

The Numerical Solution to the Differential Equations in the Viscous Critical Stress Model

John Weatherwax*

1 Introduction

This document will be a detailed discussion of the numerical code and algorithm used in the soon to be published paper [6]. We begin with a review of the mathematics and numerics described in that paper and move into very specific details regarding the FORTRAN coded implementation. The purpose of this document is to provide a method for others to use and understand the numerical code provided at the level of being able to provide extensions and updates if desired. Recently there has been a wealth of data released on high pressure shock waves in solids [1] (over 490 pages) and it is the author's hope that perhaps this code can provide some (however small) insights into this highly complicated behaviour.

We begin with the governing mathematical equations, present a discription of the numerical algorithm choosen to solve these equations, and finally present a very detailed discription of the data structures and FORTRAN implementation.

2 The Governing Viscous Equations

We solve the following system of partial differential equations

$$\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial x}(\rho u) = 0, \quad \frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho u^2 + p) = \frac{\partial}{\partial x} \left(\mu \frac{\partial}{\partial x} u \right), \quad (1)$$

where the viscosity coefficient μ is a measure of the amount of dissipation associated with phase transitions and shocks. (As mentioned above two viscosity values should be used: one ahead of the precursor shock, and a second one behind it.) Equivalently, the Lagrangian equations

*wax@alum.mit.edu

become (with ξ the Lagrangian mass coordinate).

$$\frac{\partial}{\partial t}v - \frac{\partial}{\partial \xi}u = 0, \quad \frac{\partial}{\partial t}u + \frac{\partial}{\partial \xi}P = \frac{\partial}{\partial \xi} \left(\frac{\mu}{v} \frac{\partial}{\partial \xi}u \right). \quad (2)$$

To study the effect that the added viscous term has on the free surface profiles, we solved numerically the Lagrangian equations (2); In the next section we present the numerical algorithm used in the solution of these equations.

3 Viscous numerics

A hypothetical representation of the solution at a time after impact, but before the precursor shock has collided with the free surface, is shown in Figure 1. There one can see the precursor shock broadened by the dissipation: the shock is a compression wave where the competing effects of nonlinearity and diffusion balance each other. It is clear from this figure that the phase transformation front divides the material into two domains of interest: *domain A*, containing the high-density phase, behind the transformation front, and *domain B*, containing the low-density phase, ahead of the transformation front. The natural division of the flow into separate domains,

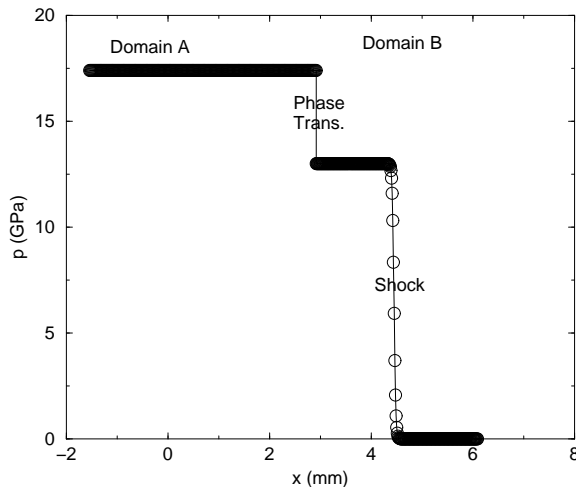


Figure 1: Hypothetical diffusive pressure wave profile at a time shortly after impact. The marks in the horizontal axis measure distance, in millimeters, from the point of impact. The vertical marks measure pressure in Giga Pascals. Domain A (respectively B) is defined as the region to the left (respectively, to the right) of the phase transformation. Here we see the right-facing phase transformation, as a discontinuity, and the right-facing shock, slightly smeared by the viscosity, both propagating toward the free surface—located at $x \approx 6 \text{ mm}$.

by a moving phase transformation discontinuity, motivates the structure of the algorithm presented below—which we describe for rather general configurations, including, possibly, more than one discontinuity and more than two domains. Specifically, we derive discrete numerical

equations for M domains, separated by $M + 1$ interfaces—including the left end of the first domain and the right end of the last domain (which coincide with the left- and right-ends of the sample, respectively), see Figure 2. It is fundamental to the vCS model (and hence the numerical algorithm that implements it) that *in each domain there is only a single phase and a single equation of state*. In particular, in each domain the equations (2) have a unique meaning, and standard finite difference discretizations can be used on them, as described in what follows. Our integration problem will thus reduce to adequately solving the equations across the phase transition fronts, where the equation of state has a discontinuity and where, the significance of the *spatial derivatives* in equations (2) must be specified. Recognizing this fact, our numerical approach then relies on use of the *method of lines* to perform the time integration once a suitable spatial discretization is defined. A specification of the discretization to be used for all of the required spatial derivatives is presented in the following subsections.

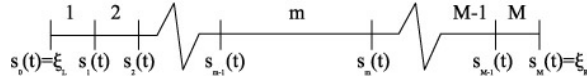


Figure 2: Several domains separated by moving discontinuities, including the left end of the first domain and the right end of the last domain (which coincide with the left and the right end-points of the sample, respectively). Each discontinuity is a phase transition, and in each domain there is a single phase. The numbers above each domain represent the domain index, running from 1 to M .

At this point it is useful to outline the rest of Section 3, which is the most technical part of this paper. In subsection 3.1 we present a transformation of the independent variable ξ that maps the individual constant phase domains (with endpoints $s_{m-1}(t)$ and $s_m(t)$) to a *fixed* stationary grid (with endpoints 0 and 1). In subsection 3.1 we then present the spatial discretization of (2) we use in the *interior* of each domain. In subsection 3.3 we present the discrete implementations we used *across* domain boundaries for the derivatives in equation (2). These implementations can be classified into several subcases depending on the type of the domain boundary under consideration; e.g. forward/backwards transformation front, contact discontinuity etc. With all spatial derivatives discretized, the numerical description can be completed by an appropriate temporal discretization; since the temporal discretization is rather standard we defer the presentation of its details to Subsection 3.2. Numerical results produced by means of the resulting algorithm are then presented in Section ??.

3.1 Fixed-domain transformation and finite-difference spatial discretization

Consider a section of the real line from ξ_L to ξ_R , containing $M + 1$ interfaces and M domains, as in Figure 2. Assume that, as a function of time t , the locations of the interfaces are given by the smooth functions $\xi = s_i(t)$, where $0 \leq i \leq M$, $\xi_L = s_0(t)$ and $\xi_R = s_M(t)$ correspond to the left and right ends of the sample, respectively. Recalling that we work in Lagrangian coordinates, in what follows each one of the functions $\xi = s_i(t)$ are identified with the mass to the left of the corresponding interface: e.g. $\xi_L = 0$, and $\mathcal{M}_i = \dot{s}_i(t)$ is the mass flux across the interface. A similar formulation is possible in Eulerian coordinates, where the values $s_i(t)$ and $\dot{s}_i(t)$ give the location in space and velocity of the i -th interface.

In the m -th ($m = 1, 2, \dots, M$) domain $s_{m-1}(t) \leq \xi \leq s_m(t)$ we define the coordinate transformation in the m -th domain $\xi \rightarrow \xi_m$ by

$$\xi_m = \frac{\xi - s_{m-1}(t)}{s_m(t) - s_{m-1}(t)}, \quad \text{so that} \quad \begin{cases} \xi = s_{m-1} & \iff \xi_m = 0. \\ \xi = s_m & \iff \xi_m = 1. \end{cases} \quad (3)$$

Thus, as is commonly done in connection with calculations involving free boundaries [3], the m -th domain is transformed into the fixed interval $[0, 1]$ and the unknown functions $s_m(t)$ determine the positions of the corresponding interfaces. Within each domain we can re-express equations (2) in terms of the local variable ξ_m using equation (3); dropping the subscript m on ξ_m and the time dependence in $s_m(t)$, the equations for the unknowns in domain m thus become

$$\frac{\partial}{\partial t} v - \frac{1}{s_m - s_{m-1}} \left[((1 - \xi) \dot{s}_{m-1} + \xi \dot{s}_m) \frac{\partial}{\partial \xi} v + \frac{\partial}{\partial \xi} u \right] = 0, \quad (4)$$

and

$$\frac{\partial}{\partial t} u - \frac{(1 - \xi) \dot{s}_{m-1} + \xi \dot{s}_m}{s_m - s_{m-1}} \frac{\partial}{\partial \xi} u + \frac{1}{s_m - s_{m-1}} \frac{\partial}{\partial \xi} P = \frac{\mu}{(s_m - s_{m-1})^2} \frac{\partial}{\partial \xi} \left(\frac{1}{v} \frac{\partial u}{\partial \xi} \right). \quad (5)$$

In detail, we break up each domain $0 \leq \xi \leq 1$ (now ξ represents our transformed coordinates) scaled into a uniform grid with N nodes including the end points:

$$\xi_i = (i - 1) \Delta \xi, \quad \text{where} \quad \Delta \xi = \frac{1}{N - 1} \quad \text{and} \quad 1 \leq i \leq N; \quad (6)$$

we call the interval $\xi_i \leq \xi \leq \xi_{i+1}$ the i -th cell—with center $\xi_{i+1/2} = \frac{1}{2} (\xi_i + \xi_{i+1})$. Then, for the specific volume v (and density $\rho = 1/v$) we use a node centered approach, in which the numerical scheme evolves the values $v(\xi_i, t)$ of the specific volume at the nodes. For the velocity, on the other hand, we use a cell centered approach: the numerical scheme evolves the values $u(\xi_{i+1/2}, t)$ of the velocity at the cell centers. See figure 4 for an example when $N = 4$. It is now convenient to introduce *notation to indicate whether a numerical variable is defined on nodes or cells*: when referring to a cell centered variable, for example u , we will use a bar, as in \bar{u} ; a node centered variable, on the other hand, will receive no special indication. Thus the notation v_ξ applies to the node centered value of the ξ derivative of the specific volume, while \bar{v}_ξ means the cell centered value of the same quantity. With these definitions we are now in a position to write down the spatial discretization of the dissipative equations (4) and (5) using the staggered grid technique. Since the method of lines applied to the conservation of mass equation in (4) involves updating the node centered variable v , we need node-centered discrete versions for the

spatial derivatives in the equation. We thus use centered differences to approximate the spatial derivatives appearing in equation (4):

$$\frac{\partial}{\partial \xi} v(\xi_i, t) \approx \frac{v(\xi_{i+1}, t) - v(\xi_{i-1}, t)}{2\Delta\xi} \quad \text{for } i = 2, 3, \dots, N-1 \quad (7)$$

and

$$\frac{\partial}{\partial \xi} u(\xi_i, t) \approx \frac{\bar{u}(\xi_{i+\frac{1}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{\Delta\xi} \quad \text{for } i = 1, 2, \dots, N \quad (8)$$

Note that the expressions on the right hand sides of these equations involve the natural domain of definition of the variables involved: node centered specific volumes and cell centered velocities.

When used in (4) (over the common domain of application $i = 2, \dots, N-1$), the approximations (7), and (8), lead to the semi-discrete conservation-of-mass system

$$\begin{aligned} \frac{\partial}{\partial t} v(\xi_i, t) &= \left(\frac{(1 - \xi_i) \dot{s}_{m-1} + \xi_i \dot{s}_m}{s_m - s_{m-1}} \right) \frac{v(\xi_{i+1}, t) - v(\xi_{i-1}, t)}{2\Delta\xi} \\ &+ \left(\frac{1}{s_m - s_{m-1}} \right) \frac{\bar{u}(\xi_{i+\frac{1}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{\Delta\xi} \end{aligned} \quad (9)$$

$$\text{for } i = 2, 3, \dots, N-1,$$

which is second-order accurate in $\Delta\xi$. A similar procedure can be applied to the conservation of momentum equation (5); this equation requires time-updates of the velocity, a cell centered variable, so that cell centered approximations for the various ξ -derivatives involved are needed. The exact discretizations used to do this are no more technically complicated than the ones used above in the conservation of mass equation, but they do contain more terms—thus a presentation of the full semi-discrete version of the conservation of momentum equation is presented next.

3.1.1 Discretization of the momentum equation

In this appendix we present the semi-discrete equations for the conservation of momentum, used in the numerical solution of the model for shock induced phase transitions, as modified by the addition of dissipation—see Section 3.1. We use the following cell centered approximations for the first order space derivatives that appear in equation (5), which follow from using centered differences:

$$\frac{\partial}{\partial \xi} \bar{u}(\xi_{i+\frac{1}{2}}) \approx \frac{\bar{u}(\xi_{i+\frac{3}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{2\Delta\xi} \quad \text{for } i = 1, 2, \dots, N-1 \quad (10)$$

and

$$\frac{\partial}{\partial \xi} \bar{p}(\xi_{i+\frac{1}{2}}, t) \approx \frac{p(\xi_{i+1}, t) - p(\xi_i, t)}{\Delta \xi} \quad \text{for } i = 1, 2, \dots, N-1. \quad (11)$$

Similarly, for the viscous term we use the spatial discretization:

$$\overline{\left(\frac{1}{v} \frac{\partial u}{\partial \xi} \right)}(\xi_{i+\frac{1}{2}}) \approx \frac{1}{\Delta \xi^2} \left[\frac{\bar{u}(\xi_{i+\frac{3}{2}}, t) - \bar{u}(\xi_{i+\frac{1}{2}}, t)}{v(\xi_{i+1}, t)} - \frac{\bar{u}(\xi_{i+\frac{1}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{v(\xi_i, t)} \right], \quad (12)$$

which holds when $i = 1, 2, \dots, N-1$. In all cases the terms on the right hand sides of the expressions are defined in their natural domains (node centered densities and cell centered velocities). With these approximations, the semi-discrete equation for the conservation of momentum (correct up to second order in $\Delta \xi$) is then:

$$\begin{aligned} \frac{\partial}{\partial t} \bar{u}(\xi_{i+\frac{1}{2}}, t) &= \left(\frac{(1 - \xi_i) \dot{s}_{m-1} + \xi_i \dot{s}_m}{s_m - s_{m-1}} \right) \frac{\bar{u}(\xi_{i+\frac{3}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{2\Delta \xi} \\ &- \left(\frac{1}{s_m - s_{m-1}} \right) \frac{p(\xi_{i+1}, t) - p(\xi_i, t)}{\Delta \xi} \\ &+ \frac{\mu}{(s_m - s_{m-1})^2} \frac{1}{\Delta \xi^2} \left[\left(\frac{\bar{u}(\xi_{i+\frac{3}{2}}, t) - \bar{u}(\xi_{i+\frac{1}{2}}, t)}{v(\xi_{i+\frac{1}{2}}, t)} \right) - \left(\frac{\bar{u}(\xi_{i+\frac{1}{2}}, t) - \bar{u}(\xi_{i-\frac{1}{2}}, t)}{v(\xi_i, t)} \right) \right]. \end{aligned} \quad (13)$$

Which is again valid for $i = 1, 2, \dots, N-1$.

At this point it should be stressed that the formulation presented thus far has produced a system of ordinary differential equations for what will be referred to as *internal* unknowns. What this means is that given values of the state variables (\bar{u} and v) on the *entire* grid:

$$\bar{\mathbf{u}}_{\frac{1}{2}}, \mathbf{v}_1, \bar{u}_{\frac{3}{2}}, v_2, \dots, v_{N-2}, \bar{u}_{N-\frac{3}{2}}, v_{N-1}, \bar{u}_{N-\frac{1}{2}}, \mathbf{v}_N, \bar{\mathbf{u}}_{N+\frac{1}{2}}, \quad (14)$$

using the ordinary differential equations defined above we can determine the new values of the state variables located at the *internal* unknowns only:

$$\bar{u}_{\frac{3}{2}}, v_2, \dots, v_{N-2}, \bar{u}_{N-\frac{3}{2}}, v_{N-1}, \bar{u}_{N-\frac{1}{2}} \quad (15)$$

In equation (14) we have printed the boundary nodes in bold. These unknowns are represented by the variables shown in dark in figure 4, (although with a slightly different notation). While this

may seem to be a minor point, the numerical formulation developed below to solve the resulting ordinary differential equations for the internal unknowns is *not* restricted to only operate on systems of ODE's. Specifically, the methodology discribed below, transforms the ODE's above into a nonlinear system of equations. The numerical solution of this large system of ordinary differential equations nominally requires an equation *to be specified* for each unknown represented in equation (14), more specifically an equation must be provided for the *boundary* unknowns:

$$\bar{\mathbf{u}}_{\frac{1}{2}}, \mathbf{v}_1 \quad \text{and} \quad \mathbf{v}_N, \bar{\mathbf{u}}_{N+\frac{1}{2}}. \quad (16)$$

The discretization we persue thus allows for boundary conditions to be specified as a purly *algebraic* relationship among the state variables, and in fact is how many of the boundary conditons are implemented¹.

3.2 Discretization of the time derivatives

In this subsection, we complete the construction of the viscous numerical algorithm. We do so by presenting the strategy used to discretize all time derivatives remaining in the the semi-discrete equations derived in the earlier section. In this section, we first discuss the full discretization performed for the inner variables and then describe how the resulting nonlinear system easily generalized to include the outer unknowns which may not necessarily be the solution to a differential equation.

For the variables who's update is determined by a system of ODE's discribed in the previous section we use the so called θ -method, which leads to some of the most commonly used (simple) algorithms for time integration. As such, it is convenient to view the semi-discrete equations for the dissipative model can be written in the compact vector form for the resulting ordinary differential equations

$$\frac{d}{dt}Y = f(Y), \quad (17)$$

where $Y = Y(t)$ is a vector representing the solution to the problem, and f is the nonlinear vector function that follows upon writing the semi-discrete equations in terms of Y . Specifically, we can write Y as a vector having one block of entries per domain, with the blocks separated by the variable giving the position of the corresponding interface s_n . Furthermore, within each block let the Y entries alternate between the node centered specific volume values v_i and the cell centered flow velocity values $\bar{u}_{i+1/2}$. Specifically, given the diagram in Figure 4, one might view the unknowns where the given ordinary differential equations are defined as a total vector

¹This is in contrast to more classical methods where at each timestep the boundary unknowns are simply *assigned* their values, rather than computed under suitable constraints.

of unknowns $Y(t)$ (here representing only *one* domain)

$$Y(t) = \begin{pmatrix} \bar{u}_{\frac{3}{2}}(t) \\ v_2(t) \\ \bar{u}_{\frac{5}{2}}(t) \\ v_3(t) \\ \vdots \\ v_{N-2}(t) \\ \bar{u}_{N-\frac{3}{2}}(t) \\ v_{N-1}(t) \\ \bar{u}_{N-\frac{1}{2}}(t) \end{pmatrix} \quad (18)$$

Now let $0 \leq \theta \leq 1$ be a parameter, whose value can be used to control the accuracy and stability of the algorithm (see below). Introduce also a time discretization, with $t_{n+1} = t_n + \Delta t$ and $Y^n = Y(t_n)$. Then the θ -method algorithm for the ode in (17) is given by solving for ΔY^n in the nonlinear equation $F(\Delta Y^n, Y^n) = 0$ given by

$$F(\Delta Y^n, Y^n) \equiv \Delta Y^n - \Delta t f(Y^n + \theta \Delta Y^n) = 0, \quad (19)$$

where $\Delta Y^n = Y^{n+1} - Y^n$, and F is defined by the formula. It is well known that: when $\theta \geq 1/2$ this algorithm is unconditionally stable, and when $\theta < 1/2$ the algorithm is only conditionally stable [4]. Particular values of θ give rise to some common time integration strategies:

- $\theta = 0$ gives the explicit forward Euler scheme.
- $\theta = \frac{1}{2}$ gives the second-order, centered implicit trapezoidal rule (or Crank-Nicholson).
- $\theta = 1$ leads to the backwards implicit Euler method.

In our calculations we used a value of θ slightly above $1/2$ (in fact $\theta = 0.55$), to assure maximum stability and accuracy.

We note that Equation 19 while valid for solving the ordinary differential equations required to update the internal nodes) is actually more general in that it simply represents an *algebraic* relationship among the unknowns vector update ΔY^n . Thus we can now include any algebraic relationships among the boundary conditions in an understood way. Simply extend the definition of the unknown update vector ΔY^n to include these additional unknowns and *include* equations/constraints relating these unknowns to the rest of the domain. For example to include all boundary unknowns for a simulation involving only *two* domains we must expand ΔY to include the unknowns:

$$s_0, \bar{u}_{\frac{1}{2}}, v_1 \quad (20)$$

for the *left* boundary, the unknowns

$$v_N, \bar{u}_{N+\frac{1}{2}}, s_1, \bar{u}_{\frac{1}{2}}, v_1 \quad (21)$$

for the *internal* boundary, and finally the unknowns,

$$v_N, \bar{u}_{N+\frac{1}{2}}, s_2 \quad (22)$$

for the *right* boundary. Of course when considering two domains one has to include the internal unknowns present in each domain individually. Algebraically when we expand the definition of Y^n to include all unknowns we require an additional *equation* for each additional variable included so that the balance between number of equations and unknowns is always equal. For instance our vector to update the unknowns ΔY^n now becomes (for a problem with two domains):

$$\Delta Y = \begin{pmatrix} \Delta \mathbf{s}_0 \\ \Delta \bar{\mathbf{u}}_{\frac{1}{2}} \\ \Delta \mathbf{v}_1 \\ \Delta \bar{u}_{\frac{3}{2}}(t) \\ \Delta v_2(t) \\ \vdots \\ \Delta v_{N-1}(t) \\ \Delta \bar{u}_{N-\frac{1}{2}}(t) \\ \Delta \mathbf{v}_N \\ \Delta \bar{\mathbf{u}}_{N+\frac{1}{2}} \\ \Delta \mathbf{s}_1 \\ \Delta \bar{\mathbf{u}}_{\frac{1}{2}} \\ \Delta \mathbf{v}_1 \\ \Delta \bar{u}_{\frac{3}{2}}(t) \\ \Delta v_2(t) \\ \vdots \\ \Delta v_{N-1}(t) \\ \Delta \bar{u}_{N-\frac{1}{2}}(t) \\ \Delta \mathbf{v}_N \\ \Delta \bar{\mathbf{u}}_{N+\frac{1}{2}} \\ \Delta \mathbf{s}_2 \end{pmatrix} \quad (23)$$

In the above the boundary unknowns are typeset in a bold font to allow understanding of the location these variables play in the global setting, in addition, I have dropped the n superscript on ΔY for simplicity. With this expanded unknown set, additional equations must be supplied to augment the previously defined vector valued F so that the “number of unknowns equals the number of equations“ and the system is uniquely solvable.

Given the current state Y^n , equation (19) is a (generally nonlinear) system of equations for the increment vector ΔY^n , which we solve using Newton’s method. This requires the calculation of the Jacobian of the nonlinear vector function F , which is a cumbersome but straightforward task—since, in fact, F is made up by fairly simple formulas. Since the equations used at

the interface change as the interface changes type, we monitor when this change take place, and then adjust the formulas for the function F and its Jacobian accordingly. An additional benefit of the formalism given in equation 19 is that it is convenient and simple to impose *algebraic* constraints among the variables (specifically the unknowns at the boundary) composing the vector unknown Y . In fact this method (setting up a nonlinear system of equations to be solved is how the *boundary conditions* are implemented, see Section ??). A unique consequence of this fact is that for the numerical method presented here, rather than imposing boundary conditions at cells/nodes “outside” of each domain (as is commonly done in classical numerical techniques) we impose them on the first few unknowns *inside* each domain.

3.3 Spatial discretization of the boundary conditions

The discretizations described above are, of course, valid only within each domain—where the relationship between p and v , as given by the equation of state, is smooth. The discretizations for the conservation of mass and momentum equations in (9) and (13) both require information about the specific volume and the velocity one node and one cell away from the cell or node on which the time derivative terms in the semi-discrete equations are centered: these equations cannot be used at points close to a boundary. We thus need to consider what happens at the end-points of each domain, where two type of situations may arise: an end-point may either coincide with the boundary of the sample (i.e.: s_i for $i = 0$ or $i = M$), or it may correspond to the location of a phase transition (i.e.: s_i for $0 < i < M$). This leads to two different types of boundary conditions at end-points; we call these *outer boundary conditions* and *inner boundary conditions*, respectively. Clearly we may restrict our discussion of boundary conditions to the two-domain case $M = 2$, such as that depicted in Figure 1—where a single right-moving phase transformation front separates the high-density phase on the left from the low-density phase on the right. In this case, as indicated in the figure caption, we will call the high-density domain the domain A, and the low-density domain the domain B—note that the precursor shock wave in the low-density phase is not a discontinuity (since dissipative equations are being used) and it therefore is not a domain boundary.

In the numerical calculations considered in this paper we are not interested in computing for times so large that wave reflections from the back end of the impactor reach the free surface. In the particular case of two domains, such as the one in Figure 1, this means that we can *assume that the left boundary of domain A is at negative infinity*—with the right boundary at the phase transformation, located at $s(t)$. Domain B, on the other hand, is bounded on its left by the phase transformation front, and on its right by the free surface, at ξ_R . Finally, as noted in [2, 5], the phase transformation wave will change type as the flow evolves: from a right moving phase transformation front, to a contact discontinuity, to a left moving phase transformation front. Thus the domain boundaries are not always phase transformation fronts, and, as part of the evolution of the system, will switch type. *Generally, the domain boundaries are just discontinuities (or interfaces), separating the high-density phase on one side from the low-density phase on the other.*

In the next Sections we present the supplemental equations used to solve for the nodes and cells near the inner and outer boundary conditions mentioned earlier in this section. The exact conditions used at any given interface depend on the type of discontinuity separating

the domains, be it a forward transformation front, a contact discontinuity, or a backwards transformation front.

3.3.1 Outer boundary conditions

The stencils of the semi-discrete equations in (9) and (13) require the specification of variables one node and one cell to either side of the node or the cell at the stencil center. As a consequence of this fact, two boundary conditions are required on each outer boundary. This is consistent with the increase in order of the model partial differential equations, due to the added dissipative term. In what follows we denote the left and right end-points of the computational domain by $\xi_L = s_0$ and $\xi_R = s_M$.

To model numerically the semi-infinite nature of domain A we argue that, for a sufficiently large and negative value of ξ , the specific volume should be constant. Thus we set $v = v_L$ at $\xi = \xi_L$, where v_L is obtained from the solution of the first Riemann problem (the collision of the impactor and the flyer), a calculation that is performed before the full viscous flow computation is started. This gives one of the two boundary conditions needed on the left end of the numerical domain. Similarly, we argue that on the right end of domain B the specific volume should also be a constant, i.e.: $v = v_R$ on ξ_R , where the constant is determined by the equation $p(v_R) = 0$. This follows because the right boundary of domain B is a free surface, and cannot support any pressure. Finally, to determine the second boundary condition required at each end, we use the conservation of mass equation in (2). From this equation it is clear that, if v is constant along a particle path $\xi = \text{constant}$, we must have

$$\frac{\partial}{\partial \xi} u = 0. \tag{24}$$

This gives a second boundary condition, applicable on any $\xi = \text{constant}$ boundary where v is kept constant.

3.3.2 Inner boundary conditions

Here we formulate the boundary conditions used at the interface between the two domains, A and B, across which there is a jump in the equation of state (the high-density phase on one side and the low-density phase on the other). In detail, we consider all the unknowns/variables (at or near the interface) that cannot be updated by either of the equations in (9) or (13), and we systematically provide additional equations that can be used to update their values.

We begin by considering the immediate neighborhood of the interface and the associated unknowns. Figure 3 shows a schematic representation of the situation. In this figure the interface is denoted with a cross, the nodes with solid dots and the cell centers with open dots. There are some *important clarifications that must be made with regard with this figure*: the rightmost node of the left domain (node N in the figure), the leftmost node of the right domain (node 1 in the figure), and the interface lie at the *same* point in space, but we have represented them

separately in the figure because they play different roles in the numerical algorithm. Similarly, there is no cell N belonging to the left domain, nor is there a cell 0 belonging to the right domain, but (for numerical reasons, which will become clear in our discussion below) these *ghost cells* have been introduced in the figure.

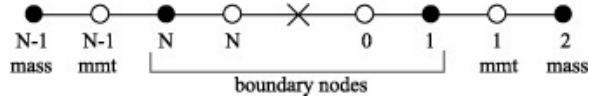


Figure 3: Schematic representation of the numerical grid near an interface. The interface is indicated by a cross, the nodes by solid dots and the cell centers by open dots. The rightmost (N -th) node of the left domain, the leftmost (1st) node of the right domain, and the interface are the *same* point in space, but, since they play different roles in the numerical algorithm, they are represented separately in the diagram. Similarly, there is no physical cell N belonging to the left domain, nor is there a physical cell 0 belonging to the right domain, but for reasons explained in the text, these *ghost cells* are introduced by the numerical algorithm—they represent extensions of the material phases from each side of the interface.

The representation of the numerical grid shown in Figure 3 provides a convenient way to label the unknowns around the interface. In this representation, each node and cell center carries one unknown (the values of the specific volume and velocity, respectively) while the interface carries two unknowns, namely: its Lagrangian coordinate position $\xi = s(t)$, and the mass flow across the interface $\mathcal{M} = \dot{s}$ —both as introduced by the transformation in (3), mapping the equations to a fixed $[0, 1]$ grid.

To update (in time) the specific volume in the node labeled $N - 1$, the formula in equation (9) can be used, since an additional cell and node exist to either side of this node. The same statement can be made about the cell center labeled $N - 1$, using the formula in equation (13). Similarly, the cell center labeled 1, and the node labeled 2, can be updated using the semi-discrete formulas in (9) and (13). This still leaves six variables for which the regular stencil used in the semi-discrete equations cannot be used, and for which extra equations are needed, namely:

$$v_N, \quad \bar{u}_N, \quad s, \quad \dot{s}, \quad \bar{u}_0, \quad \text{and} \quad v_1.$$

The first extra equation is obtained by approximating the mass flux through the interface ($\mathcal{M} = \dot{s}$) by the finite difference formula

$$\dot{s} \approx \frac{s^{n+1} - s^n}{\Delta t}, \tag{25}$$

where s^n is the value of $s = s(t)$ at time $t = t_n$, and $\Delta t = t_{n+1} - t_n$ (higher order approximations for the time derivative are also possible, of course, but this is the one that we used). The other five extra equations will depend on the type of interface between the two domains; briefly, two equations follow from the Rankine-Hugoniot jump conditions (conservation of mass and momentum across the interface), one equation from knowledge of the specific interface type, and the remaining two equations are obtained from a certain numerical extrapolation process to be described below. These extra equations are given in their continuous forms in what

follows; the corresponding discrete expressions are presented in Section 4. There are two cases to consider, depending on the interface type.

Case 1: the interface is a forward or backwards transformation front

Two of the equations follow from the conservation of mass and momentum across the interface, where care must be taken to include the fluxes due to the dissipative terms. Modifying the standard Lagrangian Rankine-Hugoniot equations to include the fluxes due to dissipation, we obtain

$$\dot{s} [v] + [u] = 0 \quad \text{and} \quad \dot{s} [u] - \left[p(v) - \frac{\mu}{v} \frac{\partial}{\partial \xi} u \right] = 0, \quad (26)$$

where the partial derivative u_ξ must be computed without crossing the discontinuity—it is because of the need to compute this derivative that the ghost cells were introduced earlier. We note that, in order to derive these equations, we have assumed that *there is no viscous contribution to the momentum flux arising from the phase transformation*. Mathematically this means that singularities of order higher than Dirac deltas are ignored. These singularities arise from the dissipative term in (2), because the variables are discontinuous across the interface. After this is done, the second jump condition above in (26) guarantees the cancellation of the remaining distribution (Dirac’s deltas) part in the momentum equation in (2).

A further condition, that is specific to transformation fronts (forward and backwards), follows from the fact that the pressure ahead of the wave must be at the critical value for the phase into which the wave propagates. This gives a third equation

$$p_N = p_{\text{crit}}^{\{\text{LD,HD}\}} \quad \text{or} \quad p_1 = p_{\text{crit}}^{\{\text{LD,HD}\}}, \quad (27)$$

where the specific formula used depends on the direction of propagation of the interface, and the type of the phase transformation.

Finally, the two remaining boundary conditions are obtained from a numerical approximation. As explained above, the ghost cells are introduced because of the need to compute the derivatives u_ξ in the right equation in (26). However, this is meaningless unless values for the velocities are provided at the ghost cell centers. A reasonable approach to doing this is to use extrapolation of the values of the velocity, from the inside the domain the ghost cell belongs to. We found, through experimentation with various possibilities, that first order extrapolation gives reasonable results for this purpose. This yields, on the uniform grids we used (see (6)), the following equations for \bar{u}_N and \bar{u}_0

$$\bar{u}_N = 2\bar{u}_{N-1} - \bar{u}_{N-2} \quad \text{and} \quad \bar{u}_0 = 2\bar{u}_1 - \bar{u}_2. \quad (28)$$

Case 2: the interface is an interior contact discontinuity

In this case

$$\frac{ds}{dt} = 0, \quad (29)$$

because no mass crosses the interface. This gives a condition specific to contact discontinuities only. Conservation of mass and momentum now take the simpler forms

$$[u] = 0 \quad \text{and} \quad \left[p(v) - \frac{\mu}{v} \frac{\partial}{\partial \xi} u \right] = 0, \quad (30)$$

where the same remarks made after the equations in (26) apply. Finally, two additional (numerical) boundary conditions are obtained by manipulation of the conservation of mass differential equation in (4). This equation yields, when evaluated on the N -th node:

$$\frac{d}{dt} v_N - \frac{1}{s - \xi_L} \frac{\partial}{\partial \xi} \bar{u}_N = 0,$$

where we have used the fact that we have only two domains, that the node is actually a particle path so that $\dot{s} = 0$, and that the cell N and the node N are the same point in space (so that the node centered and cell centered derivative are equivalent). A semi-discrete equation for v_N can now be obtained by discretizing the spatial derivative of the velocity in this equation, using finite differences. This gives

$$\frac{d}{dt} v_N - \frac{1}{s - \xi_L} \frac{3\bar{u}_N - 4\bar{u}_{N-1} + \bar{u}_{N-2}}{2\Delta\xi} = 0, \quad (31)$$

where the velocity value at the center of the ghost cell (i.e.: \bar{u}_N) follows by the same extrapolation process used earlier in equation (28). A similar procedure generates a semi-discrete equation for v_1 .

In Section 4 we present a detailed summary of all the semi-discrete equations used to update each node, cell, and interface (i.e.: s) at the boundary between domains. With this information, the numerical scheme is completely defined once we specify how to discretize the time derivatives. This is done with the rather well known “ θ -method” and as fully described in, Subsection 3.2. In the next section, we describe the results obtained using this numerical model.

4 Summary of the interface boundary conditions used in the viscous code

In section 3 a numerical algorithm was developed for the viscous Lagrangian conservation of mass and momentum equations (2), for a region containing several domains separated by moving discontinuities. In Section 3.3, boundary conditions were derived for the nodes and cells around the interface between two domains. Here we summarize the semi-discrete and fully discrete equations used at each node and cell near the interface. See Figure 3 for the notation of the

cell and node centered values used here. As we did in the body of this article, for simplicity we consider here the case where the forward transformation front is right-facing while the backwards transformation front is left-facing. In this case we present a summary of the equations used for each variable around the interface, written as a system of equations.

1. For the *left end* of the domain the interface is a contact discontinuity and we require three equations to update the state variables ($s_0(t)$, \bar{u}_0 , and v_1).

- Since the interface is a contact no mass flows across this interface and in Lagrangian to update $s_0(t)$ we use

$$\frac{ds_0(t)}{dt} = 0$$

- To update \bar{u}_0 we use extrapolation from inside the domain

$$\bar{u}_0 - \bar{u}_1 = 0$$

- To update v_1 the pressure must be kept at critical or

$$p(v_1) = p_{\text{crit}}^{\text{HD}}$$

2. If the interface is a *forward transformation front*:

- Update v_N using a discretization of the conservation of mass jump condition (26)

$$\dot{s} (v_N - v_1) + \left(\frac{1}{2}(\bar{u}_N + \bar{u}_{N-1}) - \frac{1}{2}(\bar{u}_0 + \bar{u}_1) \right) = 0.$$

- Update \bar{u}_N using first order extrapolation of velocity (28)

$$\bar{u}_N - 2\bar{u}_{N-1} + \bar{u}_{N-2} = 0.$$

- Update s using a discretization of the conservation of momentum jump condition (26)

$$\begin{aligned} \dot{s} \left(\frac{1}{2}(\bar{u}_N + \bar{u}_{N-1}) - \frac{1}{2}(\bar{u}_0 + \bar{u}_1) \right) - (p(v_N) - p(v_1)) \\ + \frac{\mu}{\Delta\xi} \left(\frac{\bar{u}_N - \bar{u}_{N-1}}{v_N} - \frac{\bar{u}_1 - \bar{u}_0}{v_1} \right) = 0. \end{aligned}$$

- Update \bar{u}_0 using first order extrapolation of velocity (28)

$$\bar{u}_0 - 2\bar{u}_1 + \bar{u}_2 = 0.$$

- Update v_1 using the critical condition (27)

$$p(v_1) - p_{\text{crit}}^{\text{LD}} = 0.$$

3. If the interface is a *contact discontinuity*:

- Update v_N using the conservation of mass partial differential equation (31), in domain A evaluated at the local node $\xi = 1$.

$$\frac{d}{dt} v_N - \frac{1}{s - \xi_L} \left(\frac{3\bar{u}_N - 4\bar{u}_{N-1} + \bar{u}_{N-2}}{2\Delta\xi} \right) = 0.$$

- Update \bar{u}_N with the conservation of mass jump condition (30)

$$\frac{1}{2} (\bar{u}_N + \bar{u}_{N-1}) - \frac{1}{2} (\bar{u}_0 + \bar{u}_1) = 0.$$

- Update s with (29)

$$\frac{ds}{dt} = 0.$$

- Update \bar{u}_0 with the conservation of momentum jump condition (30)

$$- (p(v_N) - p(v_1)) + \frac{\mu}{\Delta\xi} \left(\frac{\bar{u}_N - \bar{u}_{N-1}}{v_N} + \frac{\bar{u}_1 - \bar{u}_0}{v_1} \right) = 0.$$

- Update v_1 using the conservation of mass partial differential equation (4) in domain B, evaluated at the local node $\xi = 0$ with the one sided differences

$$\frac{d}{dt} v_1 - \frac{1}{\xi_R - s} \left(\frac{-3\bar{u}_1 + 4\bar{u}_2 - \bar{u}_3}{2\Delta\xi} \right) = 0.$$

4. If the interface is a *backwards transformation front*:

- Update v_N with the critical pressure condition (27)

$$p(v_N) - p_{\text{crit}}^{\text{HD}} = 0.$$

- Update \bar{u}_N with first order extrapolation of velocity (28)

$$\bar{u}_N - 2\bar{u}_{N-1} + \bar{u}_{N-2} = 0.$$

- Update s using the conservation of momentum jump conditions (26)

$$\begin{aligned} \dot{s} \left(\frac{1}{2}(\bar{u}_N + \bar{u}_{N-1}) - \frac{1}{2}(\bar{u}_0 + \bar{u}_1) \right) - (p(v_N) - p(v_1)) \\ + \frac{\mu}{\Delta\xi} \left(\frac{\bar{u}_N - \bar{u}_{N-1}}{v_N} - \frac{\bar{u}_1 - \bar{u}_0}{v_1} \right) = 0. \end{aligned}$$

- Update \bar{u}_0 using first order extrapolation of the velocity (28)

$$\bar{u}_0 - 2\bar{u}_1 + \bar{u}_2 = 0.$$

- Update v_1 using conservation of mass jump condition (26)

$$\dot{s} (v_N - v_1) + \left(\frac{1}{2}(\bar{u}_N + \bar{u}_{N-1}) - \frac{1}{2}(\bar{u}_0 + \bar{u}_1) \right) = 0.$$

5. For the *right end* of the domain the interface is again a contact discontinuity and we require three equations to update the state variables (v_N , \bar{u}_N , and $s_M(t)$).

- To update v_N the pressure vanish on the boundary

$$p(v_N) = 0$$

- To update \bar{u}_N we use extrapolation from inside the domain

$$\bar{u}_N - \bar{u}_{N-1} = 0$$

- Since the interface is a contact no mass flows across this interface and in Lagrangian to update $s_M(t)$ we use

$$\frac{ds_M(t)}{dt} = 0$$

With these specifications, the numerical scheme is well defined and can be implemented. Care must be taken, of course, to insure that, when the interface changes type, these changes are detected and the correct equations are used.

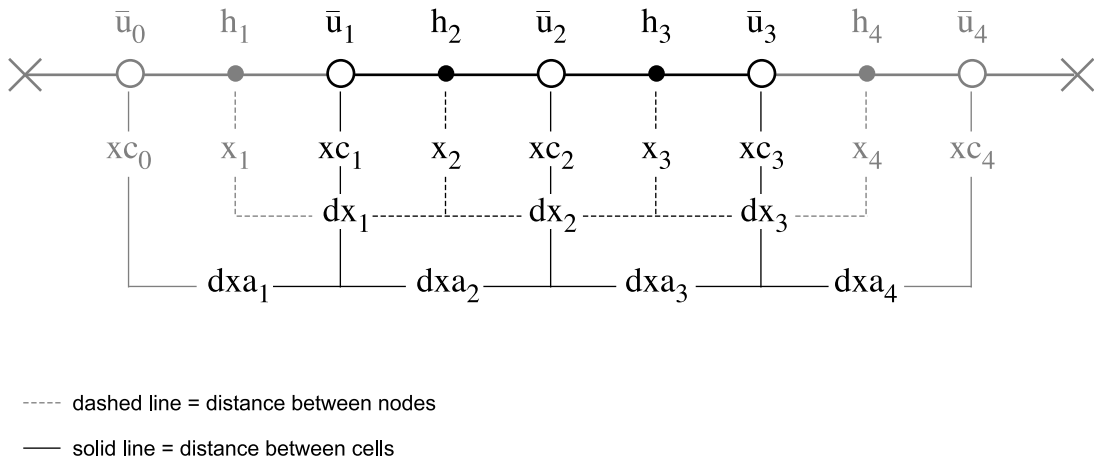


Figure 4: A representative grid when $k(d) = 4$. Note that the *cells* begin and end the grid with the nodes defined at the intermediate locations starting at index “2”. Unknowns are represented in dark bold while boundary nodes are shown in gray.

5 A Detailed Description of the Numerical Code

The code to solve the viscous CS model consists of two main programs. The first called `id.out` and is the result of compiling the program module contained in the file `Init.f`. This program is responsible for setting up the initial conditions and any needed parameters for the second program `mnn.out`. This second program is compiled from the program contained in the file `Hsn.f`. Thus once the appropriate input parameters are set, a complete execution of the code is given by entering at the shell prompt.

```
$ id.out; mnn.out
```

Before I go into the details of the existing code it can help to have a very high level overview. Basically the code works as follows. Using the analytic expression for a propagating viscous shock traveling at a known speed, we compute the “location” of the precursor shock and the phase transformation front for the given impacts. Implicitly this implies that the code, as written, will only work for impacts that produce these split waves. This is of no great loss since if the impact does not produce a split wave the viscous and non-viscous critical stress theories agree quite well in their predictions of the free surface. This is because in the cases where a split wave is *not* present, during the experimental time interval, no waves have time to return to the free surface after their creation.

For the specific cases where the impact creates a split wave we naturally have two domains separated initially by a forward phase transformation front. The program `id.out` provides the initial conditions in each domain to the program `mnn.out`. The initial conditions in the left domain are taken to be *constant* and depending on the results of the states produced by the inviscid Riemann solver. The initial conditions in the right domain are taken from an analytic expression for a traveling *viscous* shock centered at the location of the inviscid shock location.

6 A Psuedo-Code Description of the Viscous Code

In this section, I'll give a high level overview of how the viscous code operates commenting on subroutine names where specific operations are performed. The main timestepping routine is `Hsn.f` which after some code setup enters a looping structure to compute the solution profiles, from the initial time to the final time `tfinal`. Specifically the main work performed by this code can be represented as:

```
c   Pseudo code for the program: Hsn.f

do while ( time .lt. tfinal )
  do while ( time .lt. ptime )
    call updateViscosity()
    dt = cflTimeStep()
    call trystp()
    call setStressArr()
    call ...
  enddo
  call prfout()
  call prfOutAll()
enddo
```

In the above, we continue looping until the current time variable `time` reaches `tfinal`. We print out the solution profiles using the routines `prfout.f` and `prfOutAll.f` at specific points in time `ptime`. The routine `trystp.f` assumes a valid solution at all the unknowns and attempts to perform a propagation step producing a new consistent unknowns set at time `dt` later.

The subroutine `trystp.f` is considered next. This routine basically performs the newton iterations to determine the value of ΔY for this timestep, see section 3.2. In addition, this subroutine will loop over each interface and see if any potentially change type during this timestep. If any internal waves changes type during this timestep logic is exercise to reset the states to their values before the step `dt` and to use a bisection method to determine *exactly* the `dt` where the change in type takes place. This ensures a very precise estimate of the states around this very important time. Depending on the type of wave encountered (be it forward, contact, or backwards transformation front) a different functional value is used to perform this bisection. After a succesful timestep of the unknowns in this problem, the solution is updated permanently by calling `solUpdate.f`. At a highlevel the routine `trystp.f` looks like this

```
c   Pseudo code for the function: trystp.f

c   Try a normal newton step.
    call newtonITER()

c   Complicated logic for when an interface changes type goes here
c   We reset all unknowns to their initial time, compute
c   the exact location (in time) where the interface changes
```

```

c     type using the bisection method on the timestep dt ... not discribed in detail here

c     Update the global time.
      call solUpdate()
      time = time + dt

```

We next consider the most computationally intensive portion of the entire code, specifically the section involved in performing Newton iterations to update all of the dependent variables.

Newton's Method for all Grid Unknowns

Next we'll consider the routine `newtonITER.f`, which is responsible for setting up and performing the newton iterations required to compute ΔY (the amount required to update each unknowns) when moving forward in time by the amount `dt`. Since we are looking for a root to the equation $F(\Delta Y) = 0$, newton's method in this case takes the form

$$\Delta Y^{n+1} - \Delta Y^n = - (F'(\Delta Y^n))^{-1} F(\Delta Y^n) \quad (32)$$

here the superscript represents the Newton iteration index and not the timestep iteration index (many newton iterations may be required for a single timestep). The pseudocode for this routine is presented next:

```

c     Pseudo code for the function: newtonITER.f

c     Compute an initial guess for the update
      call initialGuess()

c     Here we do a maximum of 'NumberOfIteration' Newton iterations
c     to compute the nonlinear root to update with.
      do numit=1,NumberOfIterations

c         Given the update variable \Delta Y^{n} at this iteration number,
c         compute the state Y^n (Y^n = Y^0 + \Delta Y^{n})
          call midTimeGuess()

c         Compute some auxiliary variables that depend on the underlying state Y^n
c         needed to simplify calculations later
          call gvh_der_and_int()

c         Form the residual of the nonlinear function.
          call resid( ... rh,rv,rb)

c         Check if we have converged (by looking at the residuals)
          if (IsNewtonFinished()) goto 1

```

```

c      Form the entries in the Jacobian matrix  $F'(\Delta Y^n)$ 
      call formcf( ... hc,vc,hb,vb,bc_L,bc,bb)

c      LU factor the Jacobian matrix for ease of inverting:
      call hsmfac(k,k_domains, hc,vc,hb,vb,bc_L,bc,bb,rh,rv,rb)

c      Now backsubstitute (solve) with a right hand side given by the residuals:
      call hmsol(k,k_domains,hc,vc,hb,vb,bb, rh,rv,rb)

c      Update dh and dv (unkowns)
      call updateNewtIter()

      enddo
      pause 'newtonITER.f: cannot converge'

1      continue

```

From the above code one can see that the routine `newtonITER.f` is responsible for computing the residuals (by evaluating $F(\Delta Y)$), forming the Newton derivative matrix $F'(\Delta Y)$ in the routine `formcf.f`, performing its LU decomposition (in the routine `hsmfac.f`) and its solution (in `hmsol.f`). Some things about our formulation are to be mentioned. First the Jacobian is not assembled in a huge M by N matrix but rather (to save memory) held in several smaller linear arrays. In addition, the unknown state is also not stored in a single vector but in several smaller vectors. This makes the understanding of the routines `formcf.f`, `hsmfac.f`, and `hmsol.f` more difficult. Since these issues are at the core of any difficulty understanding the numerical code I'll discuss them in more detail here.

The first observation to make is that there are more than one way of ordering the system of nonlinear equations we must solve. First if we consider the unknowns in a linear ordering i.e. in the way shown in the Equation 23, we can see that since both Equations 9, and 13 involve the s unknowns our Jacobian matrix will be have better structure for performing the LU factorization if these unknowns are *reordered* so that the boundary unknowns (Equations 20, 21, and 22) occur *last*. This intuitively gives the following representation for our global unknown state vector

ΔY :

$$Y = \begin{pmatrix} \bar{u}_{\frac{3}{2}}(t) \\ v_2(t) \\ \vdots \\ v_{N-1}(t) \\ \bar{u}_{N-\frac{1}{2}}(t) \\ \hline \bar{u}_{\frac{3}{2}}(t) \\ v_2(t) \\ \vdots \\ v_{N-1}(t) \\ \bar{u}_{N-\frac{1}{2}}(t) \\ \hline \mathbf{s}_0 \\ \bar{\mathbf{u}}_{\frac{1}{2}} \\ \mathbf{v}_1 \\ \hline \mathbf{v}_N \\ \bar{\mathbf{u}}_{N+\frac{1}{2}} \\ \mathbf{s}_1 \\ \bar{\mathbf{u}}_{\frac{1}{2}} \\ \mathbf{v}_1 \\ \hline \mathbf{v}_N \\ \bar{\mathbf{u}}_{N+\frac{1}{2}} \\ \mathbf{s}_2 \end{pmatrix} = \begin{pmatrix} v_1(t) \\ h_2(t) \\ \vdots \\ h_{k(1)-1}(t) \\ v_{k(1)-1}(t) \\ \hline v_1(t) \\ h_2(t) \\ \vdots \\ h_{k(2)-1}(t) \\ v_{k(2)-1}(t) \\ \hline \mathbf{s}_0 \\ \mathbf{v}_0 \\ \mathbf{h}_1 \\ \hline \mathbf{h}_{k(1)} \\ \mathbf{v}_{k(1)} \\ \mathbf{s}_1 \\ \mathbf{v}_0 \\ \mathbf{h}_1 \\ \hline \mathbf{h}_{k(2)} \\ \mathbf{v}_{k(2)} \\ \mathbf{s}_2 \end{pmatrix} \tag{33}$$

Here in each column like representation of the total set of unknowns, I have drawn horizontal lines to emphasis the decomposition into blocks that correspond to physical domains. In addition, the first column of unknowns correspond to the notation used in the previous sections of this document while the second column of unknowns correspond to the notation used in the numerical code. This equation then provides an *explicit* translation of each unknown introduced earlier in this paper into the notation used in the code (where the symbol \mathbf{h} is used for specific volumn, and \mathbf{v} for velocity). As mentioned above the point of this rearrangement is that the resulting matrix (representing $F'(\Delta Y)$) has much nicer structure (for performing an LU decomposition). An example of this matrix structure will make the understanding of the LU factorization routine `hsmfac.f` easier to understand. As such, an example of the type of matrix structure that arrises for the Jacobian with this unknown ordering consider Figure 5, where the matrix structure is

displayed for $k(1) = 8$, $k(2) = 8$, and boundary conditions as discussed in Section 4.

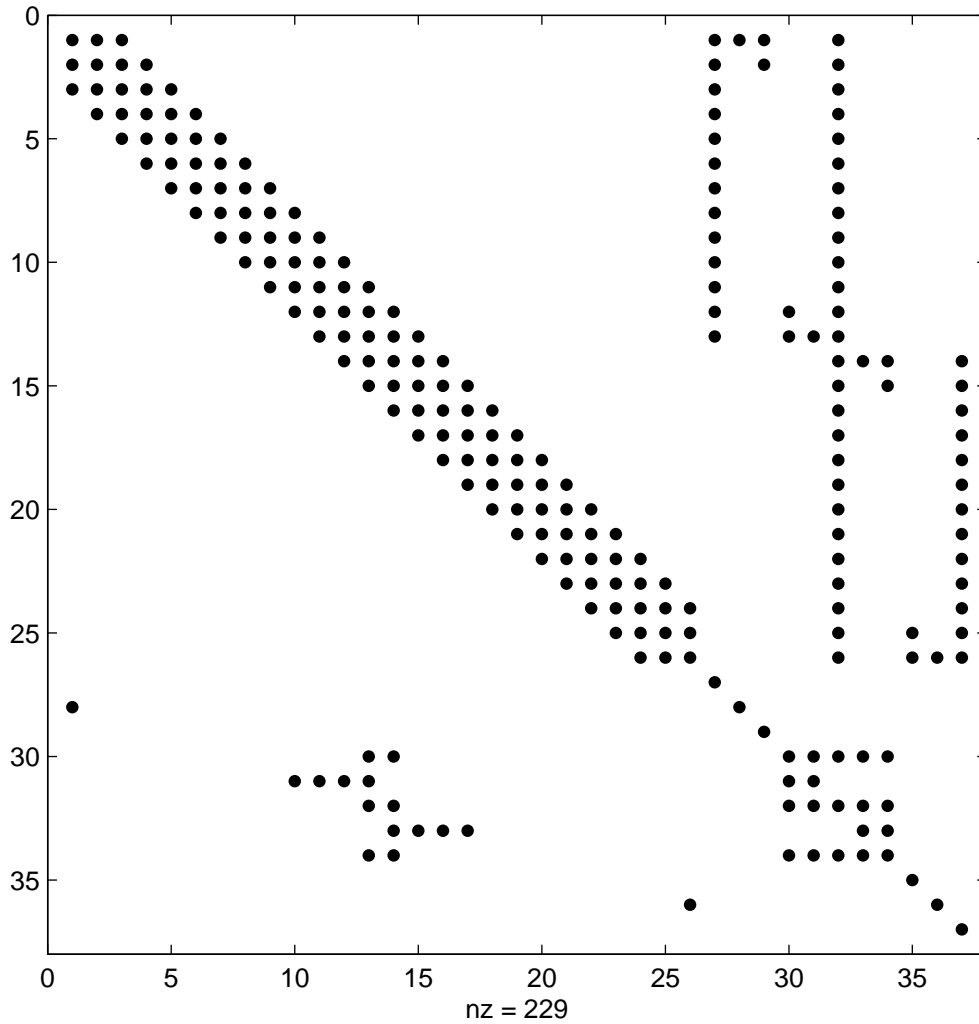


Figure 5: The representative Jacobian matrix structure for a two domain problem with $k(1) = 8$, $k(2) = 8$ and boundary conditions given as in section 4.

Continuing our discussion of the routines used in this code is the routine `resid.f` used for computing the residuals. As mentioned above the residuals are not stored in one large vector but are instead stored in three arrays, one for the boundary unknowns `rb`, one for the specific volumn unknowns `rh` and one for the velocity unknowns `rv`. For the boundary unknowns the dimensioning of the array is as follows

```
rb(-BW:BW,0:N_DOMAINS)
```

The “boundarywidth” variable `BW` is a convenient way to index the boundary unknowns. It ranges from -2 to +2 and represents the boundary unknowns we are considering from left to right. The number of domains `N_DOMAINS` variable is 2 and ranges over all interfaces in the

i	d	unknown
0	0	s_0
1	0	v_0
2	0	h_1
-2	1	$h_{k(1)}$
-1	1	$v_{k(1)}$
0	1	s_1
+2	1	v_0
+1	1	h_1
-2	2	$h_{k(2)}$
-1	2	$v_{k(2)}$
0	2	s_2

Table 1: The mapping from indices in the array `rb(i,d)` to the specific unknown’s residual. For example the element `rb(-1,1)` stores the residual for the boundary unknown $v_{k(1)}$.

code from 0 for the left most interface to 2 for the right most interface. Each element in this array represents a residual for the following unknowns correspondence (for `rb(i,d)`) is given in table 1. The residuals for the internal unknowns are stored in the arrays `rh` and `rv`, which are dimensioned as follows

```
rh(0:NMAXO,N_DOMAINS)
rv(0:NMAXO,N_DOMAINS)
```

The velocity residuals `rv`, in each domain, are computed for the equations for the velocity unknowns indexed by `1:k(1)-1`. The specific volumn residuals, `rh` correspondinly are computed for the unknowns indexed by `2:k(1)-1`. Inspection of the code and comments in `resid.f` makes how these are computed very clear. In that routine, the boundary residuals are computed first from left to right followed by the residuals of the internal unknowns.

After the residual calculation is performed we assemble the coefficient matrix representing the Jacobian. By definition this is a matrix with row entries given by the derivative of each equation with respect to all the other variables. As in the simplification of the storage of the residuals, the entries of this matrix are not entered into a dense matrix containing many zeros but are stored in a set of 6 arrays: `hc`, `vc`, `hb`, `vb`, `bc`, and `bb`. In the next few paragraphs we will discuss the meaning of each of these arrays.

The Coefficients of the Jacobian for the Internal Nodes: `hc`, `vc`, `hb`, and `vb`

We first consider the internal velocity and specific volumn unknowns. When Equation 9 and Equation 13 are converted into the form required by Equation 19 and the corresponding derivatives with respect to all the unknowns are taken in Equation 33. We can see that we will have non-zero coefficients for terms involving the node and cell to the left and right of the centered node *and* nonzero coefficients of the interface variables (s_{m-1} and s_m) to the left and right of the domain in which this internal unknown resides. The four arrays `hc`, `vc`, `hb`, and `vb`, hold

all of the non-zero coefficients for the internal nodes². Specifically, the arrays `hc` and `hb` hold the coefficients of the neighboring internal variables and the boundary unknowns for the specific volumn variable `h` respectively. In the same way `vc` and `vb` hold the coefficients of the boundary unknowns.

These four arrays are dimensioned as follows

```
hc(-BW:BW,0:NMAX0,N_DOMAINS)
vc(-BW:BW,0:NMAX0,N_DOMAINS)

hb(DOM_CEN_NUM_SPECIAL_EQS,0:NMAX0,N_DOMAINS)
vb(DOM_CEN_NUM_SPECIAL_EQS,0:NMAX0,N_DOMAINS)
```

In the declaration of the arrays used to hold the coefficients of the internal unknowns (`hc` and `vc`) the variable `BW` represents the stencil width of the spatial discretizations (2), since a given internal unknown must nessesarily involve the two unknowns to its left and right. The additional second and third dimensions in the arrays `hc` and `vc` correspond to domain indexing (as the third index) and the specific internal unknown indexing (as the second index). Since the internal specific volumn unknowns are indexed by $2:k(d)-1$ and the internal velocity unknowns are indexed by $1:k(d)-1$, these are the only valid values for the second index in the arrays `hc` and `vc`.

Since the equations describing the internal variables may involve the boundary unknowns on each end of the domain, the arrays `hb` and `vb` must contain space to store any non-zero coefficients of these unknowns. These coefficients are stored in the arrays `hb` and `vb`. The size of the first dimension of each array is specified by the variable `DOM_CEN_NUM_SPECIAL_EQS` which is defined to be 6, and represents the total possible number of boundary unknowns on either side of the given internal unknown. For example, in the second domain (defined by `d=2`) we have boundary unknowns on the left given by the interface speed, the cell centered boundary velocity, and the node centered specific volumn or

$$s_1, \quad v_0, \quad h_1,$$

while to the right we have boundary unknowns given by the specific volumn, velocity, and right interface speed i.e.

$$h_{k(2)}, \quad v_{k(2)}, \quad s_2.$$

All combined this gives a total of 6 possible additional unknowns we may need coefficients for. Note that most internal equations actually only require non-zero coefficients for the variables s_{m-1} and s_m . Finally, as in the declaration of `hc` and `vc`, the third index in the declaration of the arrays, `hb` and `vb` specifies the domain in which this internal unknowns equation falls. In the same way, the second index in the arrays `hb`, and `vb` specify the index of the specific unknown this row in the Jacobian corresponds to. In summary, for a domain with $k(d)$ specific volumn unknowns (including the boundary unknowns), valid entries for the elements of the Jacobian matrix are stored in

```
hc(-BW:BW, 2:k(d)-1, N_DOMAINS)
```

²The letter `c` stands for coefficient, and `b` stands for boundary.

second index: row	boundary unknown
-2	$h_{k(1)}$
-1	$v_{k(1)}$
0	s_1
+2	v_0
+1	h_1

Table 2: The mapping of the second index `row` in the array `bc(:,row,d)` and `bb(:,row,d)` to the specific boundary unknown's. For example, the array `bc(:, -1, 1)` holds the coefficients of the *internal* unknowns in the row of the Jacobian corresponding to the boundary constraint equation (defined for the boundary unknown $v_{k(1)}$). In addition, the array `bb(:, -1, 1)` holds the coefficients for the other *boundary* unknowns in the Jacobian of the same constraint equation. Note that in both examples we are considering the boundary unknowns corresponding to the *middle* interface (the second index is 1). This is the same convention used to represent the residuals of the boundary conditions (i.e. consider table 1).

```
vc(-BW:BW, 1:k(d)-1, N_DOMAINS)
```

```
hb(DOM_CEN_NUM_SPECIAL_EQS, 2:k(d)-1, N_DOMAINS)
```

```
vb(DOM_CEN_NUM_SPECIAL_EQS, 1:k(d)-1, N_DOMAINS)
```

When one looks at the code found in `formcf.f` which fills the arrays we have been discussing we see that these coefficients are computed in a very straightforward manner.

The Coefficients of the Jacobian for the Boundary Nodes: `bc` and `bb`

As demonstrated in figure 5, the coefficients of the equation representing the derivative of the constraint equation for each boundary node i.e. (ghost cells and interface unknowns) are stored in the bottom border of the total Jacobian matrix. These coefficients are stored in two arrays `bc` and `bb`. The reasoning for requiring two arrays is the following. As discussed earlier, each unknown including the boundary unknowns has an associated nonlinear constraint equation which must be satisfied when moving from one timestep to the next. The Jacobian will require nonzero coefficients in its matrix representation for all of the *other* unknown variables involved in the constraint equation. For a given boundary unknown, the first array `bb`, holds the coefficients of any *non boundary* unknowns that may appear in this specific nonlinear constraint equation. For example, the constraint equation for v_N is given in equation 31. This equation contains the non boundary unknowns given by \bar{u}_{N-1} and \bar{u}_{N-2} and will have nonzero entries in appropriate places (discussed below) in the array `bc`. The array `bc` is dimensioned in the code as follows

```
bc(-2*NMAX0:2*NMAX0, -BW:BW, 0:N_DOMAINS)
```

It is easiest to discuss the dimensioning of `bc` from the last index to the first. In `bc` the *third* index in the array specifies which interface we are considering. Specifically, for a two domain problem, an index of 0 implies the left most interface an index of 1 implies the central interface,

first index: i	internal unknown
\vdots	\vdots
-4	$h_{k(1)-2}$
-3	$v_{k(1)-2}$
-2	$h_{k(1)-1}$
-1	$v_{k(1)-1}$
0	unused
+1	v_1
+2	h_2
+3	v_2
+4	h_3
\vdots	\vdots

Table 3: The mapping of the first index i in the array `bc(i,row,d)` to the specific *internal* unknown whos entry in the Jacobian will be stored there. Above we explicitly show the mapping for the first (or middle) interface where $d = 1$. Other interfaces are similiar. We note that for the interfaces that border the edges of the computational domain half of the elements corresponding to the first index will be zero. For instance the left end of the sample is defined as $d = 0$ and will necessarily have zeros for the elements with i negative.

and an index of 2 the right most interface. The second index in the array `bc` denotes which of the boundary unknowns we are considering. Since $BW = 2$ as before this index ranges from -2 to +2. Table 2 contains an overview of what unknown the numerical value of the second index in `bc` corresponds to. Finally, the first index in `bc` represents storage for any non boundary coefficients that might be needed. The variables to the left of the given domain are specified with a negative first index, while those to the right of the given domain are specified with a positive first index. An example of this convention is shown in table 3.

The next array to consider is `bb`. For each of the constraint equations for the boundary unknowns, as `bc` stores the coefficients of the Jacobian for the internal unknowns, `bb` stores the coefficients of the Jacobian for the boundary unknowns. The dimensioning of the array `bb` is as follows

```
bb(-N_SHK_CEN:N_SHK_CEN,-BW:BW,0:N_DOMAINS)
```

Here the variable `N_SHK_CEN` is defined to be 5. As with the other arrays the index of the array `bb` maybe best understood working backwards. In `bb` the *third* index in the array specifies which interface we are considering. Specifically, for a two domain problem, an index of 0 implies the left most interface an index of 1 implies the central interface, and an index of 2 the right most interface. The second index in the array `bb` denotes which of the boundary unknowns we are considering. Since $BW = 2$ this index ranges from -2 to +2. Table 2 contains an overview of what unknown the numerical value of the second index in `bb` corresponds to. Finally, this first index select from among the boundary unknowns centered at the given interface *and* from boundary unknowns around the interfaces to the left and right of the interface upon which we are centered. Table 4 gives an exact example of the mapping from index to unknowns.

first index: i	internal unknown
-5	s_{d-1}
-4	v_0 (in $d1$)
-3	h_1 (in $d1$)
-2	$h_{k(d1)}$
-1	$v_{k(d1)}$
0	s_d
+1	v_0 (in dr)
+2	h_1 (in dr)
+3	$h_{k(dr)}$
+4	$v_{k(dr)}$
+5	s_{d+1}

Table 4: The mapping of the first index i in the array $\mathbf{bb}(i, \text{row}, d)$ to the specific *boundary* unknowns whos entry in the Jacobian will be stored there. We show the mapping for an interface labeled d , denote the interface to the left by $d-1$, the interface to the right by $d+1$, the domain to the left of interface d as $d1$, and the domain to the right of interface d as dr . With these defintions the above gives the mapping from coefficient to element in the Jacobian. As before for the interfaces that border the edges of the computational domain will not have some of the indices defined. For instance the left end of the sample is defined as $d = 0$ and will necessarily have zeros for the elements of \mathbf{bb} with i negative. For the indexing of the grids we have been discussing the left and righth domains are given by $d1=d$ and $dr=d+1$.

Implicit Maximum Stencil Sizes For Each Unknown

The constants that dimension the arrays that hold the Jacobian, effectively enforce the largest stencil that one can use for each of the unknowns in the grid. In this small section we explicitly enoumerate these constraints. The code was written in such a way to allow for these variables to change values, but this aspect of the code was never tested or debugged. For example, the forward and backward solve routines should operate correctly if these generic dimensioning variable are changed but no proof that this will work has been observed yet by the authors.

For the internal unknowns the dimensioning of the variable \mathbf{BW} is such that each internal unknown can use only *two* elements to its immediate left and right. This is consistent with the variable $\text{DOM_CEN_NUM_SPECIAL_EQS}$ which when at 6, incorprates the two boundary unknowns on each side ($2 * 2 = 4$), plus the value of the interface speeds on either side (2) to give a total of 6.

For the boundary unknowns the dimensioning of the array \mathbf{bc} is such that the constraint equations on the boundary unknowns can involve the unknowns at *all* internal points of the grid. The dimensioning of the array \mathbf{bb} , is such that the constraints on the boundary unknowns can involve the boundary unknowns locally to the left and right of the interface, the nearest interface, the interfaces to the left and right of the nearest interface, and the boundary unknowns nearest to the left and right interface. This is best explained by revisting table 4

These comments suggest possible further improvements to the code. The compile time

constants `DOM_CEN_NUM_SPECIAL_EQS` and `N_SHK_CEN` should be made functions of the variable `BW`. Specifically,

```
DOM_CEN_NUM_SPECIAL_EQS = 2*BW + 1
N_SHK_CEN = 2*BW + 1
```

This modification would allow higher order schemes to be used for the internal discretizations if desired.

In summary then, from this discussion, it appears that for the discretization presented here almost any reasonable boundary condition for the interface unknowns is possible without any change of code. This finishes the discussion all the individual arrays which store the coefficients of the Jacobian. We next discuss the important points in the subroutine `formcf.f` where these elements are computed and these arrays filled.

The Jacobian Matrix Assembly routine `formcf.f`

With the descriptions given above, the routine `formcf.f` is easier to understand. Below we give pseudocode for this routine.

```
c      Pseudo code for the function: formcf.f

c-----
c      Fill the Jacobian for the equations governing the interfaces
c      i.e. fill the arrays bc and bb
c-----

c      Fill in the Jacobian for the equations governing the left interface
c      (when d=0), dd=d+1
c
c      call boundary_formcf_L(x,dx,dxa,dt,theta,k,d, bc_L,bc,bb)

c      Fill in the Jacobian for the equations governing the INTERNAL interfaces.
c      -- the entries depend on what type of interface we are considering --
c
c      do d=1,k_domains-1
c        if (disc_type(d) .eq. CRIT_TRANS_FRONT_FORW) then
c          if (disc_dir(d) .eq. dir01_r) then
c
c            call critTransFrontForwardCf(k,d, ... bc_L,bc,bb)
c
c          else
c            stop 'not implemented!'
c          endif
c        elseif (disc_type(d) .eq. CONTACT_DISC) then
```

```

        call contactDiscCf(k,d, ... bc_L,bc,bb)

elseif (disc_type(d) .eq. CRIT_TRANS_FRONT_BACK) then
    if (disc_dir(d) .eq. dir01_1) then

        call critTransFrontBackwardsCf(k,d, ..., bc_L,bc,bb)

    else
        stop 'not implemented!'
    endif
else
    stop 'not implemented!'
endif
enddo

c
c Fill in the Jacobian for the equations governing the right interface
c (when d=k_domains), dd=d
c
    call boundary_formcf_R(x,dx,dxa,dt,theta,k,d, bc_L,bc,bb)

c-----
c Fill the Jacobian for the equations governing the internal unknowns
c i.e. fill the arrays hc, vc, hb, and vb
c-----

do d=1,k_domains      ! loop over each domain

c Fill the vc array
do i=1,k(d)-1
    vc(-2,i,d) = stuff
c      ^ coefficient of delta v(i-1).

    vc(-1,i,d) = stuff
c      ^ coefficient of delta h(i).

    vc(0,i,d) = stuff
c      ^ coefficient of delta v(i).

    vc(1,i,d) = stuff
c      ^ coefficient of delta h(i+1).

    vc(2,i,d) = stuff
c      ^ coefficient of delta v(i+1).

    vb(1,i,d) = stuff

```

```

c          ^ coefficient of delta s(d-1).
      vb(6,i,d) = stuff
c          ^ coefficient of delta s(d).
      enddo

c      Fill in the vb array
      i=1
      vb(2,i,d)=vc(-2,i,d)
      vb(3,i,d)=vc(-1,i,d)
      i=k(d)-1
      vb(4,i,d)=vc(1,i,d)
      vb(5,i,d)=vc(2,i,d)

c      Fill in the hc array
      do i=2,k(d)-1
          hc(-2,i,d) = stuff
c          ^ coefficient of delta h(i-1).

          hc(-1,i,d) = stuff
c          ^ coefficient of delta v(i-1).

          hc(0,i,d) = stuff
c          ^ coefficient of delta h(i).

          hc(+1,i,d) = stuff
c          ^ coefficient of delta v(i).

          hc(+2,i,d) = stuff
c          ^ coefficient of delta h(i+1).

          hb(1,i,d) = stuff
c          ^ coefficient of delta s(d-1).
          hb(6,i,d) = stuff
c          ^ coefficient of delta s(d).
      enddo

c      Fill in the hb array
      i=2
      hb(3,i,d)=hc(-2,i,d)
      i=k(d)-1
      hb(4,i,d)=hc(2,i,d)

      enddo

```

Once this routine is finished the Jacobian is complete and a full Newton step requires its inverse. This is computed in the routine `hsmfac.f` which performs an LU decomposition of the Jacobian. The update coefficients are then determined by a call to `hsmso1.f` which performs back substitution using the LU factorization and the residual to calculate the required updates ΔY . Rather than discussing these routines in detail we now document what changes are required to modify the boundary conditions.

A Quick Guide For Modifying the Boundary Conditions in this Code

As the choice of boundary conditions in codes like these is often the most controversial part of an algorithm like this, in this subsection, we now discuss how one would add different boundary conditions to this code. We first discuss how the entries in the Jacobian are computed in terms of auxillary data structures computed elsewhere in the code. We begin with some examples of how to perform this calculation for the boundary conditons already used. The implementation of any additional boundary conditions should be an extension of these techniques. Most of the boundary conditions we consider are (nonlinear) constraints that must be held in moving from timestep n to $n + 1$. In general, the following constraint must be true

$$G(Y) = 0$$

Where Y is independent of time, because our implicit constraint requirements evaluate this function at $Y = Y^n + \theta\Delta Y^n$, see subsection 3.2, this requires enforcing

$$G(Y^n + \theta\Delta Y^n) = 0,$$

and solving for ΔY^n using Newton's method. This means that when we represent the scalar grid unknowns all derivatives with respect to the variables will involve multiplication by θ . Thus the derivative of G with respect to ΔY^n (our unknown) becomes

$$G'(Y^n + \theta\Delta Y^n)\theta.$$

In specifying the scalar elemets of ΔY^n we will use the notation δ . For instance, if the constraint equation involves the last specific volumn unknown $h_{k(d)}$, then this corresponding element in ΔY will be denoted by $\delta h_{k(d)}$. We begin our discussion by considering the boundary elements corresponding to the right most interface.

For this interface (in a two domain problem), boundary condition constraint equations must be specifed for each of the following unknowns

$$h_{k(2)}, \quad v_{k(2)}, \quad s_2.$$

As discussed in section 4 the constraint equation for $h_{k(2)}$ is

$$p(h_{k(2)}) = 0,$$

where $p(\cdot)$ is the known function of the pressure, equivalently the specific volumn h . This constraint involves only *one* unknown $h_{k(2)}$, and its derivative with respect to the updating variable $\delta h_{k(2)}$ is particularly simple

$$p'(h_{k(2)})\theta\delta h_{k(2)} = 0$$

or dividing by the nonzero term $p'(h_{k(2)})$ we obtain

$$\theta \delta h_{k(2)} = 0.$$

This constraint is implemented in the subroutine `boundary_formcf_R` where we see the line

```

      bb(-2,-2,2) = theta
c      ^ coefficient of delta h(k).

```

See table 2 and table 4 for a verification of the indexing used. Since this is the only nonzero coefficient for this row in the Jacobian, all other entries in the arrays `bc` and `bb` will be zero. We next consider the constraint equation for $v_{k(2)}$.

As discussed in section 4 the constraint required on $v_{k(2)}$ is that of first order extrapolation of velocity into the interior domain or

$$v_{k(2)} - v_{k(2)-1} = 0.$$

Again with each unknown composed of a base value and a perturbation, i.e. ($v = v^n + \theta \delta v$) the derivative of this equation with respect to each unknown δv is easy to calculate and is given by

$$\theta \delta v_{k(2)} - \theta \delta v_{k(2)-1} = 0.$$

This constraint is implemented in the subroutine `boundary_formcf_R` where we see the line

```

      bc(-1,-1,d) = -theta
c      ^ coefficient of delta v(k-1).
      bb(-1,-1,d) = theta
c      ^ coefficient of delta v(k).

```

See table 2 and table 4 for a verification of the indexing used. Finally we discuss the implementation of the constraint equation for $s_2(t)$.

As discussed in section 4 the constraint required on $s_2(t)$, is that of no mass flow across the interface or

$$\frac{ds_2(t)}{dt} = 0$$

This equation directly translates into the following constraint on the update δs_2

$$\left(\frac{1}{dt} \right) \delta s_2 = 0$$

This constraint is implemented in the subroutine `boundary_formcf_R` where we see the line

```
      bb(0,0,d) = 1.d0/dt
c      ^ coefficient of delta s(k_domains).
```

See table 2 and table 4 for a verification of the indexing used. As before the other entries in the arrays `bc` and `bb` are zero. We next present a more detailed description of some of the FORTRAN routines.

7 Detailed Subroutine Discussions

In the following section I will document any specific information needed that might help explain how a particular subroutine performs its work.

The subroutine: `settols.f`

Reads input from the file `accur.inp`. The variables are passed out from the subroutine call. The main variables in the subroutine call are

- `ttol`: total tolerance (global control over all the tolerance below) a typical value is 0.001 and this variable is *only* used in this subroutine in setting the values below.
- `tolh`:
- `tolv`:
- `tols`:
- `tolfh`:
- `tolfv`:
- `tolfs`:
- `tacc`:
- `cfx`:
- `cft`:
- `k_domains`: The number of domains this problem should solve over. This has never been tested as anything other than 2.
- `SizeInitGrid0`: An array holding the number of cells/nodes in each domain. Specifically, this number is 2 raised to the power of `InitExp` which is actually read in. Typical values used in each domain are of the order of $2^8 = 256$

A few variables are stored in the common block declared in `hssize.h`. These are `zeroh`, `zerov`, `zerohv`, and `zeror`.

- **zeroh**: Numerical zero in comparisons of specific volumns
- **zerov**: Numerical zero in comparisons of velocities
- **zeror**: Not used anywhere else in the code

Miscellaneous internal variables include

- **nwtnh**: Used internally in setting **tolh**, tolerance for the newton iterations for the specific volumn grid
- **nwtnv**: Used internally in setting **tolv**, tolerance for the newton iterations for the velocity grid
- **nwtns**: Used internally in setting **tolS**, tolerance for the newton iterations for the Lagrangian interface locations
- **nwtmfh**: Used internally in setting **tolfh**
- **nwtmfv**: Used internally in setting **tolfv**
- **nwtmfs**: Used internally in setting **tolfs**
- **cfl**: A CFL timestep number
- **zeroh**: Numerical zero in comparisons of specific volumns
- **zerov**: Numerical zero in comparisons of velocities
- **zeror**: Unknown and not used

`exp_pars.f`

This reads in the experimental parameters for each numerical experiment one desires to perform.

`setMgrid.f`

This subroutine defines and fills the computation grid.

Input variables include

- **NMAX0**: `0:NMAX0` are the linear dimension of each spatial grid
- **d**: an integer specifying the domain, should be 1 or 2
- **grL**: an array specifying the physical left endpoint of each domain, should always be the value 0.0
- **grR**: an array specifying the physical right endpoint of each domain, should always be the value 1.0

- `grDX`: an array specifying the physical node spacing in each domain defined as

$$\text{grDX}(d) = (\text{grR}(d) - \text{grL}(d))/\text{SizeInitGrid0}(d) \quad (34)$$

Output variables include

- `k`: an array specifying the number of nodes in this domain.
- `x`: an array holding the spatial locations of each node unknowns are from
- `dx`:
- `dxa`:
- `xc`:

`boundary.f`

This file stores many subroutines used in implementing the boundary conditions. Both the matrix generation, matrix decomposition, and the matrix solution for the components that involve the boundary are performed in this file.

References

- [1] *Los Alamos Shock Wave Profile Data*. Univ. of California Press, 1983.
- [2] O. P. Bruno and D. D. Vaynblat. Shock-induced martensitic phase transitions: critical stresses, Riemann problems and applications. *Proceedings of the Royal Soc. of London. Series A. Mathematical and Engineering Sciences*, 457(2016):2871–2920, 2001.
- [3] James M. Hill. *One-dimensional Stefan Problems: an Introduction*. John Wiley & Sons, Inc., New York, 1987.
- [4] R. D. Richtmyer. *Difference methods for initial-value problems*. Interscience, New York, 1967.
- [5] J. L. Weatherwax. *Mathematical Modeling of Shock Induced Martensitic Phase Transitions*. PhD dissertation, MIT, Department of Mathematics, Sept 2001.
- [6] J. L. Weatherwax, D. D. Vaynblat, O. P. Bruno, and R. R. Rosales. An investigation of proposed higher order effects in a model for martensitic phase transformations. In preparation.