + deltat * BEQ)
    WB(1,:,Px) = Rgas * (rho0 + WB(1,:,RHOx)*iJac(1,:))*WB(1,:,Tx) + & T0*WB(1,:,RHOx))
  #endif
endif
end if
if (end2_BC == 'moc') then

#ifdef MAKE_LIN
OSD(:,RHOx) = -0.5d0*(-3d0*Wbp(ni,:,RHOx) + 4d0*Wbp(ni-1,:,RHOx) - &
Wbp(ni-2,:,RHOx)) * Xi_x1(ni,:
OSD(:,Ux) = -0.5d0*(-3d0*Wbp(ni,:,Ux) + 4d0*Wbp(ni-1,:,Ux) - &
Wbp(ni-2,:,Ux)) * Xi_x1(ni,:
OSD(:,Tx) = -0.5d0*(-3d0*Wbp(ni,:,Tx) + 4d0*Wbp(ni-1,:,Tx) - &
Wbp(ni-2,:,Tx)) * Xi_x1(ni,:
L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + CO*rho0*OSD(:,Tx)
BEQ(:,RHOx) = L2/(rho0*T0*Cp) - (L1+L4)/(2d0*gamma_ratio*T0) - &
rho0*Vsph(ni,:)*rcp_tubeR(ni,:
BEQ(:,Ux) = P0*(L1-L4)/(2d0*rho02*T0*CO) ! + g_rh terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = -(0.5d0*P0*(L1+L4) + T0*L2)/(rho02*T0*Cp) &
P0*Vsph(ni,:)*rcp_tubeR(ni,:
Wb(ni,:,1:4) = 0.5d0*(Wb(ni,:,1:4) + deltat * BEQ)
Wb(ni,:,Px) = Rgas * (rho0*Wb(ni,:,Tx) + T0*Wb(ni,:,RHOx))
#endif

else
OSD(:,RHOx) = -0.5d0*(-3d0*Wbp(ni,:,RHOx) + 4d0*Wbp(ni-1,:,RHOx) - &
Wbp(ni-2,:,RHOx)) * Xi_x1(ni,:
OSD(:,Ux) = -0.5d0*(-3d0*Wbp(ni,:,Ux) + 4d0*Wbp(ni-1,:,Ux) - &
Wbp(ni-2,:,Ux)) * Xi_x1(ni,:
OSD(:,Tx) = -0.5d0*(-3d0*Wbp(ni,:,Tx) + 4d0*Wbp(ni-1,:,Tx) - &
Wbp(ni-2,:,Tx)) * Xi_x1(ni,:
L1 = 0.0d0
L2 = 0.0d0
L3 = 0.0d0
L4 = C0*T0*OSD(:,RHOx) + rho0*gamma_ratio*T0*OSD(:,Ux) + CO*rho0*OSD(:,Tx)
L1p = 0.0d0
L2p = 0.0d0
L3p = 0.0d0
L4p = C0*(T0+acs(ni,:,Tx))*OSD(:,RHOx) + &
gamma_ratio*(rho0*T0+rho0*acs(ni,:,Tx)+acs(ni,:,RHOx)*T0)*OSD(:,Ux) + &
CO*(rho0+acs(ni,:,RHOx)))*OSD(:,Tx)
BEQ(:,RHOx) = (rho0*T0+C02*L2p - 0.5d0*rho0*T0*Cs*P0*(L1p+L4p) + &
acs(ni,:,Px)*(L1+L4) - (rho0*acs(ni,:,Tx) + T0*acs(ni,:,RHOx)) * &
(C02*L2 -0.5d0*P0*Cs*(L1+L4)))/(rho02*T0*Cp*C02) &
(acs(ni,:,RHOx)))*Vsph(ni,:)*Jac(ni,:
BEQ(:,Ux) = (((T0*rho0*acs(ni,:,Px) - P0*rho0*acs(ni,:,Tx) - &
2d0*P0*acs(ni,:,RHOx))*(L1-L4) + &
P0*rho0*T0*(L1-L4))/((2d0*CO*rho02*T0*rho0*CO) ! + g_rh terms
BEQ(:,Vx) = 0.0d0
BEQ(:,Tx) = ((P0*rho0*acs(ni,:,Tx) + 2d0*P0*T0*acs(ni,:,RHOx) - &
T0*rho0*acs(ni,:,Px))*(L1+L4) + 4d0*T0*T0*acs(ni,:,RHOx)*L2 - &
T0*rho0*(P0*(L1p+L4p) + 2d0*T0*L2p))/(2d0*Cp*rho02*T0*rho0*T0) &
(P0*rho0*T0 + P0*rho0*acs(ni,:,Tx) + P0*T0*acs(ni,:,RHOx) + &
rho0*T0*acs(ni,:,Px)))*Vsph(ni,:)*Jac(ni,/) / (rho02*T0*Cv)
$Wb(ni,:,1:4) = 0.5d0*(Wb(ni,:,1:4) + Wbp(ni,:,1:4) + \text{deltat} \times \text{BEQ})$

$Wb(ni,:,Px) = \text{Rgas} \times (\rho_0 + Wb(ni,:,RHOx) \times iJac(ni,:) \times Wb(ni,:,Tx) + T0 \times Wb(ni,:,RHOx))$

```fortran
#endif
end if

if (.not.(end1_BC == 'moc' .and. end2_BC == 'moc')) then
    call BC_xi(Wb)
end if

if (wall_TBC == 'iso') then
    Wb(:,nj,Tx) = 0.0d0
end if

RETURN

END SUBROUTINE BCcorr_xi
```

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------

```fortran
SUBROUTINE mu_kappa()
!======================================================================
! | Calculate temperature dependent viscosity and thermal conductivity. |
!======================================================================
implicit none
real(DP) :: C1 = 70.0d0 ! Kelvin
real(DP) :: C2 = 33.0d0 ! Kelvin

#ifdef MAKE_NDT
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif
#ifdef MAKE_ND2
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif
#ifdef MAKE_ND2T
mu = mu0*(T0+C1)/(T0+acs(:,:,Tx)+C1) * ((T0+acs(:,:,Tx))/T0)**1.5d0
kp = mu*Cp / Prandtl
#endif

! kp = k0*(T0+C2)/(T0+acs(:,:,Tx)+C2) * (T0+acs(:,:,Tx))/T0)**1.5d0

RETURN

END SUBROUTINE mu_kappa
```

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE alloc_localvars()
  !  =======================================================================
  !  | Allocate memory for the arrays local to module mod_maccormack_lin.  |
  !  =======================================================================

  integer :: astat

#define MAKE_LIN
  astat = 0
  allocate(COT(ni,nj), STAT=astat)
  allocate(VoR(ni,nj), STAT=astat)
  allocate(Vsph(ni,nj), STAT=astat)
#undef MAKE_LIN
#define MAKE_ND
  astat = 0
  allocate(COT(ni,nj), STAT=astat)
  allocate(Vsph(ni,nj), STAT=astat)
  allocate(VoR(ni,nj), STAT=astat)
  allocate(u_rh(ni,nj), STAT=astat)
  allocate(u_th(ni,nj), STAT=astat)
  allocate(v_rh(ni,nj), STAT=astat)
  allocate(v_th(ni,nj), STAT=astat)
  allocate(t_rh(ni,nj), STAT=astat)
  allocate(t_th(ni,nj), STAT=astat)
  allocate(p_rh(ni,nj), STAT=astat)
  allocate(p_th(ni,nj), STAT=astat)
  allocate(Fb2(ni,nj), STAT=astat)
  allocate(Fb3(ni,nj), STAT=astat)
  allocate(Fb4(ni,nj), STAT=astat)
  allocate(Gb2(ni,nj), STAT=astat)
  allocate(Gb3(ni,nj), STAT=astat)
#undef MAKE_ND
#define MAKE_NDT
  astat = 0
  allocate(COT(ni,nj), STAT=astat)
  allocate(Vsph(ni,nj), STAT=astat)
  allocate(VoR(ni,nj), STAT=astat)
  allocate(u_rh(ni,nj), STAT=astat)
  allocate(u_th(ni,nj), STAT=astat)
  allocate(v_rh(ni,nj), STAT=astat)
  allocate(v_th(ni,nj), STAT=astat)
  allocate(t_rh(ni,nj), STAT=astat)
  allocate(t_th(ni,nj), STAT=astat)
  allocate(p_rh(ni,nj), STAT=astat)
  allocate(p_th(ni,nj), STAT=astat)
  allocate(Fb2(ni,nj), STAT=astat)
  allocate(Fb3(ni,nj), STAT=astat)
  allocate(Fb4(ni,nj), STAT=astat)
  allocate(Gb2(ni,nj), STAT=astat)
  allocate(Gb3(ni,nj), STAT=astat)
#undef MAKE_NDT
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif
#endif MAKE_ND2
astat = 0
allocate(COT(ni,nj), STAT=astat)
allocate(Vsph(ni,nj), STAT=astat)
allocate(VoR(ni,nj), STAT=astat)
allocate(u_rh(ni,nj), STAT=astat)
allocate(u_th(ni,nj), STAT=astat)
allocate(v_rh(ni,nj), STAT=astat)
allocate(v_th(ni,nj), STAT=astat)
allocate(t_rh(ni,nj), STAT=astat)
allocate(t_th(ni,nj), STAT=astat)
allocate(p_rh(ni,nj), STAT=astat)
allocate(p_th(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif MAKE_ND2T
astat = 0
allocate(COT(ni,nj), STAT=astat)
allocate(Vsph(ni,nj), STAT=astat)
allocate(VoR(ni,nj), STAT=astat)
allocate(u_rh(ni,nj), STAT=astat)
allocate(u_th(ni,nj), STAT=astat)
allocate(v_rh(ni,nj), STAT=astat)
allocate(v_th(ni,nj), STAT=astat)
allocate(t_rh(ni,nj), STAT=astat)
allocate(t_th(ni,nj), STAT=astat)
allocate(p_rh(ni,nj), STAT=astat)
allocate(p_th(ni,nj), STAT=astat)
allocate(Fb2(ni,nj), STAT=astat)
allocate(Fb3(ni,nj), STAT=astat)
allocate(Fb4(ni,nj), STAT=astat)
allocate(Gb2(ni,nj), STAT=astat)
allocate(Gb3(ni,nj), STAT=astat)
allocate(Gb4(ni,nj), STAT=astat)
allocate(mu(ni,nj), STAT=astat)
allocate(kp(ni,nj), STAT=astat)
#endif
if (.not. (end1_BC /= 'moc' .and. end2_BC /= 'moc')) then
  astat = 0
allocate(OSD(nj,4), STAT=astat)
allocate(L1(nj), STAT=astat)
allocate(L2(nj), STAT=astat)
allocate(L3(nj), STAT=astat)
allocate(L4(nj), STAT=astat)
allocate(BEQ(nj,4), STAT=astat)

ifdef MAKE_LIN
allocate(L1p(nj), STAT=astat)
allocate(L2p(nj), STAT=astat)
allocate(L3p(nj), STAT=astat)
allocate(L4p(nj), STAT=astat)
endif

if (astat > 0) print *, "Problem allocating memory for MOC BC."
end if

RETURN

END SUBROUTINE alloc_localvars

SUBROUTINE dealloc_localvars()
!
! Deallocate memory for the arrays local to module mod_maccormack_lin.
!
!
integer :: astat

ifdef MAKE_LIN
astat = 0
dereallocate(COT, STAT=astat)
dereallocate(VoR, STAT=astat)
dereallocate(Vsph, STAT=astat)
endif
ifdef MAKE_ND
astat = 0
dereallocate(COT, STAT=astat)
dereallocate(Vsph, STAT=astat)
dereallocate(VoR, STAT=astat)
dereallocate(u_rh, STAT=astat)
dereallocate(u_th, STAT=astat)
dereallocate(v_rh, STAT=astat)
dereallocate(v_th, STAT=astat)
dereallocate(t_rh, STAT=astat)
dereallocate(t_th, STAT=astat)
dereallocate(p_rh, STAT=astat)
dereallocate(p_th, STAT=astat)
dereallocate(Fb2, STAT=astat)
dereallocate(Fb3, STAT=astat)
dereallocate(Fb4, STAT=astat)
endif
deallocate(Gb2, STAT=astat)
deallocate(Gb3, STAT=astat)
deallocate(Gb4, STAT=astat)
#endif
#ifdef MAKE_NDT
astat = 0
   deallocate(COT, STAT=astat)
deallocate(Vsph, STAT=astat)
deallocate(VoR, STAT=astat)
deallocate(u_rh, STAT=astat)
deallocate(u_th, STAT=astat)
deallocate(v_rh, STAT=astat)
deallocate(v_th, STAT=astat)
deallocate(t_rh, STAT=astat)
deallocate(t_th, STAT=astat)
deallocate(p_rh, STAT=astat)
deallocate(p_th, STAT=astat)
deallocate(Fb2, STAT=astat)
deallocate(Fb3, STAT=astat)
deallocate(Fb4, STAT=astat)
deallocate(Gb2, STAT=astat)
deallocate(Gb3, STAT=astat)
deallocate(Gb4, STAT=astat)
deallocate(mu, STAT=astat)
deallocate(kp, STAT=astat)
#endif
#ifdef MAKE_ND2
astat = 0
   deallocate(COT, STAT=astat)
deallocate(Vsph, STAT=astat)
deallocate(VoR, STAT=astat)
deallocate(u_rh, STAT=astat)
deallocate(u_th, STAT=astat)
deallocate(v_rh, STAT=astat)
deallocate(v_th, STAT=astat)
deallocate(t_rh, STAT=astat)
deallocate(t_th, STAT=astat)
deallocate(p_rh, STAT=astat)
deallocate(p_th, STAT=astat)
deallocate(Fb2, STAT=astat)
deallocate(Fb3, STAT=astat)
deallocate(Fb4, STAT=astat)
deallocate(Gb2, STAT=astat)
deallocate(Gb3, STAT=astat)
deallocate(Gb4, STAT=astat)
deallocate(mu, STAT=astat)
deallocate(kp, STAT=astat)
#endif
#ifdef MAKE_ND2T
astat = 0
   deallocate(COT, STAT=astat)
#endif
deallocate(Vsph, STAT=astat)
deallocate(VoR, STAT=astat)
deallocate(u_rh, STAT=astat)
deallocate(u_th, STAT=astat)
deallocate(v_rh, STAT=astat)
deallocate(v_th, STAT=astat)
deallocate(t_rh, STAT=astat)
deallocate(t_th, STAT=astat)
deallocate(p_rh, STAT=astat)
deallocate(p_th, STAT=astat)
deallocate(Fb2, STAT=astat)
deallocate(Fb3, STAT=astat)
deallocate(Fb4, STAT=astat)
deallocate(Gb2, STAT=astat)
deallocate(Gb3, STAT=astat)
deallocate(Gb4, STAT=astat)
deallocate(mu, STAT=astat)
deallocate(kp, STAT=astat)
#endif
if (.not. (end1_BC /= 'moc' .and. end2_BC /= 'moc')) then
  astat = 0
  deallocate(OSD, STAT=astat)
deallocate(L1, STAT=astat)
deallocate(L2, STAT=astat)
deallocate(L3, STAT=astat)
deallocate(L4, STAT=astat)
deallocate(BEQ, STAT=astat)
#endif MAKE_LIN
#else
  deallocate(L1p, STAT=astat)
deallocate(L2p, STAT=astat)
deallocate(L3p, STAT=astat)
deallocate(L4p, STAT=astat)
#endif
if (astat > 0) print *, "Problem deallocating memory for MOC BC."
end if
RETURN
END SUBROUTINE dealloc_localvars
!---------------------------------------------------------------------------
END MODULE maccormack_sph
D.4.14 mpifun.fpp

MODULE mpi_fun

!============================================================================
! Here we have the MPI routines. Those ending with lower case _mpi
! are defined here. Those beginning with upper case MPI_ are calls to
! the MPI libraries.
!
! Table of contents:
!  SUBROUTINE orient_mpi()
!  SUBROUTINE share_input_mpi()
!  SUBROUTINE make_input_mpitype()
!  SUBROUTINE share_bounds_mpi(give_down, give_up, take_up, take_down)
!  SUBROUTINE share_lobound_mpi(give_down, take_up)
!  SUBROUTINE share_hibound_mpi(give_up, take_down)
!  SUBROUTINE sync_mpi()
!  SUBROUTINE cleanup_mpi()
!============================================================================

USE global

implicit none

include 'mpif.h'

integer, parameter :: MASTER = 0
integer, parameter :: uptown = 111 ! Tag for upward bound messages
integer, parameter :: downtown = 222 ! Tag for downward bound messages

integer(i4b) :: mpierr ! MPI error flag
integer(i4b) :: myid ! process rank
integer(i4b) :: nprocs ! number of processors
integer(i4b) :: nprocs_x1, nprocs_x2 ! number of processors in each dim
integer(i4b) :: myupper_x1, mylower_x1 ! high and low neighbors (X1-dir)
integer(i4b) :: myupper_x2, mylower_x2 ! high and low neighbors (X2-dir)
integer(i4b) :: mpi_input_type
integer(i4b) :: mpi_acsdata_type
integer(i4b), dimension(:), allocatable :: reqs ! nonblocking requests
integer(i4b), dimension(MPI_STATUS_SIZE) :: mpistat
integer(i4b) :: rqx ! request index
integer(i4b) :: nrq ! number of requests

interface share_bounds_mpi
   module procedure share_bounds1D_mpi, share_bounds2D_mpi
end interface

interface share_lobound_mpi
   module procedure share_lobound1D_mpi, share_lobound2D_mpi
end interface

interface share_hibound_mpi
   module procedure share_hibound1D_mpi, share_hibound2D_mpi
end interface
SUBROUTINE orient_mpi()

! This subroutine gets information about the parallel environment.

implicit none

integer :: istat

call MPI_init(mpierr)
call MPI_comm_rank(MPI_COMM_WORLD, myid, mpierr)
call MPI_comm_size(MPI_COMM_WORLD, nprocs, mpierr)
if (myid == MASTER) print *, "There are ", nprocs, " processors."

prdigs = log10(real(nprocs)) + 1

nprocs_x1 = nprocs
nprocs_x2 = 1

myupper_x1 = mod(nprocs_x1+myid+1,nprocs_x1)
mylower_x1 = mod(nprocs_x1+myid-1,nprocs_x1)

if (myid == MASTER .or. myid == nprocs-1) then
  nrq = 2
else
  nrq = 4
end if

allocate(reqs(nrq), STAT=istat)
if (istat > 0) then
  print *, "Problem allocating reqs for orient_mpi()."
  STOP
end if

rqx = 1
RETURN
END SUBROUTINE orient_mpi

SUBROUTINE share_input_mpi()
USE input_vars

implicit none

call make_input_mpitype()
call MPI_bcast(radius, 1, mpi_input_type, MASTER, MPI_COMM_WORLD, mpierr)

RETURN

END SUBROUTINE share_input_mpi

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE make_input_mpitype()

USE input_vars

implicit none

integer, parameter :: nvar = 64 ! number of input variables
integer, dimension(nvar) :: block_lengths ! number of elements per block
integer, dimension(nvar) :: displacements
integer, dimension(nvar) :: typelist
integer :: start_address
integer :: address

! Set the number of elements in each block.
block_lengths = (/ 1,1,1,1,1,1,1,1,1, &
1,1,1,1,1,1,1,1,1, &
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, &
3,3,3,3,3,3,3,3,2,2,2, &
1,1,1, &
3,3,3, &
1,1,1,1,1,1,1,1,1,&
13,3,3,1,1,1 /)

! Specify the datatypes of each block.
typelist(1:10) = MPI_DOUBLE_PRECISION
typelist(11:17) = MPI_INTEGER
typelist(18:32) = MPI_DOUBLE_PRECISION
typelist(33:43) = MPI_CHARACTER
typelist(44:46) = MPI_LOGICAL
typelist(47:49) = MPI_CHARACTER
typelist(50:58) = MPI_INTEGER
typelist(59) = MPI_CHARACTER
typelist(60:61) = MPI_INTEGER
typelist(62:64) = MPI_DOUBLE_PRECISION

! First element is at displacement 0.
    displacements(1) = 0

! Calculate other displacements relative to the first.
call MPI_Address(radius, start_address, mpierr)
call MPI_Address(L, address, mpierr)
displacements(2) = address - start_address
call MPI_Address(taper, address, mpierr)
displacements(3) = address - start_address
call MPI_Address(CFL, address, mpierr)
displacements(4) = address - start_address
call MPI_Address(min_delta1_x1, address, mpierr)
displacements(5) = address - start_address
call MPI_Address(min_delta2_x1, address, mpierr)
displacements(6) = address - start_address
call MPI_Address(min_delta1_x2, address, mpierr)
displacements(7) = address - start_address
call MPI_Address(min_delta2_x2, address, mpierr)
displacements(8) = address - start_address
call MPI_Address(max_delta_x1, address, mpierr)
displacements(9) = address - start_address
call MPI_Address(max_delta_x2, address, mpierr)
displacements(10) = address - start_address
call MPI_Address(nBL, address, mpierr)
displacements(11) = address - start_address
call MPI_Address(i_size, address, mpierr)
displacements(12) = address - start_address
call MPI_Address(j_size, address, mpierr)
displacements(13) = address - start_address
call MPI_Address(txmin, address, mpierr)
displacements(14) = address - start_address
call MPI_Address(txmax, address, mpierr)
displacements(15) = address - start_address
call MPI_Address(ic_choice, address, mpierr)
displacements(16) = address - start_address
call MPI_Address(SPL, address, mpierr)
displacements(17) = address - start_address
call MPI_Address(theta, address, mpierr)
displacements(18) = address - start_address
call MPI_Address(k, address, mpierr)
displacements(19) = address - start_address
call MPI_Address(bigB, address, mpierr)
displacements(20) = address - start_address
call MPI_Address(S_0, address, mpierr)
displacements(21) = address - start_address
call MPI_Address(S_1, address, mpierr)
displacements(22) = address - start_address
call MPI_Address(S_2, address, mpierr)
displacements(23) = address - start_address
call MPI_Address(fsrc, address, mpierr)
displacements(24) = address - start_address
call MPI_Address(phi, address, mpierr)
displacements(25) = address - start_address
call MPI_Address(src_pos1, address, mpierr)
displacements(26) = address - start_address
call MPI_Address(src_pos2, address, mpierr)
displacements(27) = address - start_address
call MPI_Address(SPpeak, address, mpierr)
displacements(28) = address - start_address
call MPI_Address(SAramp_rate, address, mpierr)
displacements(29) = address - start_address
call MPI_Address(SAramp_dur, address, mpierr)
displacements(30) = address - start_address
call MPI_Address(exagT, address, mpierr)
displacements(31) = address - start_address
call MPI_Address(exagV, address, mpierr)
displacements(32) = address - start_address
call MPI_Address(fdrv_type, address, mpierr)
displacements(33) = address - start_address
call MPI_Address(wall_BC, address, mpierr)
displacements(34) = address - start_address
call MPI_Address(ctr_BC, address, mpierr)
displacements(35) = address - start_address
call MPI_Address(end1_BC, address, mpierr)
displacements(36) = address - start_address
call MPI_Address(end2_BC, address, mpierr)
displacements(37) = address - start_address
call MPI_Address(wall_TBC, address, mpierr)
displacements(38) = address - start_address
call MPI_Address(end1_TBC, address, mpierr)
displacements(39) = address - start_address
call MPI_Address(end2_TBC, address, mpierr)
displacements(40) = address - start_address
call MPI_Address(wall_VBC, address, mpierr)
displacements(41) = address - start_address
call MPI_Address(end1_VBC, address, mpierr)
displacements(42) = address - start_address
call MPI_Address(end2_VBC, address, mpierr)
displacements(43) = address - start_address
call MPI_Address(gravity, address, mpierr)
displacements(44) = address - start_address
call MPI_Address(analytic, address, mpierr)
displacements(45) = address - start_address
call MPI_Address(probing, address, mpierr)
displacements(46) = address - start_address
call MPI_Address(streamout, address, mpierr)
displacements(47) = address - start_address
call MPI_Address(output, address, mpierr)
displacements(48) = address - start_address
call MPI_Address(outfmt, address, mpierr)
displacements(49) = address - start_address
call MPI_Address(outarg1, address, mpierr)
displacements(50) = address - start_address
call MPI_Address(outarg2, address, mpierr)
displacements(51) = address - start_address
call MPI_Address(datapos, address, mpierr)
displacements(52) = address - start_address
call MPI_Address(prb_xi, address, mpierr)
displacements(53) = address - start_address
call MPI_Address(prb_eta, address, mpierr)
displacements(54) = address - start_address
call MPI_Address(prb_start, address, mpierr)
displacements(55) = address - start_address
call MPI_Address(prb_stop, address, mpierr)
displacements(56) = address - start_address
call MPI_Address(prb_per_cyc, address, mpierr)
displacements(57) = address - start_address
call MPI_Address(restint, address, mpierr)
displacements(58) = address - start_address
call MPI_Address(setname, address, mpierr)
displacements(59) = address - start_address
call MPI_Address(drv_pdur, address, mpierr)
displacements(60) = address - start_address
call MPI_Address(drv_rint, address, mpierr)
displacements(61) = address - start_address
call MPI_Address(P_tgt, address, mpierr)
displacements(62) = address - start_address
call MPI_Address(Req_pos, address, mpierr)
displacements(63) = address - start_address
call MPI_Address(Uvol_pos, address, mpierr)
displacements(64) = address - start_address

! Build the datatype.
call MPI_Type_struct(nvar, block_lengths, displacements, typelist, &
                   mpi_input_type, mpierr)

! Commit the datatype.
call MPI_Type_commit(mpi_input_type, mpierr)

RETURN

END SUBROUTINE make_input_mpitype

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE find_x1pos_mpi(distances, proc_id)

! -----------------------------------------------
! | Find the processor with the minimum distance along the x1 grid from |
! | a defined x1 position. | |
!----------------------------------------------------------------------------

implicit none

real(DP), dimension(:), intent(IN) :: distances
integer(i4b), intent(OUT) :: proc_id
integer(i4b), dimension(1) :: proc_idx
real(DP), dimension(nprocs) :: min_dists

min_dists = 0.0d0
min_dists(myid+1) = minval(distances)

call MPI_allreduce(MPI_IN_PLACE, min_dists, nprocs, &
                MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, mpierr)

proc_idx = minloc(min_dists)
proc_id = proc_idx(1) - 1

RETURN

END SUBROUTINE find_x1pos_mpi

!----------------------------------------------------------------------------

SUBROUTINE get_real_mpi(real_num, sender)
!----------------------------------------------------------------------------

! | Send a real number from specified processor to MASTER node. |
!----------------------------------------------------------------------------

implicit none

real(DP), intent(INOUT) :: real_num
integer(i4b), intent(IN) :: sender
integer(i4b), parameter :: tag = 10

if (myid == sender) then
    call MPI_send(real_num, 1, MPI_DOUBLE_PRECISION, MASTER, tag, &
              MPI_COMM_WORLD, mpierr)
elseif (myid == MASTER) then
    call MPI_recv(real_num, 1, MPI_DOUBLE_PRECISION, sender, tag, &
                   MPI_COMM_WORLD, mpistat, mpierr)
end if

! Everyone wait here.
call MPI_Barrier(MPI_COMM_WORLD, mpierr)

RETURN
End subroutine get_real_mpi
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
 implicit none

real(DP), dimension(:,,:), intent(IN) :: give_up2, give_down2
real(DP), dimension(:,,:), intent(OUT) :: take_up2, take_down2
integer(i4b) :: ngvup, ngvdn, ntkup, ntkdn
integer, parameter :: nreq = 2
integer(i4b), dimension(nreq) :: requests
integer, dimension(MPI_STATUS_SIZE,nreq) :: stats

ngvup = size(give_up2,1)*size(give_up2,2)
ngvdn = size(give_down2,1)*size(give_down2,2)
tkup = size(take_up2,1)*size(take_up2,2)
tkdn = size(take_down2,1)*size(take_down2,2)

call MPI_isend(give_down2, ngvdn, MPI_DOUBLE_PRECISION, mylower_x1, &
downtown, MPI_COMM_WORLD, requests(1), mpierr)
call MPI_isend(give_up2, ngvup, MPI_DOUBLE_PRECISION, myupper_x1, &
uptown, MPI_COMM_WORLD, requests(2), mpierr)
call MPI_recv(take_up2, ntkup, MPI_DOUBLE_PRECISION, mylower_x1, &
uptown, MPI_COMM_WORLD, mpistat, mpierr)
call MPI_recv(take_down2, ntkdn, MPI_DOUBLE_PRECISION, myupper_x1, &
downtown, MPI_COMM_WORLD, mpistat, mpierr)
call MPI_waitall(nreq, requests, stats, mpierr)

RETURN

END SUBROUTINE share_bounds2D_mpi

! *---------------------------------------------------------------------

SUBROUTINE share_lobound2D_mpi(give_down2, take_up2)
 ! | Share boundary information with neighboring lower subdomain. |
 ! | Use nonblocking MPI sends and blocking receives. |
 ! *---------------------------------------------------------------------
 implicit none

real(DP), dimension(:,,:), intent(IN) :: give_down2
real(DP), dimension(:,,:), intent(OUT) :: take_up2
integer(i4b) :: ngvdn, ntkup
integer, parameter :: nreq = 1
integer(i4b) :: requests
integer, dimension(MPI_STATUS_SIZE) :: stats
ngvdn = size(give_down2,1)*size(give_down2,2)
ntkup = size(take_up2,1)*size(take_up2,2)

call MPI_isend(give_down2, ngvdn, MPI_DOUBLE_PRECISION, mylower_x1, &
downtown, MPI_COMM_WORLD, requests, mpierr)

call MPI_recv(take_up2, ntkup, MPI_DOUBLE_PRECISION, mylower_x1, &
uptown, MPI_COMM_WORLD, mpistat, mpierr)

call MPI_waitall(nreq, requests, stats, mpierr)

RETURN

END SUBROUTINE share_lobound2D_mpi

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE share_lobound1D_mpi(give_down1, take_up1)
! =======================================================================
! | Share boundary information with neighboring lower subdomain. |
! | Use nonblocking MPI sends and blocking receives. |
! =======================================================================

implicit none

real(DP), dimension(:,), intent(IN) :: give_down1
real(DP), dimension(:,), intent(OUT) :: take_up1
integer(i4b) :: ngvdn, ntkup
integer, parameter :: nreq = 1
integer(i4b) :: requests
integer, dimension(MPI_STATUS_SIZE) :: stats

ngvdn = size(give_down1,1)
ntkup = size(take_up1,1)

call MPI_isend(give_down1, ngvdn, MPI_DOUBLE_PRECISION, mylower_x1, &
downtown, MPI_COMM_WORLD, requests, mpierr)

call MPI_recv(take_up1, ntkup, MPI_DOUBLE_PRECISION, mylower_x1, &
uptown, MPI_COMM_WORLD, mpistat, mpierr)

call MPI_waitall(nreq, requests, stats, mpierr)

RETURN

END SUBROUTINE share_lobound1D_mpi

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE share_hibound2D_mpi(give_up2, take_down2)
! =======================================================================
! | Share boundary information with neighboring upper subdomain. | !
! | Use nonblocking MPI sends and blocking receives. | !
!======================================================================

implicit none

real(DP), dimension(:,,:), intent(IN) :: give_up2
real(DP), dimension(:,,:), intent(OUT) :: take_down2
integer(i4b) :: ngvup, ntkdn
integer, parameter :: nreq = 1
integer(i4b) :: requests
integer, dimension(MPI_STATUS_SIZE) :: stats

ngvup = size(give_up2,1)*size(give_up2,2)
ntkdn = size(take_down2,1)*size(take_down2,2)

call MPI_isend(give_up2, ngvup, MPI_DOUBLE_PRECISION, myupper_x1, &
    uptown, MPI_COMM_WORLD, requests, mpierr)

call MPI_recv(take_down2, ntkdn, MPI_DOUBLE_PRECISION, myupper_x1, &
    downtown, MPI_COMM_WORLD, mpistat, mpierr)

call MPI_waitall(nreq, requests, stats, mpierr)

RETURN

END SUBROUTINE share_hibound2D_mpi

!---------------------------------------------------------------------

!---------------------------------------------------------------------

SUBROUTINE share_hibound1D_mpi(give_up1, take_down1)
!----------------------------------------------------------------------------
! | Share boundary information with neighboring upper subdomain. | |
! | Use nonblocking MPI sends and blocking receives. | |
!----------------------------------------------------------------------------

implicit none

real(DP), dimension(:), intent(IN) :: give_up1
real(DP), dimension(:), intent(OUT) :: take_down1
integer(i4b) :: ngvup, ntkdn
integer, parameter :: nreq = 1
integer(i4b) :: requests
integer, dimension(MPI_STATUS_SIZE) :: stats

ngvup = size(give_up1,1)
ntkdn = size(take_down1,1)

call MPI_isend(give_up1, ngvup, MPI_DOUBLE_PRECISION, myupper_x1, &
    uptown, MPI_COMM_WORLD, requests, mpierr)
call MPI_recv(take_down1, ntkdn, MPI_DOUBLE_PRECISION, myupper_x1, &
downtown, MPI_COMM_WORLD, mpistat, mpierr)

call MPI_waitall(nreq, requests, stats, mpierr)

RETURN

END SUBROUTINE share_hibound1D_mpi

!----------------------------------------------------------------------------
!
SUBROUTINE bcast_real_mpi(rval, source)
!
! Broadcast a value of type real from source to the other processors.
!
! implicit none
!
real(DP), intent(INOUT) :: rval
integer(i4b), intent(IN) :: source
!
call MPI_bcast(rval, 1, MPI_DOUBLE_PRECISION, source, MPI_COMM_WORLD, &
mpierr)

RETURN

END SUBROUTINE bcast_real_mpi

!----------------------------------------------------------------------------
!
SUBROUTINE bcast_intarray_mpi(intarray, source)
!
! Broadcast an array of integers from source to the other processors.
!
implicit none
!
integer(i4b), dimension(:), intent(INOUT) :: intarray
integer(i4b), intent(IN) :: source
integer(i4b) :: n
!
n = size(intarray,1)
!
call MPI_bcast(intarray, n, MPI_INTEGER, source, MPI_COMM_WORLD, mpierr)

RETURN

END SUBROUTINE bcast_intarray_mpi

!----------------------------------------------------------------------------
SUBROUTINE bcast_ICheader_mpi(start_tx, amplitude, phase)
!
! Broadcast from master three data from IC file header.
!
implicit none

integer(i4b), intent(INOUT) :: start_tx
real(DP), intent(INOUT) :: amplitude, phase

call MPI_bcast(start_tx, 1, MPI_INTEGER, MASTER, MPI_COMM_WORLD, mpierr)
call MPI_bcast(amplitude, 1, MPI_DOUBLE_PRECISION, MASTER, &
             MPI_COMM_WORLD, mpierr)
call MPI_bcast(phase, 1, MPI_DOUBLE_PRECISION, MASTER, &
              MPI_COMM_WORLD, mpierr)

RETURN
END SUBROUTINE bcast_ICheader_mpi

SUBROUTINE share_ICarray_mpi(IC_array, receiver)
!
! Send array of type acs_data from MASTER to receiver unless they are |
! the same.
!
USE acs_types

implicit none

type(acsdata_type), dimension(:,:,), intent(INOUT) :: IC_array
integer(i4b), intent(IN) :: receiver
integer(i4b) :: n
integer(i4b), parameter :: tag = 10

n = size(IC_array,1) * size(IC_array,2)

if (myid == MASTER) then
    call MPI_send(IC_array, n, mpi_acsdata_type, receiver, tag, &
                MPI_COMM_WORLD, mpierr)
elseif (myid == receiver) then
    call MPI_recv(IC_array, n, mpi_acsdata_type, MASTER, tag, &
                 MPI_COMM_WORLD, mpierr)
end if

! Everyone wait here.
SUBROUTINE share_ICarray_mpi

call MPI_Barrier(MPI_COMM_WORLD, mpierr)

RETURN

END SUBROUTINE share_ICarray_mpi

SUBROUTINE make_acsdata_mpitype(example_in)

USE acs_types

implicit none

type(acsdata_type), intent(IN) :: example_in

type(acsdata_type) :: example ! local copy

integer, parameter :: nvar = 9 ! number of acsdata variables
integer, dimension(nvar) :: block_lengths ! number of elements per block
integer, dimension(nvar) :: displacements
integer, dimension(nvar) :: typelist
integer :: start_address
integer :: address

! Need a local copy of the struct to read the relative addresses.

example = example_in

! Set the number of elements in each block.

block_lengths = (/ 1,1,1,1,1,1,1,1,1 /)

! Calculate other displacements relative to the first.

call MPI_Address(example, start_address, mpierr)

call MPI_Address(example%Xi, address, mpierr)

displacements(1) = address - start_address

call MPI_Address(example%Eta, address, mpierr)

displacements(2) = address - start_address

call MPI_Address(example%grid_x1, address, mpierr)

displacements(3) = address - start_address

call MPI_Address(example%grid_x2, address, mpierr)

displacements(4) = address - start_address

call MPI_Address(example%vars%rho, address, mpierr)

displacements(5) = address - start_address
call MPI_Address(example%vars%u, address, mpierr)
displacements(6) = address - start_address

call MPI_Address(example%vars%v, address, mpierr)
displacements(7) = address - start_address

call MPI_Address(example%vars%t, address, mpierr)
displacements(8) = address - start_address

call MPI_Address(example%vars%p, address, mpierr)
displacements(9) = address - start_address

! Specify the datatypes of each block.
typelist(1:2) = MPI_INTEGER
typelist(3:9) = MPI_DOUBLE_PRECISION

! Build the datatype.
call MPI_Type_struct(nvar, block_lengths, displacements, typelist, &
                    mpi_acsdata_type, mpierr)

! Commit the datatype.
call MPI_Type_commit(mpi_acsdata_type, mpierr)

RETURN

END SUBROUTINE make_acsdata_mpitype

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE dist_ICstat_mpi(statnumber)
! =========================================================================
! | Collect IC_status from all processors and distribute their sum to all. |
! =========================================================================

implicit none

integer(i4b), intent(INOUT) :: statnumber
integer(i4b) :: statsum

call MPI_Allreduce(statnumber, statsum, 1, MPI_INTEGER, MPI_SUM, &
                    MPI_COMM_WORLD, mpierr)

statnumber = statsum

RETURN

END SUBROUTINE dist_ICstat_mpi

!----------------------------------------------------------------------------
!----------------------------------------------------------------------------
SUBROUTINE collect_grid_mpi(grid_x, grid_y, globalgrid_x, globalgrid_y)
Each processor besides the master inserts its piece of the grid into the global array. Then the global arrays from each processor are summed and transmitted to the master processor. Global arrays are \([ni_{\text{global}}, nj_{\text{global}}-1]\) omitting the ghost point across the symmetric center line.

```fortran
implicit none

real(DP), dimension(:,,:), intent(IN) :: grid_x, grid_y
real, dimension(:,,:), intent(OUT) :: globalgrid_x, globalgrid_y
real, dimension(size(globalgrid_x,1),size(globalgrid_x,2)) :: tempgrid
integer :: i_start, i_end, NP

NP = size(globalgrid_x,1) * size(globalgrid_y,2)

! Initialize global grids.
globalgrid_x = 0.0
globalgrid_y = 0.0
tempgrid = 0.0

i_start = i_offset+3
i_end = i_offset + ni

! If I'm not master then position my grids according to offset. Also convert them to lower precision.
if (myid .ne. MASTER) then
    globalgrid_x(i_start:i_end,:) = real(grid_x(3:ni,:),SP)
end if

call MPI_Reduce(globalgrid_x, tempgrid, NP, MPI_Real,_MPI_Sum, 0, MPI_Comm_World, mpierr)
globalgrid_x = tempgrid

! Same thing for grid_y.
tempgrid = 0.0

if (myid .ne. MASTER) then
    globalgrid_y(i_start:i_end,:) = real(grid_y(3:ni,:),SP)
end if

Call MPI_Reduce(globalgrid_y, tempgrid, NP, MPI_Real, _MPI_Sum, 0, MPI_Comm_World, mpierr)
globalgrid_y = tempgrid

RETURN

END SUBROUTINE collect_grid_mpi
```
SUBROUTINE collect_acs_mpi_SP(local_acs, global_acs)
  ! Transfer 3D array from all processors to the master node in the
  ! proper order for output in Plot 3D format. Input array is double
  ! precision while data is transferred and output as single precision.
  !
  implicit none

  real(DP), dimension(ni,nj,1), intent(IN) :: local_acs
  real(SP), dimension(ni_global,nj_global,1), intent(OUT) :: global_acs
  real(SP), dimension(ni_global,nj_global,1) :: temp_acs
  integer :: i_start, i_end, n_elements !, j, k

  n_elements = ni_global * nj_global

  ! Initialize global array.
  global_acs = 0.0
  temp_acs = 0.0

  i_start = i_offset+3
  i_end = i_offset + ni

  if (myid .ne. MASTER) then
    global_acs(i_start:i_end,:,,:) = real(local_acs(3:ni,:,,:),SP)
  end if

  call MPI_Reduce(global_acs, temp_acs, n_elements, MPI_Real, MPI_Sum, &
                   0, MPI_Comm_World, mpierr)

  global_acs = temp_acs

  RETURN
END SUBROUTINE collect_acs_mpi_SP

SUBROUTINE collect_acs_mpi_DP(local_acs, global_acs)
  ! Transfer 3D array from all processors to the master node in the
  ! proper order for output in Plot 3D format. Input array is double
  ! precision while data is transferred and output as single precision.
  !
  implicit none

  real(DP), dimension(ni,nj,1), intent(IN) :: local_acs
real(DP), dimension(ni_global,nj_global,1), intent(OUT) :: global_acs
real(DP), dimension(ni_global,nj_global,1) :: temp_acs
integer :: i_start, i_end, n_elements

n_elements = ni_global * nj_global

! Initialize global array.
global_acs = 0.0
temp_acs = 0.0

i_start = i_offset+3
i_end = i_offset + ni

if (myid .ne. MASTER) then
  global_acs(i_start:i_end,:,,:) = local_acs(3:ni,:,::)
end if

call MPI_Reduce(global_acs, temp_acs, n_elements, MPI_DOUBLE_PRECISION, &
    MPI_Sum, 0, MPI_Comm_World, mpierr)

global_acs = temp_acs

RETURN

END SUBROUTINE collect_acs_mpi_DP

SUBROUTINE sync_mpi()
! Synchronize all processes.
implicit none

call MPI_barrier(MPI_COMM_WORLD, mpierr)

RETURN

END SUBROUTINE sync_mpi

SUBROUTINE cleanup_mpi()
! Take necessary steps to shutdown MPI communications.
implicit none

call MPI_finalize(mpierr)
RETURN

END SUBROUTINE cleanup_mpi

END MODULE mpi_fun
D.4.15  mpivars.fpp

MODULE mpi_vars
!===========================================================================
! This module contains MPI variables. Routines are in mod_mpifun.f90
!===========================================================================
 implicit none

 include 'mpif.h'

 integer, parameter :: MASTER = 0

 integer :: mpierr ! MPI error flag
 integer :: myid  ! process rank
 integer :: nprocs ! number of processors

END MODULE mpi_vars
D.4.16 plot3d.fpp

MODULE plot3d
!===========================================================================
! This module contains subroutines for writing data in Plot3D format.
! Plot3D is readable by TecPlot and is basically Fortran unformatted
! binary output.
!===========================================================================
implicit none

PRIVATE

integer, parameter :: DP = kind(1.0d0)
integer, parameter :: SP = kind(1.0)
integer, parameter :: i4b = Selected_Int_Kind(9)

public :: write_P3Dgrid, write_P3D, size_P3D

CONTAINS

!---------------------------------------------------------------------------
SUBROUTINE write_P3Dgrid(filename, grid_x, grid_y)
! =======================================================================
! | This subroutine writes 2D grid data in Plot3D format. |
! =======================================================================
character(len=*), intent(in) :: filename
real, dimension(:,,:), intent(in) :: grid_x, grid_y
integer, parameter :: iounit = 75
integer :: i, j, i_size, j_size, ierr

i_size = size(grid_x, 1)
j_size = size(grid_x, 2)

! Open file
open (UNIT=iounit, FILE=cat(trim(filename),".p3dgr"), STATUS='REPLACE', &
     ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=ierr)

! Write header
write(iounit) i_size, j_size

! Write grid positions
write(iounit) ((grid_x(i,j), i=1,i_size), j=1,j_size), &
    ((grid_y(i,j), i=1,i_size), j=1,j_size)

! Close file
close (iounit)

RETURN
END SUBROUTINE write_P3Dgrid

!---------------------------------------------------------------------------
! SUBROUTINE write_P3D(filename, outarr)
!                                                                 |
! character(len=*), intent(IN) :: filename                  |
! real(SP), dimension(:,,:), intent(IN) :: outarr          |
! integer, parameter :: iounit = 70                          |
! integer :: i, j, vx, i_size, j_size, vx_max, ierr        |

i_size = size(outarr,1)
j_size = size(outarr,2)
vx_max = size(outarr,3)

! Open file
open (UNIT=iounit, FILE=cat(trim(filename),".p3d"), STATUS='REPLACE', &
ACTION='WRITE', FORM='UNFORMATTED', IOSTAT=ierr)

! Write header
write(iounit) i_size, j_size, vx_max

! Write data
write(iounit) (((outarr(i,j,vx), i=1,i_size), j=1,j_size), vx=1,vx_max)

! Close file
close (UNIT=iounit)

RETURN

END SUBROUTINE write_P3D

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
! SUBROUTINE size_P3D(filename, i_size, j_size, k_size)
!                                                                 |
! implicit none                                                     |

character(len=*), intent(IN) :: filename
integer, intent(OUT) :: i_size, j_size, k_size
integer, parameter :: iounit = 70
integer :: ierr
! Open file
open (UNIT=iounit, FILE=trim(filename), STATUS='OLD', &
     ACTION='READ', FORM='UNFORMATTED', IOSTAT=ierr)

! Read header
read(iounit) i_size, j_size, k_size

! Close file
close (UNIT=iounit)

RETURN

END SUBROUTINE size_P3D

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
FUNCTION cat(str1, str2) RESULT(str3)
!============================================================================
! | This function does a simple concatenation of two strings. It is |
! | included here because the C preprocessor treats Fortran concatenation |
! | operator as a comment and truncates the line at that point. |
!============================================================================
implicit none

character(len=*) :: str1, str2
character(len=len(str1)+len(str2)) :: str3

str3(1:len(str1)) = str1
str3(len(str1)+1:len(str3)) = str2

END FUNCTION cat

!---------------------------------------------------------------------------
END MODULE plot3d
D.4.17 source.fpp

MODULE source
!============================================================================
! This module contains routines and variables for inserting a source in
! the tube model.
!============================================================================

USE global
USE input_vars

implicit none

real(DP) :: minFPW ! "Force piston" width at scale 1
real(DP) :: maxFPW ! maximum "Force piston" width
real(DP) :: FPD ! "Force piston" distance
real(DP) :: FPM ! "Force piston" mask distance
real(DP) :: maxScale
real(DP) :: force_z ! Body force in z-direction per unit mass
integer(i4b) :: transx ! Hybrid source transition time
integer(i4b) :: SPpeakx, FPdelayx
real(DP) :: SPwidth
integer(i4b) :: SARx, phindx, t_indx
integer(i4b) :: period_and_half_tx ! 1.5 periods time steps
real(DP) :: shake, fdrv_Ampl
real(DP) :: gdrv_x2, gdrv_x1
real(DP) :: A_0, Piston_ampl
real(DP) :: FPW ! ** Legacy, to be removed **
real(DP), dimension(:), allocatable :: Pressure_src
real(DP), dimension(:,,:), allocatable :: Temperature_src
real(DP), dimension(:,,:), allocatable :: Density_src
real(DP), dimension(:,,:), allocatable :: Vr_src
real(DP), dimension(:,,:), allocatable :: Velocity_src
real(DP), dimension(:,,:), allocatable :: Velocity_SP
real(DP), dimension(:,,:), allocatable :: Temperature_BL
real(DP), dimension(:,,:), allocatable :: Gaussian_BL
real(DP), dimension(:,,:), allocatable :: Gauss_src1, Gauss_src2
real(DP), dimension(:,,:), allocatable :: iGaussian_BL
real(DP) :: Gauss_width, Gauss_vol1, Gauss_vol2

real(DP) :: wt, Usrc1
real(DP) :: Psource, Psrc_ampl, Pampl
real(DP) :: phi_TA ! phase shift for analytic temperature calc.

CONTAINS
!============================================================================

SUBROUTINE init_source()

USE analyt
USE grid

implicit none

real(DP) :: U_0
integer :: astat
real(DP) :: Therm_BLT
real(DP) :: erfA, erfB, erfX, Gauss_area
integer(i4b) :: i

astat = 0
allocate(Density_src(ni,nj), STAT=astat)
allocate(Temperature_src(ni,nj), STAT=astat)
if (taper > 0.0d0) then
   astat = 0
   allocate(Vr_src(ni,nj), STAT=astat)
end if

select case (fdrv_type)

case ('lin')

   SARx = nint(periodx * SAramp_dur)
   phi_TA = Pi/2.0d0
   Gauss_width = 6d0*max_delta_x1

   astat = 0
   allocate(Temperature_BL(ni,nj), STAT=astat)
   allocate(Gaussian_BL(ni,nj), STAT=astat)

   if (src_pos2 < 0d0 .or. src_pos2 > 100d0) then
      ! Single line source:
      Gaussian_BL = dexp(-0.5d0*(((tube_x1-src_pos1*L/100d0) &
         /Gauss_width)**2d0))
   elseif (src_pos1 < 0d0 .or. src_pos1 > 100d0) then
      Gaussian_BL = dexp(-0.5d0*(((tube_x1-src_pos2*L/100d0) &
         /Gauss_width)**2d0))
   else
      ! Two line sources:
      Gaussian_BL = (dexp(-0.5d0*(((tube_x1-src_pos1*L/100d0) &
         /Gauss_width)**2d0)) + &
         dexp(-0.5d0*(((tube_x1-src_pos2*L/100d0) &
         /Gauss_width)**2d0))))
   end if

   ! Compute area of Gaussian by erf(z) approximation
   erfA = Gauss_width
   erfX = L/(erfA*dsqrt(2d0))
   erfB = 8d0*(Pi-3d0)/(3d0*Pi*(4d0-Pi))
Gauss_area = erfA*2d0*dsqrt(Pi/2d0)*erfX/abs(erfX) * &
dsqrt(1d0-dexp(-erfX**2d0) * &
(4d0/Pi+erfB*erfX**2d0)/(1d0+erfB*erfX**2d0)))

Pampl = S_0 * S_2 / Gauss_area  ! S_2 is fudge factor

case ('shk')
  if (SPpeak < 0.25) SPpeak = 0.25  ! Source pulse no shorter than T/4
  SPpeakx = nint(periodx * SPpeak)
  SPwidth = SPpeakx / 3.0d0
  SARx = nint(periodx * SAramp_dur)

case ('pis')
  if (SPpeak < 0.25) SPpeak = 0.25  ! Source pulse no shorter than T/4
  SPpeakx = nint(periodx * SPpeak)
  SPwidth = SPpeakx / 3.0d0
  SARx = nint(periodx * SAramp_dur)

case ('hyb')
  ! Hybrid source transition time index from shake to "force piston":
  transx = nint(periodx * 0.25)

case ('PRS')  ! Pressure Source
  if (SPpeak < 0.25) SPpeak = 0.25  ! Source pulse no shorter than T/4
  SPpeakx = nint(periodx * SPpeak)
  SPwidth = SPpeakx / 3.0d0
  phi_TA = Pi/2.0d0

  astat = 0
  allocate(Temperature_BL(ni,nj), STAT=astat)
  allocate(Gaussian_BL(ni,nj), STAT=astat)
  allocate(iGaussian_BL(ni,nj), STAT=astat)
  ! Define Gaussian boundary layer array.
  Therm_BLT = sqrt(2.0d0*0.11402d0/(omega0*7.462d0*5192.0d0))
  Gaussian_BL = dexp(-0.5d0*(((L-tube_x1)/(4d0*Therm_BLT))**2d0))
  iGaussian_BL = 1.0d0 - Gaussian_BL

case ('non')  ! No drive -- free vibration

end select

! 1.5 periods pass before acceleration regulator updates from measured pressure
period_and_half_tx = nint(periodx * 1.5)

! Initialize current and last acceleration amplitude to be the same as the
! input value S_1. Target acceleration for the next adjustment is initialized
! to be input value S_2.
A_0 = S_1
A0_last = S_1
A0_tgt = S_2
SUBROUTINE fin_source()

implicit none

integer :: astat

astat = 0
deallocation(Density_src, STAT=astat)
dallocate(Temperature_src, STAT=astat)
if (taper > 0.0d0) then
  astat = 0
  deallocate(Vr_src, STAT=astat)
end if

if (fdrv_type == 'PRS' .or. &
   fdrv_type == 'lac' .or. &
   fdrv_type == 'lma') then
  astat = 0
  deallocate(Temperature_BL, STAT=astat)
dallocate(Gaussian_BL, STAT=astat)
end if

if (fdrv_type == 'lin') then
  astat = 0
  deallocate(Temperature_BL, STAT=astat)
dallocate(Gaussian_BL, STAT=astat)
end if

RETURN

END SUBROUTINE fin_source
USE FDiff_vars
USE analyt
USE grid
implicit none

integer(i4b), intent(IN) :: timex
real(DP), dimension(nj) :: Psrc, Tsrc, Usrc
real(DP), dimension(ni) :: fdrv_dist
real(DP) :: P_amp0, time !, BLthick
real(DP) :: SPg, SPampl, SPscale !, fdrv_Ampl
real(DP) :: SFPD ! Scaled force pulse distance
real(DP) :: SFPW ! Scaled force pulse width
real(DP) :: trans !, shake ** Legacy, to be removed **
real(DP) :: Rwall, BLvisc
real(DP) :: shk_ampl

(t_indx = timex
time = timex*deltat
phindx = modulo(timex,periodx)
wt = omega0 * phindx * deltat

if (end1_TBC == 'src' .or. end2_TBC == 'src') then
  P_amp0 = P_ampl*(1.0d0 - dexp(-time*omega0/(8.0d0*Pi)))
Psrc = P_amp0*dcos(wt - srcphi)
Tsrc = (Psrc-P_amp0*dexp(-(radius-tube_x2(1,:))/delta_K) * &
  dcos(wt - srcphi - (radius-tube_x2(1,:))/delta_K * &
  (1.0d0 - dexp(-time*omega0/(0.5d0*Pi)))) ) / (rho0*Cp)
Pressure_src = Psrc * Jac(1,:)
end if

select case (fdrv_type)

case ('lin') ! line source
  ! Density and temperature line source:
  if (timex < SARx) then
    Psrc_ampl = Pampl * &
    dexp(-0.5d0*(real(timex-SARx)/real(periodx*SAramp_rate))**2d0)
  else
    Psrc_ampl = Pampl
  end if

  Density_src = Psrc_ampl/C02 * Gaussian_BL * omega0*dsin(wt)
  Temperature_src = Psrc_ampl/(rho0*Cp) * Gaussian_BL * omega0*dsin(wt)
  Vr_src = 0.0d0
case ('shk') ! Shake the tube

! Starting pulse Gaussian function: (0,1]
SPg = dexp(-0.5d0*((timex - SPpeakx)/(SPwidth))**2.0d0)

! Drive amplitude:
fdrv_Ampl = S_1 * &
(1.0d0 - dexp(-0.5d0*(real(timex,DP)/real(SARx,DP))**2.0d0))

! Starting pulse acceleration amplitude: (0,S_0]
SPampl = S_0*SPg

fdrv = (fdrv_Ampl + SPampl) * dsin(wt)

! Drive the tube with a uniform body acceleration for 0.5 periods
! of a sine wave.
if (timex < SARx) then
  shk_ampl = S_0
else
  shk_ampl = S_1
end if

gdrv_x1 = shk_ampl * dsin(wt) * dcos(taper)
gdrv_x2 = shk_ampl * dsin(wt) * dsin(taper)


elsewhere
  fdrv_dist = 0.0d0
end where

! Apply to the x2 direction.
  fdrv = spread(fdrv_dist, 2, nj)

case ('hyb')
! Crossfade starting with shake and transitioning to piston.
  if (timex < transx) then
    shake = S_0 * dsin(wt)
    fdrv_dist = shake
  elseif (timex < periodx) then
    trans = dexp(-0.5d0*(((timex-transx)/(0.5d0*transx))**2.0d0)) * &
        (1.0d0 - dexp(-0.5d0*(((L-Xgrid(:,1))/(0.5d0*FPW))**2.0d0)) * &
         dsin(wt))
    shake = S_0 * dsin(wt) * trans
    fdrv_dist = shake + (1.0d0 - trans) * S_1 * &
        (1.0d0 - dexp(-0.5d0*(((L-Xgrid(:,1))/(0.5d0*FPW))**2.0d0)) * &
         dsin(wt))
  elseif (timex < period_and_half_tx) then
    fdrv_dist = S_1 * &
        (1.0d0 - dexp(-0.5d0*(((L-Xgrid(:,1))/(0.5d0*FPW))**2.0d0)) * &
         dsin(wt))
  else
    call A_regulator(timex)
    fdrv_dist = A_0 * &
        (1.0d0 - dexp(-0.5d0*(((L-Xgrid(:,1))/(0.5d0*FPW))**2.0d0)) * &
         dsin(wt))
  end if

  fdrv = spread(fdrv_dist, 2, nj)

case ('PRS')
  Psrc_ampl = P_tgt*(1.0d0-dexp(-0.5d0*(real(timex,DP)/SPwidth)**2d0))
  Psource = Psrc_ampl * dcos(wt-phi_TA)

  if (end2_TBC == 'adi' .and. wall_TBC == 'adi') then
    Temperature_BL = Gaussian_BL * Psource / (rho0*Cp)
  else
    if (taper > 0.0d0) then
      if (timex < periodx) then
        Temperature_BL = Gaussian_BL * Psrc_ampl * (dcos(wt-phi_TA) - &
            (dexp(-(L-tube_x1)/delta_K)* &
             dcos(wt-phi_TA-min(max(0d0,wt-phi_TA),(L-tube_x1)/ &
})
\delta_K) + \text{dexp}(- (taper-tube_x2)*(Rho1+tube_x1)/\delta_K) * \\
cos(wt-phi_TA-min(max(0d0,wt-phi_TA), \\
(taper-tube_x2)*(Rho1+tube_x1)/\delta_K)) - \\
\text{dexp}(-(L-tube_x1 + (taper-tube_x2)*(Rho1+tube_x1))/\delta_K) * \\
dcos(wt-phi_TA-min(max(0d0,wt-phi_TA),(L-tube_x1)/ \\
\delta_K)-min(max(0d0,wt-phi_TA),(taper-tube_x2)*(Rho1+tube_x1)/ \\
\delta_K))))/(\rho0*Cp)

\text{else}
\text{Temperature_BL = Gaussian_BL * Psrc_ampl} * \\
(\cos(wt-phi_TA) - \\
(\text{dexp}(-(L-tube_x1)/\delta_K) * \\
dcos(wt-phi_TA-(L-tube_x1)/\delta_K) + \\
\text{dexp}(-(taper-tube_x2)*(Rho1+tube_x1)/\delta_K) * \\
dcos(wt-phi_TA-(taper-tube_x2)*(Rho1+tube_x1)/\delta_K) - \\
\text{dexp}(-(L-tube_x1+(taper-tube_x2)*(Rho1+tube_x1)) / \\
\delta_K) * dcos(wt-phi_TA-(L-tube_x1 + \\
\delta_K) * dcos(wt-phi_TA-(L-tube_x1)/\delta_K))/\rho0*Cp)

\text{else}
\text{if (timex < periodx) then}
\text{Temperature_BL = Gaussian_BL * Psrc_ampl * (cos(wt-phi_TA) - \\
(\text{dexp}(-(L-tube_x1)/\delta_K)* \\
dcos(wt-phi_TA-L-tube_x1)/\delta_K) + \\
\text{dexp}(-(taper-tube_x2)*(Rho1+tube_x1)/\delta_K) * \\
dcos(wt-phi_TA-(taper-tube_x2)*(Rho1+tube_x1)/\delta_K) - \\
\text{dexp}(-(L-tube_x1+radius - tube_x2)/\delta_K) * \\
dcos(wt-phi_TA-(L-tube_x1)/\delta_K)-min(max(0d0,wt-phi_TA),(radius-tube_x2)/ \\
\delta_K))))/(\rho0*Cp)

\text{else}
\text{Temperature_BL = Gaussian_BL * Psrc_ampl * \cos(wt-phi_TA) - \\
(\text{dexp}(-(L-tube_x1)/\delta_K) * \\
dcos(wt-phi_TA-(L-tube_x1)/\delta_K) + \\
\text{dexp}(-(radius-tube_x2)/\delta_K) * \\
dcos(wt-phi_TA-(radius-tube_x2)/\delta_K) - \\
\text{dexp}(-(L-tube_x1+radius-tube_x2) / \\
\delta_K) * dcos(wt-phi_TA-(L-tube_x1 + \\
\delta_K) * dcos(wt-phi_TA-(L-tube_x1)/\delta_K))/\rho0*Cp)

\text{end if}
\text{end if}
\text{case ('non')}
\text{fdrv = 0.0d0}
\text{end select}
\text{RETURN}
SUBROUTINE A_regulator(timex)
! Adjust source acceleration according to measured pressure amplitude at the monitor point.
USE analyt
implicit none
integer, intent(IN) :: timex
!
! Update A0_last and A0_tgt when PA_mon is updated.
if (timex == PA_monitor_tx) then
  A0_last = A_0
  A0_tgt = A0_last * P_tgt / PA_mon
end if
!
A_0 = A0_last + (A0_tgt - A0_last) * &
  (1.0d0 - dexp(-0.5d0*((timex-PA_monitor_tx)/(0.25d0*transx))**2.0d0))
END SUBROUTINE A_regulator
END MODULE source
D.5 Ancillary Programs

D.5.1 post.fpp

PROGRAM post
!===========================================================================
! Program to do post processing on stream output.
!===========================================================================

USE postproc

implicit none

integer(i4b) :: t

! Read the post-processing input file.
call read_ppinput()

! Merge the data files from multiple processors and write to a single file.
if (merging) then
  do t = txstart, txstop, txstep
    call merge_acs(t)
  end do
end if

! Read combined data file.
! Select data to display.
! Convert data to viewable format.
select case (viewfmt)
  case ("p3d")
    do t = txstart, txstop, txstep
      call bin2p3d(t)
    end do
  case ("ncf")
    print *, "NCF is not an option on this system."
  case default
    print *, "Skipping conversion to viewable format."
end select

END PROGRAM post

D.5.2 postproc.fpp

MODULE postproc
!===========================================================================
! Contains routines for preparing single and multiple processor output
! for viewing.
!===========================================================================


USE global
USE input_vars, ONLY : output

implicit none

character(len=15) :: fileset
integer(i4b) :: txdigits, txstart, txstop, txstep
logical :: merging
character(len=3) :: viewfmt
character(len=3) :: acsvar

CONTAINS

SUBROUTINE read_ppinput()

USE IO_tools

implicit none

character(len=7), parameter :: fname = 'ppinput'
integer, parameter :: IOU = 110
integer, parameter :: tabstop = 12
integer(i4b) :: istat
character(len=5) :: timexfmt

open(UNIT=IOU, FILE=fname, STATUS='OLD', ACTION='READ', &
    FORM='FORMATTED', IOSTAT=istat)

call skipline(IOU, 1)
call skiptab(IOU, tabstop)
read(IOU, "(l2)", IOSTAT=istat) merging
call skiptab(IOU, tabstop)
read(IOU, "(a)", IOSTAT=istat) viewfmt
call skiptab(IOU, tabstop)
read(IOU, "(a)", IOSTAT=istat) acsvar

call skipline(IOU, 2)
call skiptab(IOU, tabstop)  !"per" for filename, "reg" for fileset
read(IOU, "(a)", IOSTAT=istat) output
call skiptab(IOU, tabstop)
read(IOU, "(a)", IOSTAT=istat) fileset

call skiptab(IOU, tabstop)
read(IOU, "(i2)", IOSTAT=istat) txdigits
call skiptab(IOU, tabstop)
read(IOU, "(i2)", IOSTAT=istat) prdigs
write(timexfmt, "(a,i2,a)", IOSTAT=istat) "(i", txdigits, ")"
call skiptab(IOU, tabstop)
read(IOU,timexfmt, IOSTAT=istat) txstart
call skiptab(IOU, tabstop)
read(IOU,timexfmt, IOSTAT=istat) txstop
call skiptab(IOU, tabstop)
read(IOU,timexfmt, IOSTAT=istat) txstep
close(UNIT=IOU)

txdigs = txdigits
call report_ppinput()
RETURN

END SUBROUTINE read_ppinput

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE report_ppinput()
! | Report what was just read to stout. |
! | ------------------------------------------------------------------------ |

implicit none

print *, "--- Action ---"
print *, "merging = ", merging
print *, "format = ", viewfmt
print *, "variables = ", acsvar
print *, ""
print *, "--- File Choice ---"
print *, "output = ", output
print *, "setname = ", fileset
print *, "tx digits = ", txdigits
print *, "pr digits = ", prdigs
print *, "start ndx = ", txstart
print *, "stop ndx = ", txstop
print *, "step ndx = ", txstep
RETURN

END SUBROUTINE report_ppinput

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE merge_acs(timex)
This routine reads data from multiple processors (with processor number .nn suffix) and writes the merged data to a single file with no suffix. Assume that domain decomposition is over only x1-dimension (xi) and that they are in order.

USE IO_tools
USE acs_types

implicit none

integer(i4b), intent(IN) :: timex

integer(i4b), parameter :: MFID = 200
integer(i4b), parameter :: SFID = 210
integer(i4b) :: nfiles
character(len=16+txdigits+prdigs+1) :: sfname
character(len=16+txdigits) :: mfname

! source file header:
integer(i4b), dimension(2) :: sf_array_size, tmp_array_size, data_size
integer(i4b), dimension(4) :: sf_comp_range, tmp_comp_range, data_range
integer(i4b) :: sf_timex, tmp_timex
real(DP) :: sf_time, sf_S_0, sf_phi, tmp_time, tmp_S_0, tmp_phi
character(len=txdigits) :: txstr
character(len=prdigs) :: numstr
integer(i4b) :: i, j, N, istat, n_i, n_j
integer(i4b) :: minXi, maxXi, minEta, maxEta
type(acsdata_type), dimension(:,), allocatable :: data_array
real(DP) :: time

! Count how many files there are at the specified timestep.
nfiles = 0
nfiles = count_files(trim(fileset), timex)
print *, "There are ", nfiles, "files to merge."
if (nfiles == 0) RETURN

! Figure out header information for the merge file.
! read header of the first file.
if (output == 'reg') then
    sfname = filenamer(trim(fileset), timex, 0)
else
    numstr = int2str(0, prdigs)
sfname = trim(fileset) // "." // numstr
end if
open(UNIT=SFID, FILE=trim(sfname), STATUS='OLD', ACTION='READ', &
FORM='UNFORMATTED', IOSTAT=istat)
print *, "File name: ", trim(sfname)
read(SFID, IOSTAT=istat) sf_array_size
print *, "Array size = ", sf_array_size
read(SFID, IOSTAT=istat) sf_comp_range
print *, "Domain range = ", sf_comp_range
read(SFID, IOSTAT=istat) sf_timex
print *, "timex = ", sf_timex
read(SFID, IOSTAT=istat) sf_time
print *, "time = ", sf_time
read(SFID, IOSTAT=istat) sf_S_0
print *, "S_0 = ", sf_S_0
read(SFID, IOSTAT=istat) sf_phi
print *, "phi = ", sf_phi

close(SFID)

n_i = sf_array_size(1) - 1 ! omit one ghost point
n_j = sf_array_size(2)
minXi = sf_comp_range(1)
minEta = sf_comp_range(2)
maxXi = sf_comp_range(3)
maxEta = sf_comp_range(4)

! loop over remaining files.
do N = 2, nfiles
  ! read header.
  if (output == 'reg') then
    sfname = filenamer(trim(fileset), timex, N-1)
  else
    numstr = int2str(N-1, prdigs)
    sfname = trim(fileset) // "." // numstr
  end if
  open(UNIT=SFID, FILE=trim(sfname), STATUS='OLD', ACTION='READ', &
       FORM='UNFORMATTED', IOSTAT=istat)
  print *, "File name: ", trim(sfname)
  read(SFID, IOSTAT=istat) tmp_array_size
  print *, " Array size = ", sf_array_size
  if (tmp_array_size(2) .ne. sf_array_size(2)) print *, &
     "Warning: Source files inconsistent in X2-resolution."
  read(SFID, IOSTAT=istat) tmp_comp_range
  print *, " Domain range = ", tmp_comp_range
  read(SFID, IOSTAT=istat) tmp_timex
  print *, " timex = ", tmp_timex
  if (tmp_timex .ne. sf_timex) print *, &
     "Warning: Source files inconsistent in time index."
  read(SFID, IOSTAT=istat) tmp_time
  print *, " time = ", tmp_time
  read(SFID, IOSTAT=istat) tmp_S_0
  print *, " S_0 = ", tmp_S_0
  read(SFID, IOSTAT=istat) tmp_phi
  print *, " phi = ", tmp_phi
  close(SFID)
end do

! Update array size and computational range for header of merge file.
n_i = n_i + tmp_array_size(1) - 2 ! omit ghost points
if (tmp_comp_range(1) < minXi) minXi = tmp_comp_range(1)
if (tmp_comp_range(2) < minEta) minEta = tmp_comp_range(2)
if (tmp_comp_range(3) > maxXi) maxXi = tmp_comp_range(3)
if (tmp_comp_range(4) > maxEta) maxEta = tmp_comp_range(4)
end do

n_i = n_i + 1 ! Add last ghost point.

! Allocate data array.
istat = 0
allocate(data_array(n_j), STAT=istat)
if (istat .ne. 0) print *, "Problem allocating data_array."

! Make a filename with no processor number suffix.
if (output == 'per') then
  mfname = trim(fileset)
else
  txstr = int2str(timex,txdigs)
  mfname = trim(fileset) // txstr
end if

! Open the destination "merge" file.
open(UNIT=MFID, FILE=trim(mfname), STATUS='REPLACE', ACTION='WRITE', &
FORM='UNFORMATTED', IOSTAT=istat)

! Write the header.
data_size = (/ n_i, n_j /)
print *, "Data size = ", data_size
data_range = (/ minXi, minEta, maxXi, maxEta /)
print *, "Data range = ", data_range

write(MFID) data_size
write(MFID) data_range
write(MFID) sf_timex
write(MFID) sf_time
write(MFID) sf_S_0
write(MFID) sf_phi

! Loop over source data files. Assume domain decomposition only in the i- 
! direction and that domains overlap one bin on each end.
N = 1
! open first source file.
if (output == 'reg') then
  sfname = filenamer(trim(fileset), timex, N-1)
else
  numstr = int2str(N-1,prdigs)
  sfname = trim(fileset) // "." // numstr
end if
open(UNIT=SFID, FILE=trim(sfname), STATUS='OLD', ACTION='READ', &
FORM='UNFORMATTED', IOSTAT=istat)

! read past header info.
read(SFID, IOSTAT=istat) tmp_array_size
read(SFID, IOSTAT=istat) tmp_comp_range
read(SFID, IOSTAT=istat) tmp_timex
read(SFID, IOSTAT=istat) tmp_time
read(SFID, IOSTAT=istat) tmp_S_0
read(SFID, IOSTAT=istat) tmp_phi

! Loop over i size omitting the last row.
do i = 1, tmp_array_size(1) - 1
! read record from source file.
read(SFID) data_array
! write record to merge file.
write(MFID) data_array
end do
! close source file.
close(SFID)
do N = 2, nfiles-1
! open new source file.
if (output == 'reg') then
  sfname = filenamer(trim(fileset), timex, N-1)
else
  numstr = int2str(N-1,prdigs)
sfname = trim(fileset) // "." // numstr
end if
open(UNIT=SFID, FILE=trim(sfname), STATUS='OLD', ACTION='READ', &
FORM='UNFORMATTED', IOSTAT=istat)
! read past header info.
read(SFID, IOSTAT=istat) tmp_array_size
read(SFID, IOSTAT=istat) tmp_comp_range
read(SFID, IOSTAT=istat) tmp_timex
read(SFID, IOSTAT=istat) tmp_time
read(SFID, IOSTAT=istat) tmp_S_0
read(SFID, IOSTAT=istat) tmp_phi
! read and skip first record from source file.
read(SFID) data_array
! Loop over i size omitting the first and last rows.
do i = 2, tmp_array_size(1) - 1
! read record from source file.
read(SFID) data_array
! write record to merge file.
write(MFID) data_array
end do
! close source file.
close(SFID)
! End loop
end do

N = nfiles
! open last source file.
if (output == 'reg') then
  sfname = filenamer(trim(fileset), timex, N-1)
else
    numstr = int2str(N-1, prdigs)
    sfname = trim(fileset) // "." // numstr
end if
open(UNIT=SFID, FILE=trim(sfname), STATUS='OLD', ACTION='READ', &
     FORM='UNFORMATTED', IOSTAT=istat)
read past header info.
    read(SFID, IOSTAT=istat) tmp_array_size
    read(SFID, IOSTAT=istat) tmp_comp_range
    read(SFID, IOSTAT=istat) tmp_timex
    read(SFID, IOSTAT=istat) tmp_time
    read(SFID, IOSTAT=istat) tmp_S_0
    read(SFID, IOSTAT=istat) tmp_phi
! read and skip first record from source file.
    read(SFID) data_array
! Loop over i size omitting the first row.
    do i = 2, tmp_array_size(1)
        ! read record from source file.
        read(SFID) data_array
        ! write record to merge file.
        write(MFID) data_array
    end do
! close source file.
    close(SFID)
! Close the merge file.
    close(MFID)

! Deallocate data array.
    istat = 0
    deallocate(data_array, STAT=istat)
    if (istat .ne. 0) print *, "Problem deallocating data_array."
RETURN
END SUBROUTINE merge_acs

!---------------------------------------------------------------------------
!---------------------------------------------------------------------------
SUBROUTINE bin2p3d(timex)
! =========================================================================
! | Convert Fortran binary acs data file to Plot3D format. |
! =========================================================================
USE IO_tools
USE acs_types
USE plot3d
implicit none

integer(i4b), intent(IN) :: timex
character(len=txdigits) :: txstr
real(DP) :: time

! Type(acsdata_type), dimension(:,,:), allocatable :: acsarray
real, dimension(:,,:), allocatable :: X1grid, X2grid
real, dimension(:,,:), allocatable :: p3darray
character(len=3+txdigits) :: fname
integer(i4b), dimension(2) :: acssize
integer(i4b) :: istat
logical :: ex

! Name the file to be read.
  txstr = int2str(timex, txdigs)
  fname = trim(fileset) // txstr

! See if it exists.
  ex = .false.
  inquire(FILE=fname, EXIST=ex)
  if (.not. ex) RETURN

! Find size of the data array.
  acssize = read_filesize(fname)

! Allocate arrays.
  istat = 0
  allocate(acsarray(acssize(1),acssize(2)), STAT=istat)
  if (istat > 0) then
    print *, "Problem allocating acsarray for bin2p3d()."
    STOP
  end if
  istat = 0
  allocate(X1grid(acssize(1),acssize(2)-1), STAT=istat)
  if (istat > 0) then
    print *, "Problem allocating X1grid for bin2p3d()."
    STOP
  end if
  istat = 0
  allocate(X2grid(acssize(1),acssize(2)-1), STAT=istat)
  if (istat > 0) then
    print *, "Problem allocating X2grid for bin2p3d()."
    STOP
  end if
  istat = 0
  allocate(p3darray(acssize(1),acssize(2)-1,5), STAT=istat)
  if (istat > 0) then
    print *, "Problem allocating p3darray for bin2p3d()."
    STOP
  end if

! Read binary file.
  call read_acsdata(fname, time, acsarray)
! Assign data arrays.
X1grid = acsarray(:,2:acssize(2))%grid_x1
X2grid = acsarray(:,2:acssize(2))%grid_x2
p3darray(:,1) = acsarray(:,2:acssize(2))%vars%rho
p3darray(:,2) = acsarray(:,2:acssize(2))%vars%u
p3darray(:,3) = acsarray(:,2:acssize(2))%vars%v
p3darray(:,4) = acsarray(:,2:acssize(2))%vars%t
p3darray(:,5) = acsarray(:,2:acssize(2))%vars%p

if (timex == txstart) then
! Write the plot3d grid file.
  print *, "Creating Plot3D grid file ", fname // ".p3dgr"
  call write_P3Dgrid(fname, real(X1grid,SP),real(X2grid,SP))
end if

! Write the plot3d data file.
  print *, "Creating Plot3D ", fname // ".p3d"
  call write_P3D(fname, real(p3darray, SP))

RETURN
END SUBROUTINE bin2p3d

!---------------------------------------------------------------------------
END MODULE postproc

D.5.3 meanACS.fpp

PROGRAM meanACS

USE plot3d
implicit none

! Precision
integer, parameter :: DP = kind(1.0d0)
integer, parameter :: SP = kind(1.0)
integer, parameter :: i4b = Selected_Int_Kind(9)

integer(i4b) :: ni
integer(i4b) :: nj
integer(i4b) :: nSACSvar

real(DP), dimension(:,:,), allocatable :: sum_arr
real(DP), dimension(:,:,), allocatable :: acs_arr
real(DP), dimension(:,:), allocatable :: str_arr

integer(i4b), parameter :: nSTRarr = 2
character(len=9) :: infilename = 'stream.ls'
character(len=20) :: meanfout, streamfout, nextfilename
integer(i4b) :: INU = 10
integer(i4b) :: istat, astat, filecount

real(DP) :: rho0 = 7.462d0

! Open the input file with the list of file names to merge.
! Omit the '.p3d' from the filename as it will be added by write_P3D.
open (UNIT=INU, FILE=infilename, STATUS='OLD', ACCESS='SEQUENTIAL', &
     IOSTAT=istat)

! Read the first line for the output filename.
read (INU,"(a)",IOSTAT=istat) meanfout

! Initialize file counter.
filecount = 0

! Loop over the remaining filenames until end of the input file.
filelist: do

! Read next filename from input.
read (INU,"(a)",IOSTAT=istat) nextfilename

! If there’s not one, exit the loop.
if (istat < 0) exit

! Update the file counter.
filecount = filecount + 1

! Get the array size from the first file.
if (filecount == 1) then
    call size_P3D(nextfilename, ni, nj, nSACSvar)
! Allocate arrays.
    astat = 0
    allocate(acs_arr(ni,nj,nSACSvar), STAT=astat)
    allocate(sum_arr(ni,nj,nSACSvar), STAT=astat)
    allocate(str_arr(ni,nj,nSTRarr), STAT=astat)
    if (astat > 0) print *, "Problem allocating arrays."
! Initialize the sum array to zero.
    sum_arr = 0.0d0
end if

! Read binary double-precision format file.
call read_array(nextfilename, acs_arr)

! Add the ACS values to the sum array.
sum_arr = sum_arr + acs_arr

! End loop.
end do filelist

! Close the input file.
close (UNIT=INU)
!! Divide the sum array by the number of files.
acs_arr = sum_arr / filecount
!!=======================================================================
!! The components of sumacs are
!! <rho'>, <u>, <v>, <T'>, <p'>, <rho'*u>, <rho'*v>
!!=======================================================================
!! Write the result in Plot3D format.
call write_P3D(cat(trim(meanfout),"mn"), real(acs_arr,SP))
!! Calculate streaming arrays. See LB4 p.83. Yano:
str_arr(:,:,1) = (rho0*acs_arr(:,:,2) + acs_arr(:,:,6)) / rho0
str_arr(:,:,2) = (rho0*acs_arr(:,:,3) + acs_arr(:,:,7)) / rho0
call write_P3D(cat(trim(meanfout),"stY"), real(str_arr,SP))
!! Calculate streaming arrays. See LB4 p.83. Boluriaan <V>E2:
!! This simplifies to <u> and <v>. ** Check this calculation **
str_arr(:,:,1) = ((rho0*acs_arr(:,:,2) + acs_arr(:,:,6)) - &
acs_arr(:,:,6)) / rho0
str_arr(:,:,2) = ((rho0*acs_arr(:,:,3) + acs_arr(:,:,7)) - &
acs_arr(:,:,7)) / rho0
call write_P3D(cat(trim(meanfout),"stE2"), real(str_arr,SP))
!! Calculate streaming arrays. See LB4 p.83. Boluriaan <V>l:
str_arr(:,:,1) = acs_arr(:,:,2) + acs_arr(:,:,6) / (rho0+acs_arr(:,:,1))
str_arr(:,:,2) = acs_arr(:,:,3) + acs_arr(:,:,7) / (rho0+acs_arr(:,:,1))
call write_P3D(cat(trim(meanfout),"stL"), real(str_arr,SP))
!! Calculate streaming arrays. See LB4 p.83. Atkas and Farouk:
str_arr(:,:,1) = (rho0*acs_arr(:,:,2) + acs_arr(:,:,6)) / &
(rho0+acs_arr(:,:,1))
str_arr(:,:,2) = (rho0*acs_arr(:,:,3) + acs_arr(:,:,7)) / &
(rho0+acs_arr(:,:,1))
call write_P3D(cat(trim(meanfout),"stAF"), real(str_arr,SP))
!! Clean up arrays.
astat = 0
deallocate(acs_arr, STAT=astat)
deallocate(sum_arr, STAT=astat)
deallocate(str_arr, STAT=astat)
if (astat > 0) print *, "Problem deallocating arrays."
CONTAINS
SUBROUTINE read_array(filename, inarr)
!
!  Read in an array of multivariable 2D data in Fortran binary double precision.
!

implicit none

character(len=*) , intent(IN) :: filename
real(DP), dimension(:,:,:), intent(OUT) :: inarr
integer(i4b), parameter :: iounit = 70
integer(i4b) :: ierr, i, j, k, i_size, j_size, k_size

! Open file
open (UNIT=iounit, FILE=trim(filename), STATUS='OLD', &
      ACTION='READ', FORM='UNFORMATTED', IOSTAT=ierr)

! Read header
read(iounit) i_size, j_size, k_size

! Read data
read(iounit) (((inarr(i,j,k), i=1,i_size), j=1,j_size), k=1,k_size)

! Close file
close (UNIT=iounit)

RETURN
END SUBROUTINE read_array

FUNCTION cat(str1, str2) RESULT(str3)
!
!  This function does a simple concatenation of two strings. It is included here because the C preprocessor treats Fortran concatenation operator as a comment and truncates the line at that point.
!
implicit none

character(len=*) :: str1, str2
character(len=len(str1)+len(str2)) :: str3

str3(1:len(str1)) = str1
str3(len(str1)+1:len(str3)) = str2

END FUNCTION cat

END PROGRAM meanACS
Bibliography


Vita

Brian C. Tuttle

Brian Tuttle began his studies of sound at Ball State University with a major in music theory and composition and a minor in applied physics. He spent many hours in the percussion studio and many more hours in the electronic music composition and recording studios. These efforts resulted in two solo percussion recitals and a recital of his compositions including his senior Honors project, “Computer in C” for log drum and interactive electronics.

He received a Bachelor of Music degree in 1991 and proceeded to work as a mastering engineer at Cinram, Inc., an audio production and duplication facility in Richmond, Indiana. While in Richmond he attended Indiana University East to learn the prerequisites for engineering mathematics before applying to the Graduate Program in Acoustics at Penn State.

As a masters student in acoustics, Brian worked as system administrator of the department’s computer facilities. He wrote a paper on the acoustics of mallet percussion instruments entitled The Effects of a Resonator Tube on the Timbre and Directivity of Sound Radiated from a Vibraphone Bar, and in 1999 he completed a Master of Science degree with the thesis topic Wavelet Parameters for Speech Synthesis.

Once accepted into the doctoral program, he served as president of both the Penn State chapter of the Audio Engineering Society and the Central Pennsylvania Chapter of the Acoustical Society of America. He gained experience with parallel computer programming by maintaining a Beowulf style cluster for research in acoustic streaming. This experience helped him to support himself during the completion of this research through his work as a scientific computing consultant for the Geosciences Department, where he has tackled such problems as modeling ice sheets, predicting sea-level rise, and optimizing global economic/climate policies.

Throughout his graduate career Brian has been able to make music with various groups in the central Pennsylvania region. He hopes to continue to find opportunities to practice science, engineering, and the arts well into the future.