Spring 1-1-2015

Implementing a Loosely Coupled Fluid Structure Interaction Finite Element Model in PHASTA

David Pope
University of Colorado at Boulder, david.e.pope@colorado.edu

Implementing a Loosely Coupled Fluid Structure Interaction Finite Element Model in PHASTA

David Pope

B.S., University of Colorado, 2006

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment of the requirement for the degree of
Masters of Science Department of Aerospace Engineering Sciences

This thesis entitled: Implementing a Loosely Coupled Fluid Structure Interaction Finite Element Model in PHASTA written by David Pope has been approved for the Department of Aerospace Engineering Sciences

Kenneth Jansen

John A. Evans

Eric Frew

Date

The final copy of this thesis has been examined by the signatories, and we Find that both the content and the form meet acceptable presentation standards Of scholarly work in the above mentioned discipline.

Abstract

Pope, David (M.S., Department of Aerospace Engineering Sciences)

Implementing a Loosely Coupled Fluid Structure Interaction Finite Element Model in PHASTA

Thesis directed by Professor Kenneth Jansen

Fluid Structure Interaction problems are an important multi-physics phenomenon in the design of aerospace vehicles and other engineering applications. A variety of computational fluid dynamics solvers capable of resolving the fluid dynamics exist. PHASTA is one such computational fluid dynamics solver. Enhancing the capability of PHASTA to resolve Fluid-Structure Interaction first requires implementing a structural dynamics solver. The implementation also requires a correction of the mesh used to solve the fluid equations to account for the deformation of the structure. This results in mesh motion and causes the need for an Arbitrary Lagrangian-Eulerian modification to the fluid dynamics equations currently implemented in PHASTA. With the implementation of both structural dynamics physics, mesh correction, and the Arbitrary Lagrangian-Eulerian modification of the fluid dynamics equations, PHASTA is made capable of solving Fluid-Structure Interaction problems.

List of Figures

Contents

1 Introduction

The dynamics of a structural system in the presence of a fluid exerting forces on the structural body is an important multi-physics modeling problem. As the structure deforms, changes in the flow of the fluid around the structural body occur, causing a change in the forces exerted on the structure. This non-linear relation to the structural deformation caused by aerodynamic forces is relevant across a variety of engineering fields. In civil engineering, the forces due to winds on large structures such as skyscrapers or suspension bridges should be accounted for to ensure that oscillations of the structure due to strong winds stay within structural tolerances. In aerospace engineering, the problem is especially relevant, particularly as it concerns flutter on lifting surfaces. To address these structural forces and possible divergent dynamic modes, a coupled system is needed that accounts for the changing forces exerted on the structure.

The modeling of both fluid flows and structural systems on computer systems is commonly done using Finite Element Methods. Other methods are available, such as Finite Difference Methods, but this paper focuses solely on the Finite Element Method, and specifically on the implementation of fluid structure interactions in the high fidelity fluid dynamics solver PHASTA.

Multiple methods for formulating a fluid structure system have been considered using the finite element method. In a fully coupled system, the fluid and structural equations of motion are formulated as one system of equations[1]. Another alternative for resolving fluid structure interaction is to account for fluid loading in a non-linear finite element model[2].

However, this thesis exams modeling fluid structure interaction by coupling independent structural and fluid finite element solvers. A simple linear dynamic finite element solver is coupled with PHASTA, a complex finite element fluid solver. The structural displacements of the structure interact with the fluid solver by deforming the mesh on the boundaries of the fluid domain, while the fluid solver interacts with the structural solver by passing fluid forces exerted on the structural domain to the structural solver.

This thesis will examine the utility of such a fluid-structure interaction model, specifically the data structures necessary for implementation of the loosely coupled system in PHASTA.

2 Formulation of Equations

2.1 Fluid Equations

PHASTA uses the Navier-Stokes equations to resolve fluid flow. The Navier-Stokes equations are the standard equations used to express fluid dynamics. The Navier-Stokes equations are conservation of momentum, conservation of energy and continuity. The continuity equation is described as[3]:

$$\rho_{,t} + \sum_{i=1}^{3} (\rho u_i)_{,i} = 0 \tag{1}$$

The conservation of momentum equations can be express as:

$$[\rho U_j]_{,t} + \sum_{i=1}^{3} [\rho u_i u_j]_{,i} + P_{,j} = \sum_{i=1}^{3} \tau_{ij,i} + b_j \tag{2}$$

Finally, the conservation of energy equations can be described as:

$$[\rho e_{tot}]_t + \sum_{i=1}^{3} [\rho u_i e_{tot}]_{,i} + \sum_{i=1}^{3} [u_i P]_{,i} = \sum_{i=1}^{3} \sum_{j=1}^{3} [\tau_{ij} u_j]_{,i} - \sum_{i=1}^{3} q_{ii} + \sum_{j=1}^{3} b_j u_j + \gamma \tag{3}$$

These equations can be express in a compact notation by introducing a new variable vector, the flux vector, and a body force vector. The new variable vector is:

$$\underline{U} = \begin{bmatrix} \rho \\ \rho u_j \\ \rho e_{tot} \end{bmatrix} \tag{4}$$

The flux vector is:

$$\underline{F}_i = \begin{bmatrix} \rho U_i \\ \rho U_i U_j \\ \rho U_i e_{tot} \end{bmatrix} + \begin{bmatrix} 0 \\ P\delta_{ij} \\ PU_i \end{bmatrix} - \begin{bmatrix} 0 \\ \tau_{ij} \\ \tau + ijU_j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ q_i \end{bmatrix} \tag{5}$$

The body force vector is:

$$\underline{\varphi} = \begin{bmatrix} 0 \\ b_j \\ b_j U_j + r \end{bmatrix} \tag{6}$$

With these new variables, the Navier-Stokes equations can be expressed as:

$$\underline{U}_{,t} + \underline{F}_{i,i} = \underline{\varphi} \tag{7}$$

It is convenient to express the flux vector in terms of advective and diffusive flux. Thus, we express the total flux vector in terms of two new vectors.

$$F_i^{adv} = u_i \underline{U} + P \begin{bmatrix} 0 \\ \delta_{ij} \\ u_i \end{bmatrix} \tag{8}$$

$$F_i^{diff} = -\begin{bmatrix} 0 \\ \tau_{ij} \\ u_i\tau_{ij} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ q_i \end{bmatrix} \tag{9}$$

With these two flux vectors expressed independently, the Navier-Stokes equations are written as

$$\underline{U}_{,t} + F_{i,i}^{adv} + F_{i,i}^{diff} = \underline{\varphi} \tag{10}$$

This expression of the Navier stokes equation must now be converted from five general partial differential equations to the finite element form. First, all terms are moved to the left hand side.

$$\underline{U}_{.t} + \underline{F}_{i,i} - \underline{\varphi} = 0 \tag{11}$$

Now the equations are multiplied by a weight function W. This can be considered an arbitrary function at this stage. This equation is then integrated over the entire fluid domain.

$$\int_\Omega \underline{W}[\underline{U}_{,t} + \underline{F}_{i,i} - \underline{\varphi}] = 0 \tag{12}$$

Distributing the weight function and using the product rule leads to the weak Galerkin formulation of the Navier-Stokes equations.

$$\int_\Omega \underline{W} \cdot \underline{U}_{,t} d\Omega - \int_\Omega \underline{W}_{,i} \cdot \underline{F}_i d\Omega - \int_\Omega \underline{W} \cdot \underline{\varphi} d\Omega + \int_\Gamma \underline{W} \cdot \underline{F}_i n_i d\Gamma = 0 \tag{13}$$

Using the finite element method, the values of the solution are sought at discrete nodal points. The values can be interpolated to any arbitrary location with a function of space. The most common functions used are element shape functions[4] Using these shape functions, the interpolated values for $\underline{W}$ and the derivatives of $\underline{U}$ at any location can be rewritten in terms of the solution vector U.

$$\underline{W}(x) = \sum_{B=1}^{n} N_B(x)\underline{W}_B \tag{14}$$

$$\underline{U}_{,t} = \sum_{A=1}^{n} N_A U_{,t} \tag{15}$$

$$\underline{U}_{,x} = \sum_{A=1}^{n} N_{A,x}(x) U_A \tag{16}$$

Then the weak Galerkin form of the Navier-Stokes equations can be rewritten.

$$\sum_{B=1}^{n} \underline{W}_B \int_\Omega N_B[\sum_{A=1}^{n} N_A U_{A,t} - \varphi(U) - N_{B,i}\underline{F}_i(U)]d\Omega + \int_\Gamma N_B \underline{F}_i(U) n_i d\Gamma = 0 = \sum_{B=1}^{n} \underline{W}_B \underline{G}_B \tag{17}$$

Where $\underline{G}_B$ is the residual.
This residual is formed at the element level according to

$$\underline{G}_b^e = \int_{\Omega_e}[N_b\sum_{a=1}^{nen} N_a U^e_{a,t} - \underline{\varphi}(\sum_{a=1}^{nen} N_a U^e_a) - N_{b,i}\underline{F}_i(\sum_{a=1}^{nen} N_a U^e_a)]d\Omega_e + \int_{\Gamma_e} N_b \underline{F}_i(\sum_{a=1}^{nen} N_a U^e_a) n_i d\Gamma_e \tag{18}$$

The local element residual is then stabilized according to the Streamline Upwind Petrov-Galerkin (SUPG) method, and a stabilized element residual $\hat{\underline{G}}_b^e$ is formed.

$$\hat{\underline{G}}_b^e = \underline{G}_b^e + \int_{el} \hat{\mathcal{L}} N_b(\xi)\underline{\tau}[\mathcal{L}\underline{U} - \varphi(\sum_{a=1}^{nen} N_a U^e_a)]D(\xi)d(el) \tag{19}$$

Where the differential operators $\hat{\mathcal{L}}$ and $\mathcal{L}$ are defined:

$$\hat{\mathcal{L}} N_b = \underline{A}_i(\sum_{a=1}^{nen} N_a(\xi)\underline{U}^e_a)N_{b,\xi_e}\xi_{l,i} \tag{20}$$

$$\mathcal{L}\underline{U} = \sum_{a=1}^{nen} N_a \underline{U}^e_{a,t} + \underline{A}_i(\sum_{c=1}^{nen} N_c \underline{U}^e_c)\sum_{a=1}^{nen} N_{a,\xi_e}\xi_{l,i}\underline{U}^e_a \tag{21}$$

The variable vector U can be converted to an arbitrary variable vector Y. This requires a series of transformations to change the dependance of the advective flux, diffusive flux and body transformation terms from dependent on U to dependent on Y. These transformations are summarized as:

$$\underline{U}_{,t} = \underline{U}_{,Y} Y_{,t} = A_0 Y_{,t} \tag{22}$$

$$F^{adv}_{i,i} = F^{adv}_{i,Y} Y_{,i} = A_i Y, i \tag{23}$$

$$F^{diff}_{i,i} = -K_{ij} Y_{,j} \tag{24}$$

With these variable transformations, the differential operator $\mathcal{L}$ can be expressed as:

$$\mathcal{L}\underline{U} = \mathcal{L}Y = [A_0 \frac{\partial}{\partial t} + A_i \frac{\partial}{\partial x_i} - \frac{\partial}{\partial x_j}[-K_{ij}\frac{\partial}{\partial x_j}]]Y \tag{25}$$

The variable A_0 and A_i in (22) and (23) are transformation matrices necessary for the transformation of the Navier-Stokes equations from the variable set U to the variable set Y and should not be mistaken for the $\underline{A}_i$ matrices in (20) and (21) which are part the linearization operator.

The element residual can be expressed in terms of variable Y according to:

$$\underline{\hat{G}}_b^e = \int_{el} N_b[A_0 Y_{,t} - \varphi - N_{b,i} F_i] d(el) + \int_{el} \hat{\mathcal{L}} N_b \tau (\mathcal{L} Y - \varphi) d(el) + \int_{el} N_b F_i n_i D_\Gamma d(el)_\Gamma \quad (26)$$

The global residual vector $\underline{\hat{G}}_B$ is assembled from the element residual vectors. The system of equations is then integrated in time using the generalized alpha method, a predictor multi-corrector algorithm. First, the solution vector at the next time step is predicted according to[3]:

$$Y^{n+1} = Y^n + \Delta t Y_{,t}^n + \Delta t \gamma (Y_{,t}^{n+1} - Y_{,t}^n) \quad (27)$$

While requiring that:

$$\underline{\hat{G}}_B (Y^{n+\alpha f}, Y_{,t}^{n+\alpha m}) = 0 \quad (28)$$

The values of Y at intermediate time steps αm and αf are found according to:

$$Y_{,t}^{n+\alpha m} = Y_{,t}^n + \alpha m [Y_{,t}^{n+1} - Y_{,t}^n] \quad (29)$$

$$Y^{n+\alpha f} = Y^n + \alpha f [Y^{n+1} - Y^n] \quad (30)$$

So from the initial guess for the values of the solution vector Y, the solution is corrected iteratively using Newtons method, where the change in solution vector is found by satisfying:

$$\underline{\hat{G}}_B (Y_{,t}^{n+\alpha m}(i), Y^{n+\alpha f}(i)) + \sum_{a=1}^{n} \frac{\partial \underline{\hat{G}}}{\partial Y^{n+\alpha f}(i)}) \Delta Y(i) = 0 \quad (31)$$

Introducing a new symbol for the tangent matrix of the residual in terms of the solution vector and the residual, the solution for each iteration can be described in a matrix equation of the form AX=b.

$$\frac{\partial \underline{\hat{G}}}{\Delta Y^{n+\alpha f}(i)} = M_{AB} \quad (32)$$

$$\underline{\hat{G}}_B (Y_{,t}^{n+\alpha m}(i), Y^{n+\alpha f}(i)) = R_B \quad (33)$$

$$\sum_{A=1}^{n} M_{BA}\Delta Y = -R_B \tag{34}$$

$$Y(i+1) = Y(i) + \Delta Y(i) \tag{35}$$

In PHASTA, this is solved using the Generalized Modified Residual method (GMRES)[10] for each iteration, and iterated until the residual is within a desired tolerance for each time step.

2.2 Arbitrary Lagrangian Eulerian Fluid Modification

In fluid-structure interaction problems, the structural mesh will deform. Because fine boundary layer meshes are desired to resolve the fluid flow near the border of the structure, there is a risk of inverting fluid elements if the mesh is left in its original state. For this reason, the fluid mesh is corrected at each time step so that it correctly adheres to the structure boundary, and the fluid elements do not become inverted. This introduces a velocity into the fluid mesh, and means that the Eulerian formulation will no longer be accurate. With the Eulerian formulation as a baseline, the mesh velocity can be accounted for by adapting to an Arbitrary Lagrangian-Eulerian (ALE) formulation. Starting with the Navier-Stokes equations, where $\dot{x}$ represents the mesh velocity, the equations of continuity, conservation of momentum and conservation of energy are changed to account for the changing mesh velocity.

$$\rho_{,t} + (\rho[u_i - \dot{x}_i])_{,i} = 0 \tag{36}$$

$$[\rho u_j]_{,t} + [\rho(u_i - \dot{x}_i)u_j]_{,i} + P_{,j} = \tau_{ij,i} + b_j \tag{37}$$

$$[\rho e_{tot}]_t + [\rho(u_i - \dot{x}_i)e_{tot}]_{,i} + [(u_i - \dot{x}_i)P]_{,i} = [\tau_{ij}u_j]_{,i} - q_{ii} + b_j u_j + \gamma \tag{38}$$

This means that the advective flux portion of the equations is modified

$$F_i^{adv} = (u_i - \dot{x}_i)\underline{U} + P\begin{bmatrix} 0 \\ \delta_{ij} \\ (u_i - \dot{x}_i) \end{bmatrix} \tag{39}$$

The addition of these terms are reflected in the transformation matrices A_i that relate the advective flux vector to spatial derivative of the solution vector Y_i. For a complete modification of all of these transformation matrices in Arbitrary Lagrangian-Eulerian formulation see Appendix B.

With these modifications, the Eulerian formulation is converted to Arbitrary Lagrangian-Eulerian and can account for mesh velocity in the solution.

2.3 Structural Dynamics Equations

The weak form of the structural dynamics equation is[4]:

$$(w, \rho)(w, \ddot{u}) + a(w, u) = (w, f) + (w, h)_\Gamma \tag{40}$$

where u represents displacement, a represents the strain energy inner product, f are the prescribed body forces and h the prescribed Neumann boundary forces, and w is any arbitrary weight function. More commonly , the discrete equations of motion are given as opposed to the variational form. In structural dynamics, the equations of motion for the structural system are often expressed in the form

$$M\ddot{d} + C\dot{d} + Kd = F \tag{41}$$

In these equations, M represents the mass, C represents damping, and K represents internal energy transferred into the structure due to deformation. This statement is equivalent to Newtons law of motion. In the FEM formulation, d is a vector of nodal displacements of the system, F is a vector of the forces acting on the nodes, while M is the mass matrix, C is the damping matrix, and K is the stiffness matrix of the structure.

These matrices are calculated at the element level and assembled to create the global mass, damping, and stiffness matrices for the system. At the element level, the mass matrix is found

$$m^e_{ab} = \int_{\Omega^e} N_a \rho N_b \tag{42}$$

This is found through numerical integration of the element according to

$$m^e_{ij} = \sum_{i=1}^{n_{gauss}} \sum_{j=1}^{n_{gauss}} W det(J) N_i \rho N_j \tag{43}$$

This particular formation of the mass matrix is called the consistent mass matrix. Other formulations such as the lumped mass matrix are also possible[4].

The discrete form of the stiffness matrix can be derived from the strain energy in the weak form. The strain energy can be expressed in terms of the elastic coefficients of the material.

$$a(w, u) = w_{(i,j)} c_{ijkl} u_{(k,l)} \tag{44}$$

This can be expressed equivalently in terms of the infinitesimal strain tensor and material elastic moduli matrix as

$$a(w, u) = \int_\Omega \epsilon(w)^T D \epsilon(u) d\Omega \tag{45}$$

This infinitesimal strain tensor can be expressed in terms of element shape function derivatives such that

$$\epsilon(N_A e_i) = B_A e_i \tag{46}$$

where e_i represents the ith column vector of the identity matrix. B is a matrix of the element shape function spatial derivatives

$$B_A = \begin{bmatrix} N_{A,1} & 0 & 0 \\ 0 & N_{A,2} & 0 \\ 0 & 0 & N_{A,3} \\ 0 & N_{A,3} & N_{A,2} \\ N_{A,3} & 0 & N_{A,1} \\ N_{A,2} & N_{A,1} & 0 \end{bmatrix} \tag{47}$$

The stiffness is the integral of the strain energy inner across the domain, found at the global equation number PQ as

$$K_{PQ} = e_i^T \int_\Omega B_A^T D B_B d\Omega e_j \tag{48}$$

This can be calculated at the element level as

$$k^e = \int_{\Omega^e} B^T D B d\Omega^e \tag{49}$$

This is most commonly done with numerical integration using Gaussian quadrature so that

$$k^e = \sum_{i=1}^{n_{gauss}} det(J) W B^T D B \tag{50}$$

Performing this integration numerically requires calculating the B matrices for each shape local equation number

$$k^e_{ab} = \sum_{i=1}^{n_{gauss}} det(J) W B_a^T D B_b \tag{51}$$

This can be expressed alternatively in terms of the Lamé constants so that

$$k^e_{PQ} = k^e_{iajb} = \sum_{i=1}^{n_{gauss}} det(J) W N_{a,k} c_{ijkl} N_{b,l} \tag{52}$$

After calculating the stiffness matrix for each individual element, the global stiffness matrix for the system is assembled. This is referred to as the direct stiffness method.

The damping matrix C is assembled as a linear combination of the mass and stiffness. This is called the Rayleigh damping matrix[5].

$$C = aM + bK \tag{53}$$

The coefficients b and a are chosen to give the desired damping characteristics to the system. Commonly the damping coefficients are based on the damping ratio desired at the natural frequency of the modes the system.

$$\zeta_n = a + b\omega_n \tag{54}$$

This means that the desired damping ratio at at least two modes of the system must be known to solve for the unknowns a and b. This also requires knowledge of the natural frequencies of the system at these modes. Generally, the lowest frequency mode and highest frequency mode of interest are used[5]. To determine the natural frequencies of the system eigenvalue analysis can be performed. However, determining eigenvalues of the entire system can be computationally expensive. An approximation is made for this solver that the lowest and highest eigenvalues in any element correspond to the lowest and highest eigenvalues of the system. The natural frequency of free vibration, ω, is found from the corresponding eigenvalue, p, according to[6]

$$p_i^2 = \omega_i^2 \tag{55}$$

With the formation of K, C and M the equations of motion of the system have been formed. The system must be constrained using boundary conditions to complete the definition of the problem. The zero displacement boundary condition is the only boundary condition allowed for the structure in the solver implemented. The method used is a reduced system, or reaction recovery system. This means that the global equations corresponding to zero displacement are reduced to zero in the system, and one on the diagonal. This means the contribution from these nodes becomes a trivial equation. A small example is demonstrated for a three by three system. We assume that the displacement u_3 is a known zero displacement boundary condition. The full system is shown first, and the reduced system is shown below it.

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} \ddot{u_1} \\ \ddot{u_2} \\ \ddot{u_3} \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{bmatrix} \dot{u_1} \\ \dot{u_2} \\ \dot{u_3} \end{bmatrix} + \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{56}$$

$$\begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \ddot{u_1} \\ \ddot{u_2} \\ 0 \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} & 0 \\ c_{21} & c_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{u_1} \\ \dot{u_2} \\ 0 \end{bmatrix} + \begin{bmatrix} k_{11} & k_{12} & 0 \\ k_{21} & k_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ 0 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ 0 \end{bmatrix} \tag{57}$$

With the formulation of the system complete, a time integration method must be selected to determine the solution of the problem. The generalized alpha method can be used to determine the solution of the problem[7].

The equations of motion in the discrete form are used to form a residual equation.

$$M\ddot{d} + C\dot{d} + Kd - F = R_s \tag{58}$$

With known initial conditions, the solution at the next time step is predicted with a constant velocity assumption.

$$v_{n+1} = v_n \tag{59}$$

$$a_{n+1} = \frac{\gamma - 1}{\gamma} a_n \tag{60}$$

$$d_{n+1} = d_n + \Delta t v_n + \frac{\Delta t^2}{2}((1-2\beta)a_n + 2\beta a_{n+1}) \tag{61}$$

The values of the variables and the residual are then found at intermediate time steps.

$$d_{n+\alpha_f} = d_n + \alpha_f(d_{n+1} - d_n) \tag{62}$$

$$v_{n+\alpha_f} = v_n + \alpha_f(v_{n+1} - v_n) \tag{63}$$

$$a_{n+\alpha_m} = a_{n+1} + \alpha_m(a_{n+1} - a_n) \tag{64}$$

$$R_{n+1} = R(d_{n+\alpha_f}, v_{n+\alpha_f}, a_{n+\alpha_m}) \tag{65}$$

This is iterated to improve the values for the solution according to:

$$\frac{\partial R}{\partial a_{n+1}} \Delta a = -R_{n+1} \tag{66}$$

The residual of the partial derivative of the residual of the time step in terms of the acceleration is found according to:

$$\frac{\partial R}{\partial a_{n+1}} = \alpha_m M + \alpha_f \Delta t C + \alpha_f \beta(\Delta t^2)K \tag{67}$$

Then the incremental correction Δa is solved for using the GMRES method. This correction is used to update the variables at the next time step according to:

$$a_{n+1}^{i+1} = a_{n+1}^{i} + \Delta a \tag{68}$$

$$v_{n+1}^{i+1} = v_{n+1}^{i} + \gamma \Delta t \Delta a \tag{69}$$

$$d_{n+1}^{i+1} = d_{n+1}^{i} + \beta(\Delta t^2)\Delta a \tag{70}$$

This iterative process is repeated until the norm of the residual of the system is reduced below a desired value.

2.4 Mesh Correction

A variety of strategies exist to correct the mesh when a prescribed deformation occurs. Some of these range in complexity from minimizing the change in the determinant of the Jacobian of each element to simply displacing the nodes with the same angular displacement equally across the mesh. The strategy adapted here is to treat the mesh correction as a prescribed displacement structural FEM problem. This problem then takes the form:

$$Ku = F \tag{71}$$

where K is a stiffness matrix, u is a vector of the mesh nodal displacements and F is a force vector. The applied forces where nodal displacements are unknown are initially set to zero. The simple three by three example for reduction will be used again to illustrate this process. Lets take u_1 as the known displacement from the structural deformation. The original system takes the form[8]:

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{72}$$

The system is partially reduced where the displacements are known to a trivial equation. The forces are modified to make the trivial equation reflect the prescribed displacements. For the example system, this is

$$\begin{bmatrix} 1 & 0 & 0 \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{73}$$

Next, the force modification due to prescribed displacements is found. This means that forces corresponding to locations where the displacement is unknown are modified due to the prescribed displacements.

$$\begin{bmatrix} 1 & 0 & 0 \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} (u_1) \\ (f_2 - k_{21}u_1) \\ (f_3 - k_{31}u_1) \end{bmatrix} \tag{74}$$

With force modification complete, the system is fully reduced. This means that columns corresponding to known displacements are zeroed out.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & k_{22} & k_{23} \\ 0 & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} (u_1) \\ (f_2 - k_{21}u_1) \\ (f_3 - k_{31}u_1) \end{bmatrix} \tag{75}$$

This leaves a linear equations for the unknown displacements in the mesh. This equation is then solved for using GMRES.

2.5 Loose Coupling Mechanism of Fluid-Structure Interaction

The structure equations and fluid equations are loosely coupled by interaction of their residuals. The fluid forces acting on the structural body are passed to the structure model as nodal forces along the fluid-structure boundary. The deformation of the structure is passed to the fluid model as nodal velocity data. The deformation of the structure thus alters the fluid residual, resulting in an updated fluid solution. Likewise, the updated fluid solution results in new forces acting on the structural body, changing the residual of the structural model, and resulting in new displacement data.

A solution within tolerance for each time step is found in a highly iterative process. The fluid solution is resolved first. The solution is iterated until it is within tolerance. The fluid forces on the structural body are then passed to the structure as nodal forces. The structure solution is then resolved. The deformation of the structure is used to deform the mesh. Nodal velocity data from the mesh is then passed back to the fluid model. The fluid is solved again with this nodal velocity data to generate the new resultant forces for the structure model. This process is iterated until the change in fluid forces and structural deformation between iterations is negligible. At this point, the solution for the time step has converged.

The aerodynamic forces are resolved in PHASTA using the consistent boundary flux calculation technique. This technique solves for the viscous and heat flux on a boundary where the discrete solution for primitive variables is known[9].

The consistent boundary flux calculation treats the flux vector, f, consisting of the boundary viscous flux $\tau_{ij}n_j$ and heat flux $-q_in_i$ as an unknown variable. In these equations, the subscript w refers to the surface on the boundary where we are interested in resolving the fluxes.

$$f_{flux} = \begin{bmatrix} 0 \\ \tau_{ij}n_j \\ -q_in_i \end{bmatrix} = F_wn_i \tag{76}$$

This variable is then solved for according to the weak Galerkin formulation as

$$\int_{\Gamma_w} \underline{W} f_{flux} d\Gamma + \int_{\Gamma-\Gamma_w} \underline{W} F_w(U) n_i d\Gamma = \int_\Omega \underline{W}\cdot\underline{U}_{,t} d\Omega - \int_\Omega \underline{W}_{,i}\cdot\underline{F}_i d\Omega - \int_\Omega \underline{W}\cdot\underline{\varphi} d\Omega + \int_\Gamma \underline{W}\cdot\underline{F}_i n_i d\Gamma \tag{77}$$

The heat flux terms are not necessary for the structure, but the coupled system can be approximated based on the consistent boundary flux calculation technique as a system where

$$M\ddot{d} + C\dot{d} + Kd = \tau_{ij} n_j \tag{78}$$

3 Computer Implementation of Equations

This section provides very specific detail at a variable level necessary for the implementation of the necessary subroutines for FSI in PHASTA. A more broad overview is provided in a hierarchy of subroutines for each section. The implementation required the creation of a new structural dynamics solver for PHASTA, ALE modification of existing PHASTA subroutines, creation of a new mesh corrector for PHASTA, and creation of interfaces for the fluid solver, structure solver, and mesh corrector.

This is not an exhaustive list of the functionality and subroutines contained in PHASTA, but a brief overview of the most important subroutines.

3.1 Fluid Equations

The solution of the Navier-Stokes equations in PHASTA occurs on on five main levels. At the highest level, the solver loops through time steps, acquires solutions, and updates solution values according to input. At the level below this, the solution is found using either the GMRES method or the Matrix Free Galerkin method as specified by input. Below this level, blocks of element data are assembled and placed in appropriate data structures. Below this, element blocks are broken down into individual elements. At the lowest level, individual elements are used for computation of the components of the residual, and the tangent matrix of the residual. A hierarchy diagram for important subroutines in PHASTA is shown in Figure 1. Subroutines added to PHASTA for the solution of structural dynamics are colored in green, while subroutines added for mesh correction are shown in yellow.

Five important subroutines correspond to the five main levels of the solution. At the highest level, *itrdrv*. The main subroutine at the solution level for a sparse GMRES solve is *solgmrs*. At the element block level, the subroutine *elmgmrs* is used to assemble data. Within element blocks, the subroutine *asigmr* is used to assemble element data to corresponding global equation numbers. At the element level, the subroutine *e3* is used to calculate the residual and tangent information for each individual element according to (19).

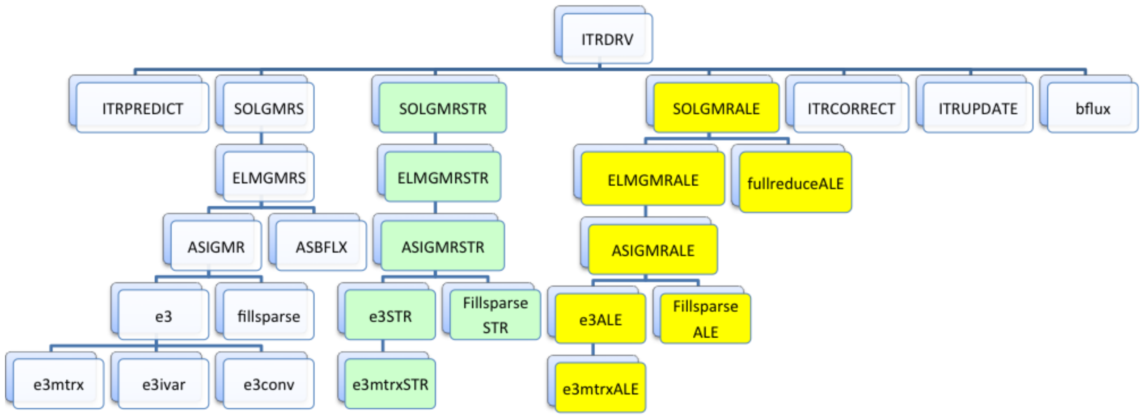


Figure 1: Hierarchy Diagram of Important PHASTA Subroutines

At the top level, *itrdrv* loops through time steps. For each time time step, it first predicts the solution at the next time step by calling the subroutine *itrpredict* as shown in Figure 3.

Within each time sequence, *itrdrv* follows the sequence of solves and updates specified by the user. *Itrdrv* loops through the sequence and performs a check to determine if a solve or update is occurring and which solver to use. The check for whether a solve or update is occurs as seen in Figure 4. In *itrdrv*, the variable Y is used to store the solution variables, corresponding to the Y vector in (22).

When the solution for the fluid using a sparse GMRES solver is desired, the subroutine *solgmrs* is called from *itrdrv* as seen in Figure 5. This assembles the global tangent of the residual as the variable lhsk, equivalent to M_{AB} in 34, and the a diagonalized version of the tangent matrix, BDiag, for the GMRES solve.

The flow solution variables at the end of the time step are contained in the vector y, while the solution variables at the beginning of the time step are contained in the vector

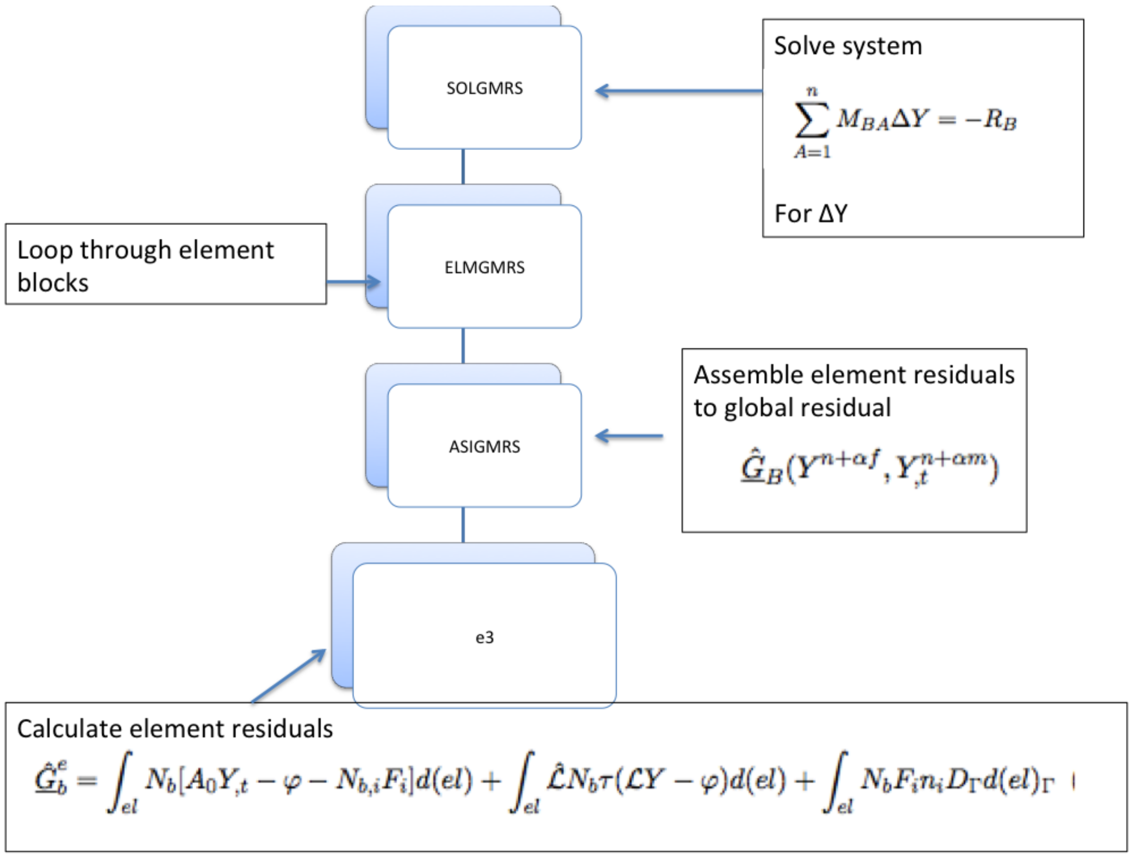
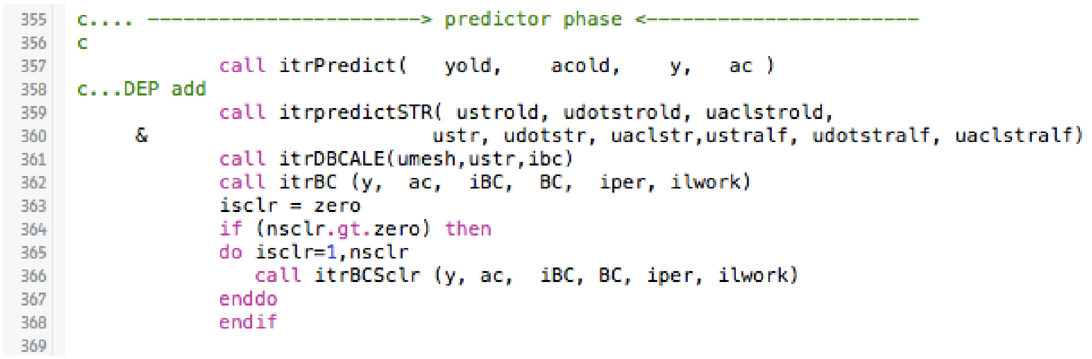

SOLGMRS
Solve system
$\sum_{A=1}^{n} M_{BA}\Delta Y = -R_B$
For ΔY
Loop through element blocks
ELMGMRS
Assemble element residuals to global residual
$\hat{\underline{G}}_B(Y^{n+\alpha f}, Y_{,t}^{n+\alpha m})$
ASIGMRS
e3
Calculate element residuals
$\hat{\underline{G}}_b^e = \int_{el} N_b[A_0 Y_{,t} - \varphi - N_{b,i}F_i]d(el) + \int_{el} \hat{\mathcal{L}} N_b \tau(\mathcal{L}Y - \varphi)d(el) + \int_{el} N_b F_i n_i D_\Gamma d(el)_\Gamma$

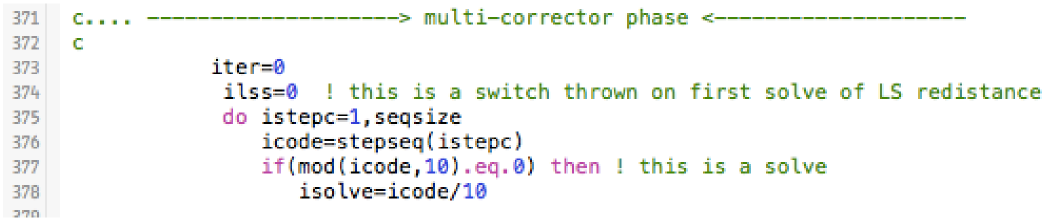
Figure 2: Solution Process for Fluid Dynamics



```
c.... -----------------------> predictor phase <-----------------------
c
            call itrPredict(   yold,    acold,    y,   ac )
c...DEP add
            call itrpredictSTR( ustrold, udotstrold, uaclstrold,
     &                      ustr, udotstr, uaclstr,ustralf, udotstralf, uaclstralf)
            call itrDBCALE(umesh,ustr,ibc)
            call itrBC (y,  ac,  iBC,  BC,  iper, ilwork)
            isclr = zero
            if (nsclr.gt.zero) then
            do isclr=1,nsclr
               call itrBCSclr (y, ac,  iBC, BC, iper, ilwork)
            enddo
            endif
```

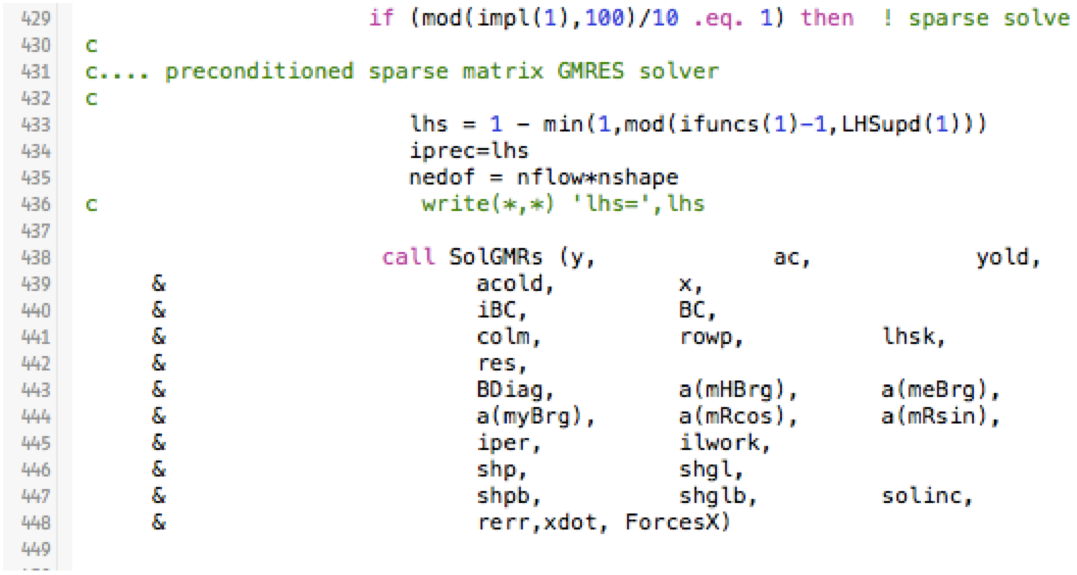
Figure 3: Predictor Phase in ITRDRV.f

yold. The nodal coordinates are contained in the vector x, and the vector ForcesX contains



```
c.... ---------------------> multi-corrector phase <---------------------
c
           iter=0
            ilss=0  ! this is a switch thrown on first solve of LS redistance
            do istepc=1,seqsize
               icode=stepseq(istepc)
               if(mod(icode,10).eq.0) then ! this is a solve
                  isolve=icode/10
```

Figure 4: Sequence Check in ITRDRV.f



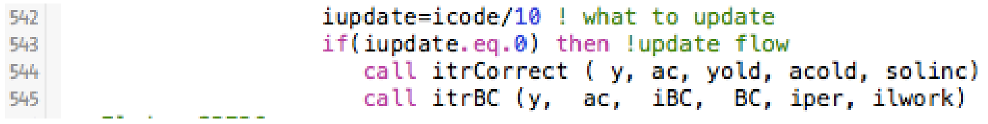
```
                     if (mod(impl(1),100)/10 .eq. 1) then  ! sparse solve
c
c.... preconditioned sparse matrix GMRES solver
c
                        lhs = 1 - min(1,mod(ifuncs(1)-1,LHSupd(1)))
                        iprec=lhs
                        nedof = nflow*nshape
c                        write(*,*) 'lhs=',lhs

                      call SolGMRs (y,             ac,             yold,
     &                       acold,          x,
     &                       iBC,            BC,
     &                       colm,           rowp,           lhsk,
     &                       res,
     &                       BDiag,          a(mHBrg),       a(meBrg),
     &                       a(myBrg),       a(mRcos),       a(mRsin),
     &                       iper,           ilwork,
     &                       shp,            shgl,
     &                       shpb,           shglb,          solinc,
     &                       rerr,xdot, ForcesX)
```

Figure 5: Sparse GMRES call in ITRDRV.f

the fluid forces to be passed to the structural solver. The tangent matrix is contained in the sparse data structure lhsk, while the residual is the vector res. Res is equivalent to the global residual in (28).

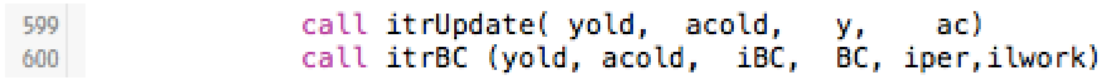
After the solution has been iterated the desired number of times, *itrdrv* calls *itrcorrect* to convert the variables from the solution at the intermediate time step to the variables at the end of the time step. Finally, the solution variables are placed in the yold vector for use as the starting values at the next time step. This process is detailed in (29) through (35). The function calls are shown in Figures 6 and 7.

These steps represent the solution process at the time and step sequence level. At the solution level, sparse data structures are used with the GMRES method within the subroutine solgmrs. Within this subroutine, the tangent stiffness matrix and residual are formed by calling elmgrs. The block diagonal of the tangent stiffness matrix is LU factorized and used to precondition the tangent stiffness and residual. GMRES is then applied to this



```
                    iupdate=icode/10 ! what to update
                    if(iupdate.eq.0) then !update flow
                       call itrCorrect ( y, ac, yold, acold, solinc)
                       call itrBC (y,  ac,  iBC,  BC, iper, ilwork)
```

Figure 6: ITRCORRECT call in ITRDRV.f



```
                 call itrUpdate( yold,  acold,   y,    ac)
                 call itrBC (yold, acold,  iBC,  BC, iper,ilwork)
```

Figure 7: ITRUPDATE call in ITRDRV.f

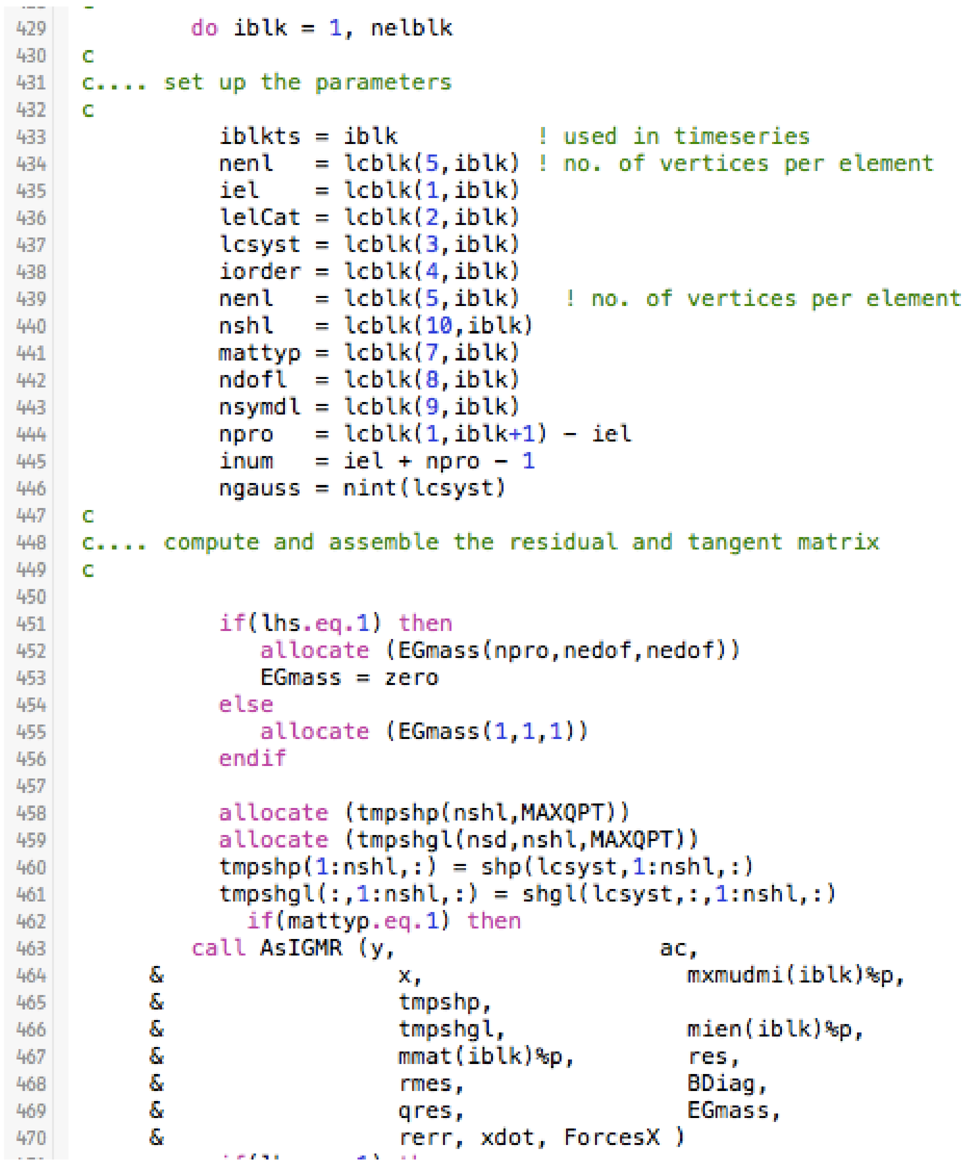
system using a set number of Krylov vectors to determine the correction to the solution variables.

Within *elmgrs* the tangent stiffness and residual terms for blocks of elements are assembled to the global terms. The global mesh is divided into blocks, each containing a set number of elements to give best performance on a given architecture. *Elmgrs* loops through these blocks and calls the subroutine *asigmr* to determine the tangent stiffness term for elements within the block, defined as the variable EGMass at the element level. The residual terms are also calculated. The loop through element blocks and call to *asigmr* is shown in Figure 8.

The element tangent stiffness EGMass is used to fill the sparse global tangent stiffness matrix lhsk using the subroutine *fillsparse*, with row and col as vectors used for the global equation number index of each term, as shown in Figure 9. The residual information, resl at the element level, is assembled to the global residual using the subroutine *local*. Resl is equivalent to the local element stiffness expressed in (26).

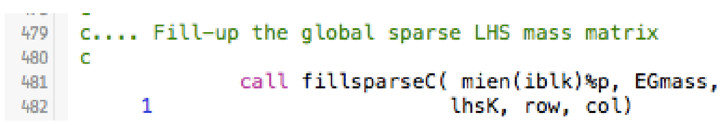
The subroutine *asigmr* localizes the mesh location data in x and the solution variables in y to the element level for calculation of element residual and tangent stiffness. Then the subroutine *e3* is called to calculate the residual and tangent stiffness for each element. The localization of variables and call to *e3* is shown in Figure 10. Then the localization process is reversed to assemble the residual, and the block diagonal of the tangent stiffness matrix is formed.

Within the subroutine *e3*, element level calculations are performed to find the residual and tangent stiffness matrix for each element. This is done in a loop over Gaussian quadrature points for the element. At each Gaussian quadrature point the integration variables, for example shape function derivatives, needed for the calculations are determined in *e3ivar*. The A matrices necessary to relate the advective flux term to the solution variable are calculated in *e3mtrx*. These two subroutines are called as shown in Figure 11. Further subroutines calculate the contribution to the residual and tangent stiffness from convective, mass, and least squares terms.



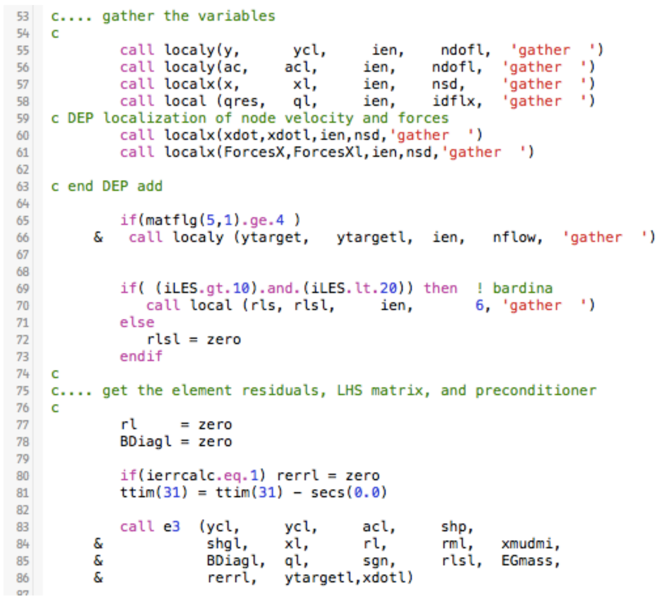
```
        do iblk = 1, nelblk
c
c.... set up the parameters
c
          iblkts = iblk          ! used in timeseries
          nenl   = lcblk(5,iblk) ! no. of vertices per element
          iel    = lcblk(1,iblk)
          lelCat = lcblk(2,iblk)
          lcsyst = lcblk(3,iblk)
          iorder = lcblk(4,iblk)
          nenl   = lcblk(5,iblk)   ! no. of vertices per element
          nshl   = lcblk(10,iblk)
          mattyp = lcblk(7,iblk)
          ndofl  = lcblk(8,iblk)
          nsymdl = lcblk(9,iblk)
          npro   = lcblk(1,iblk+1) - iel
          inum   = iel + npro - 1
          ngauss = nint(lcsyst)
c
c.... compute and assemble the residual and tangent matrix
c

          if(lhs.eq.1) then
             allocate (EGmass(npro,nedof,nedof))
             EGmass = zero
          else
             allocate (EGmass(1,1,1))
          endif

          allocate (tmpshp(nshl,MAXQPT))
          allocate (tmpshgl(nsd,nshl,MAXQPT))
          tmpshp(1:nshl,:) = shp(lcsyst,1:nshl,:)
          tmpshgl(:,1:nshl,:) = shgl(lcsyst,:,1:nshl,:)
            if(mattyp.eq.1) then
        call AsIGMR (y,                   ac,
     &                 x,                   mxmudmi(iblk)%p,
     &                 tmpshp,
     &                 tmpshgl,             mien(iblk)%p,
     &                 mmat(iblk)%p,        res,
     &                 rmes,                BDiag,
     &                 qres,                EGmass,
     &                 rerr, xdot, ForcesX )
```

Figure 8: Loop Through Element Blocks



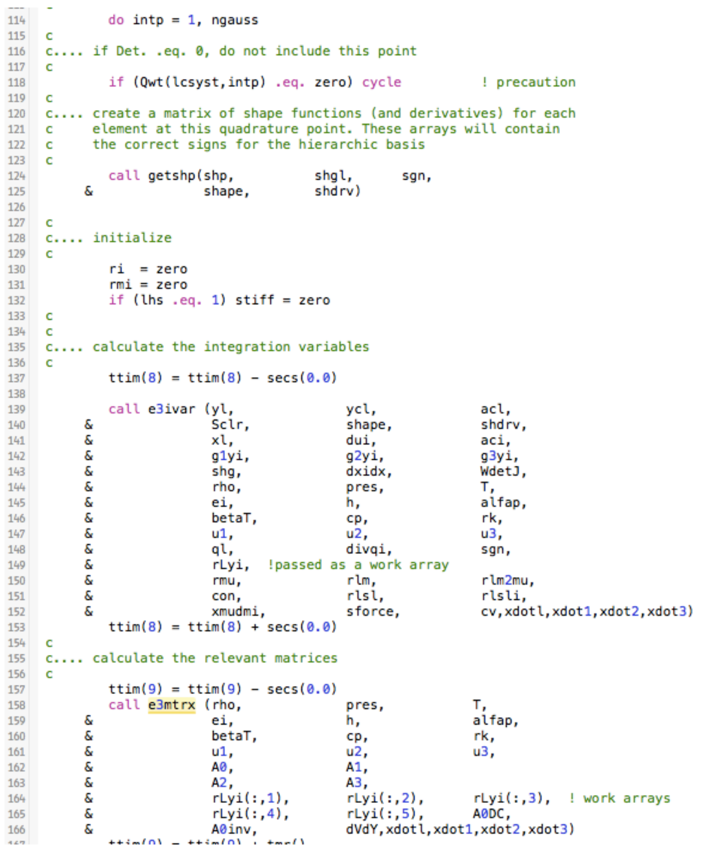
```
c.... Fill-up the global sparse LHS mass matrix
c
           call fillsparseC( mien(iblk)%p, EGmass,
     1                          lhsK, row, col)
```

Figure 9: Sparse Data Structure Fill



```
c.... gather the variables
c
        call localy(y,      ycl,     ien,    ndofl,  'gather  ')
        call localy(ac,    acl,     ien,    ndofl,  'gather  ')
        call localx(x,      xl,     ien,    nsd,    'gather  ')
        call local (qres,   ql,     ien,    idflx,  'gather  ')
c DEP localization of node velocity and forces
        call localx(xdot,xdotl,ien,nsd,'gather  ')
        call localx(ForcesX,ForcesXl,ien,nsd,'gather  ')

c end DEP add

        if(matflg(5,1).ge.4 )
     &  call localy (ytarget,   ytargetl,  ien,   nflow,  'gather  ')


        if( (iLES.gt.10).and.(iLES.lt.20)) then  ! bardina
           call local (rls, rlsl,     ien,       6, 'gather  ')
        else
           rlsl = zero
        endif
c
c.... get the element residuals, LHS matrix, and preconditioner
c
        rl     = zero
        BDiagl = zero

        if(ierrcalc.eq.1) rerrl = zero
        ttim(31) = ttim(31) - secs(0.0)

        call e3  (ycl,     ycl,     acl,     shp,
     &           shgl,    xl,      rl,      rml,   xmudmi,
     &           BDiagl,  ql,      sgn,     rlsl,  EGmass,
     &           rerrl,   ytargetl,xdotl)
```

Figure 10: Localization of variables and e3 call



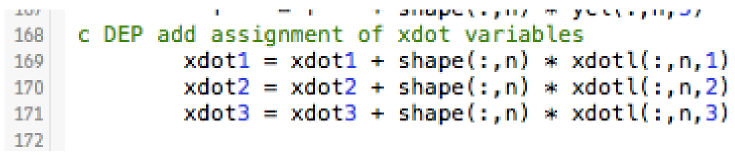
```
        do intp = 1, ngauss
c
c.... if Det. .eq. 0, do not include this point
c
        if (Qwt(lcsyst,intp) .eq. zero) cycle          ! precaution
c
c.... create a matrix of shape functions (and derivatives) for each
c     element at this quadrature point. These arrays will contain
c     the correct signs for the hierarchic basis
c
        call getshp(shp,          shgl,      sgn,
     &              shape,        shdrv)

c
c.... initialize
c
        ri  = zero
        rmi = zero
        if (lhs .eq. 1) stiff = zero
c
c
c.... calculate the integration variables
c
        ttim(8) = ttim(8) - secs(0.0)

        call e3ivar (yl,             ycl,             acl,
     &               Sclr,           shape,           shdrv,
     &               xl,             dui,             aci,
     &               g1yi,           g2yi,            g3yi,
     &               shg,            dxidx,           WdetJ,
     &               rho,            pres,            T,
     &               ei,             h,               alfap,
     &               betaT,          cp,              rk,
     &               u1,             u2,              u3,
     &               ql,             divqi,           sgn,
     &               rLyi,  !passed as a work array
     &               rmu,            rlm,             rlm2mu,
     &               con,            rlsl,            rlsli,
     &               xmudmi,         sforce,          cv,xdotl,xdot1,xdot2,xdot3)
        ttim(8) = ttim(8) + secs(0.0)
c
c.... calculate the relevant matrices
c
        ttim(9) = ttim(9) - secs(0.0)
        call e3mtrx (rho,            pres,           T,
     &               ei,             h,              alfap,
     &               betaT,          cp,             rk,
     &               u1,             u2,             u3,
     &               A0,             A1,
     &               A2,             A3,
     &               rLyi(:,1),      rLyi(:,2),      rLyi(:,3),  ! work arrays
     &               rLyi(:,4),      rLyi(:,5),      A0DC,
     &               A0inv,          dVdY,xdotl,xdot1,xdot2,xdot3)
```

Figure 11: Quadrature Point Loop in e3.f

3.2 ALE Modification of Fluid Equations

The fluid equations are modified to account for mesh velocity at the element level. The mesh velocity is found by dividing the deformation of the mesh by the time step after the mesh solver returns a value. These velocities are localized in the subroutine *asigmr*.

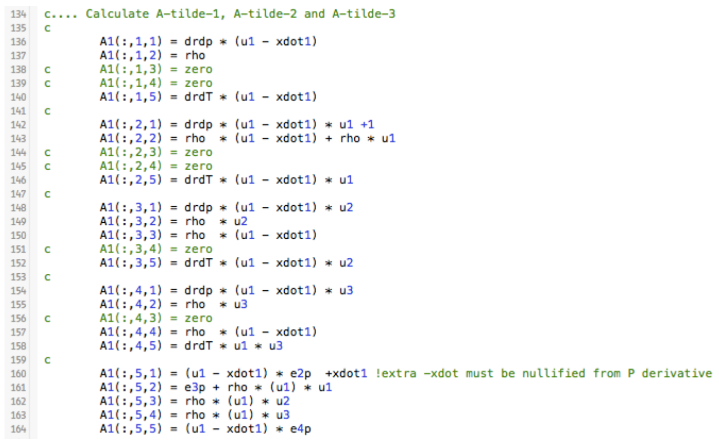
At the element level, mesh velocity is found at the quadrature point in the subroutine *e3ivar* with the statements seen in Figure 12. This change corresponds to the one detailed in (39) at the element level.



```
c DEP add assignment of xdot variables
         xdot1 = xdot1 + shape(:,n) * xdotl(:,n,1)
         xdot2 = xdot2 + shape(:,n) * xdotl(:,n,2)
         xdot3 = xdot3 + shape(:,n) * xdotl(:,n,3)
```

Figure 12: Mesh velocity at Quadrature Point Determination

These mesh velocities are then used to modify the A matrices in the subroutine *e3mtrx*. This accounts for the necessary change shown (39) being reflected in the transformation in (22) and (23). An example for the matrix A0 can be seen in Figure 13. For the complete modification, see Appendix B.



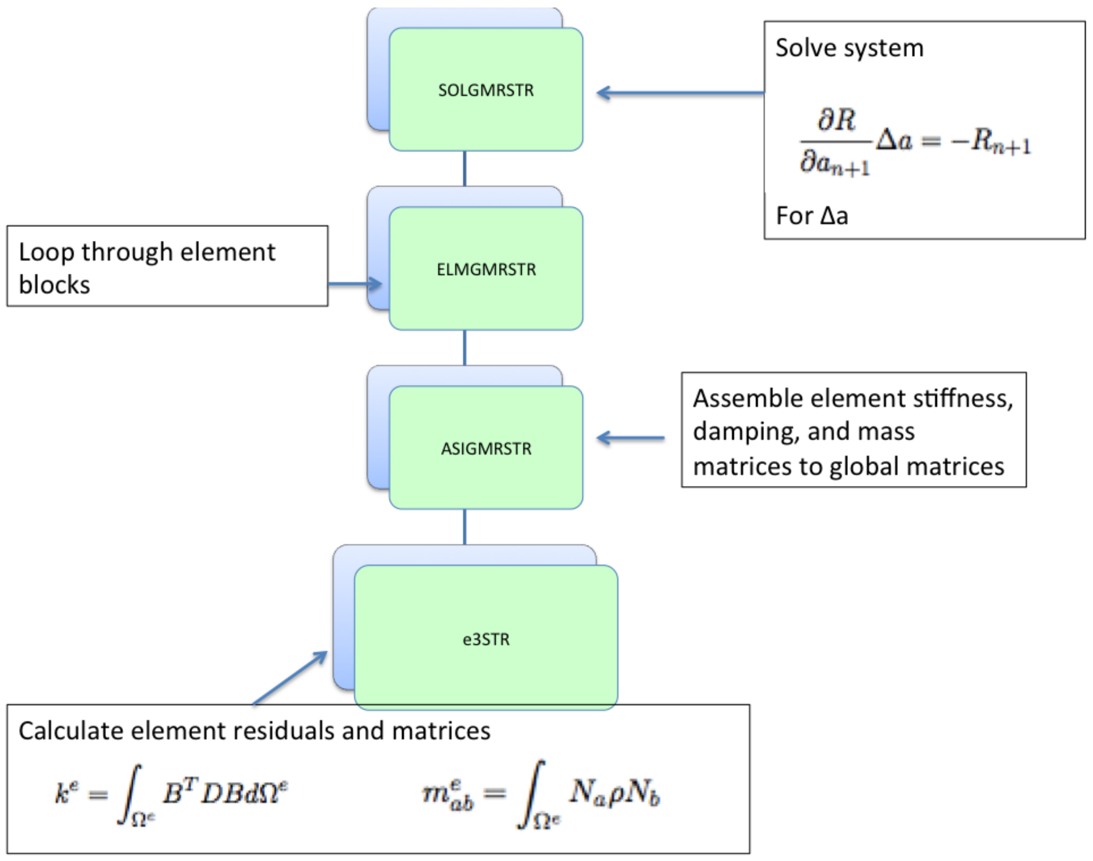
```
c.... Calculate A-tilde-1, A-tilde-2 and A-tilde-3
c
         A1(:,1,1) = drdp * (u1 - xdot1)
         A1(:,1,2) = rho
c        A1(:,1,3) = zero
c        A1(:,1,4) = zero
         A1(:,1,5) = drdT * (u1 - xdot1)
c
         A1(:,2,1) = drdp * (u1 - xdot1) * u1 +1
         A1(:,2,2) = rho  * (u1 - xdot1) + rho * u1
c        A1(:,2,3) = zero
c        A1(:,2,4) = zero
         A1(:,2,5) = drdT * (u1 - xdot1) * u1
c
         A1(:,3,1) = drdp * (u1 - xdot1) * u2
         A1(:,3,2) = rho  * u2
         A1(:,3,3) = rho  * (u1 - xdot1)
c        A1(:,3,4) = zero
         A1(:,3,5) = drdT * (u1 - xdot1) * u2
c
         A1(:,4,1) = drdp * (u1 - xdot1) * u3
         A1(:,4,2) = rho  * u3
c        A1(:,4,3) = zero
         A1(:,4,4) = rho  * (u1 - xdot1)
         A1(:,4,5) = drdT * u1 * u3
c
         A1(:,5,1) = (u1 - xdot1) * e2p  +xdot1 !extra -xdot must be nullified from P derivative
         A1(:,5,2) = e3p + rho * (u1) * u1
         A1(:,5,3) = rho * (u1) * u2
         A1(:,5,4) = rho * (u1) * u3
         A1(:,5,5) = (u1 - xdot1) * e4p
```

Figure 13: Modificiation of A1 for Mesh Velocity

3.3 Structural Equations

A hierarchy of the basic solution structure for the structural dynamics solver is shown in Figure 14.

Structural elements are determined by element type. This solver assumes all structural elements are linear hexahedra and that all linear hexahedra are structural elements. Ideally,



SOLGMRSTR
Solve system
$\frac{\partial R}{\partial a_{n+1}}\Delta a = -R_{n+1}$
For Δa
Loop through element blocks
ELMGMRSTR
Assemble element stiffness, damping, and mass matrices to global matrices
ASIGMRSTR
e3STR
Calculate element residuals and matrices
$k^e = \int_{\Omega^e} B^T D B d\Omega^e$
$m^e_{ab} = \int_{\Omega^e} N_a \rho N_b$

Figure 14: Solution Process for Structural Dynamics

this would be accomplished in the pre-processing phase. However, to keep work on the pre-processor from diverting work from PHASTA, the decision was made to use element type to determine the physics. The mesh data is separated into element blocks in the subroutine *genadj*. Within this subroutine, all linear hexahedra are flagged as structural elements by setting the variable material to 2 for the corresponding element block. All other elements are given the material flag 1 and assumed to be fluid elements. Nodal connectivity is stored in arrays ienf and iens for the fluid and structure respectively.

The structural equations are implemented in a manner that mirrors the fluid equations. The structural solver, the subroutine *solgmrstr*, is called from the subroutine *itrdrv*. This subroutine forms the global tangent stiffness matrix of the residual by calling the subroutine *elmgmrstr*. The residual is calculated and *solgmrstr* uses the GMRES method to find the appropriate change in the solution vector for the most accurate solution at the end of the time step. At the global level, the stiffness matrix K is stored in sparse format as the variable BigK, corresponding to K in (41). The consistent mass matrix is stored in sparse format as the variable BigM, corresponding to M in (41), and the Rayleigh damping matrix is stored as the variable BigC, corresponding to C in (41). The appropriate tangent stiffness matrix for the residual using the generalized alpha method is stored as BigMstar, and the block diagonal of this matrix is stored as BDiagSTR. BigMstar is the derivative of the residual expressed in (67). The solution variables consist of the nodal displacement, stored as the vector ustr, the nodal velocity, stored as the vector udotstr, and the nodal acceleration, stored as the vector uaclstr. The residual of the structural equations of motion is stored in the vector resSTR. The nodal forces generated by the fluid acting on the structure are passed into *solgmrstr* in the variable ForcesX.

The subroutine *elmgrstr* loops over element blocks. Within the loop for each element block, the subroutine *asimgrstr* is called to form the components of the tangent stiffness matrix and calculate their contribution to the global tangent stiffness matrix.

Within the subroutine *asigmrstr*, the subroutine *e3str* is called to form the local stiffness, constant mass and damping matrices for each element. The local matrices are then assembled to their global sparse counterparts using the subroutine *fillsparsestr*. The subroutine *fillsparsestr* also reduces the system of equations to account for zero displacement boundary conditions by setting the entries to zero on the off diagonal terms where zero displacement boundary conditions are present. The zero displacement boundary conditions are determined using the input variable scalar 2 in PHASTA. It is assumed to be a zero displacement boundary condition wherever this variable is set on a structural node.

The subroutine *e3str* forms the element stiffness matrix as the variable kl, the element damping matrix as the variable cl, and the element consistent mass matrix as the variable ml. This is done with a loop through element quadrature points as shown in (52). First, the shape function and shape function derivative values at the quadrature point are found using the subroutines *getshp* and *e3ivarstr*. The components of the tangent stiffness are formed in the subroutines *e3mtrxstr* and *e3mtrxstr2*. The stiffness matrix is formed assuming an isotropic homogenous material using the direct stiffness method. The consistent mass is formed through direct numerical integration of the element mass equation. The subroutine *e3mtrxstr2* is used for the portion of the computation of the element stiffness that occurs after the loop through quadrature points. This subroutine also assembles nodal values of the element mass and stiffness matrix.

Within each time step, the subroutine *itrcorrectstr* is used to update the values of the solution at the next time step and then the intermediate time step values of the solution for the nodal displacement, velocity, and acceleration. This is the same process summarized in (62), (63), (64), and (65). After the prescribed number of iterations per time step has been completed, these solutions are updated using the subroutine *itrupdatestr*.

3.4 Mesh Correction

A hierarchy of the basic solution structure for the mesh correction is shown in Figure 15.

The mesh correction takes the form a prescribed displacement structural problem. The

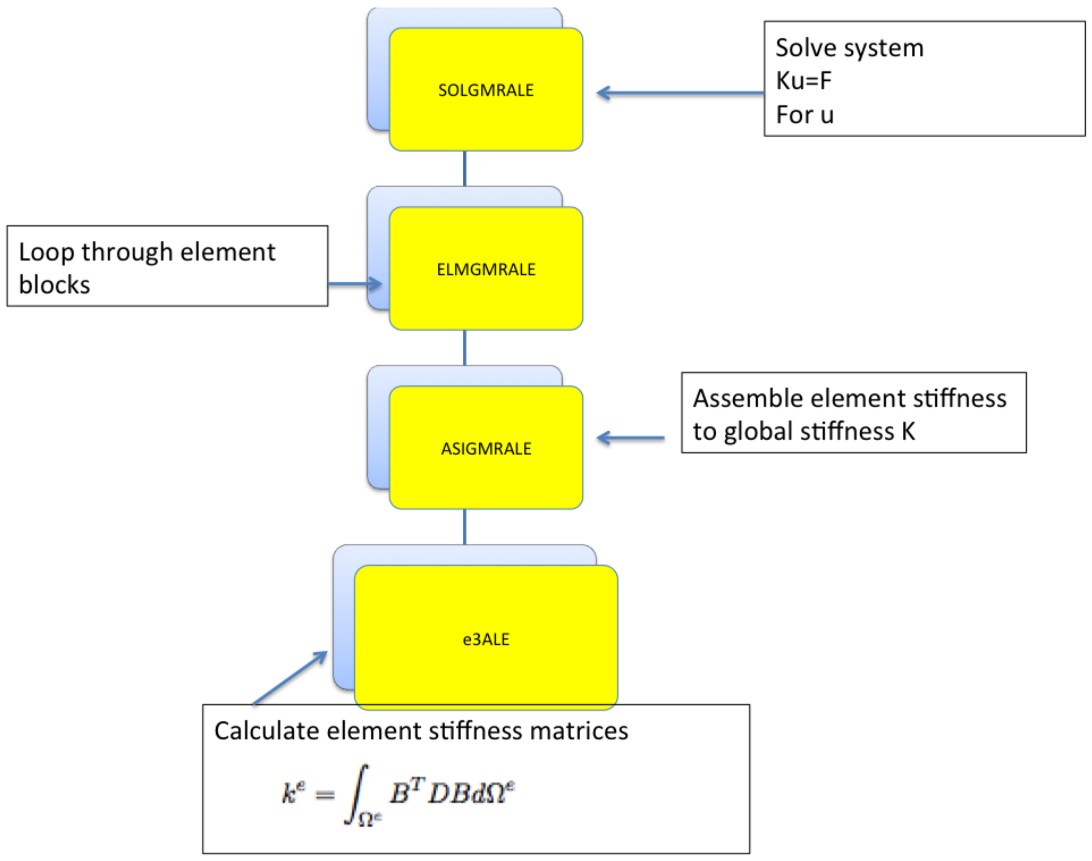

SOLGMRALE
Solve system
Ku=F
For u
ELMGMRALE
Loop through element blocks
ASIGMRALE
Assemble element stiffness to global stiffness K
e3ALE
Calculate element stiffness matrices
$k^e = \int_{\Omega^e} B^T D B d\Omega^e$

Figure 15: Solution Process for Mesh Correction

computational structure is deliberately made to follow the basic architecture of the fluid solver and structural solver. When a mesh correction is required, the subroutine *solgmrALE* is called from *itrdrv*.

Solgmrale solves the basic linear static structural displacement problem by forming a global stiffness matrix K through a call to the subroutine *elmgrmALE*. Displacment boundary condition force modification is then used to determine the applied forces for the system, and the correct displacement solution to satisfy the stiffness equation is found using the GMRES method.

The subroutine *elmgrmALE* contains a loop over the element blocks. Within the loop, a call is made to the subroutine *asigmrALE*, where the assembly of the global stiffness matrix and reduction for conversion to displacement force boundary conditions occurs. *Asigmrale* localizes the mesh coordinate data, and calls the subroutine *e3ALE* to form element stiffness matrices. These stiffness matrices are then assembled the global stiffness matrix using the subroutine *fillsparseALE*. The block diagonal of the global stiffness matrix is also formed as BDiagALEl and assembled the global block diagonal, BDiagALE.

The element stiffness is modified based on the element volume so that the smaller elements are stiffer and experience lesser deformations. This is done by giving identical Lamé constants for all elements, and modifying these constants by dividing by the determinant of the Jacobian of the element. Smaller elements have a smaller determinant of the Jacobian, and thus retain more stiffness than larger elements.

Modification of the equations for displacement force boundary conditions occurs in two phases. The first phase occurs during assembly from local to global stiffness matrix. During this assembly process in the subroutine *fillsparseALE*, a check is used to determine where the PHASTA input variable scalar 3 is set. Wherever this variable is set, displacement boundary conditions are assumed. The rows of the stiffness matrix corresponding to these nodes are set to zero on off diagonal terms and one on the diagonal. The force vector for the mesh deformation is then modified so that force terms are set equal to prescribed displacements. The result is a reduced system with trivial equations for prescribed nodal displacements. The altered sparse matrix is then multiplied by the prescribed displacement vector to determine the corresponding force vector for the solution. The second phase is the reduction of the columns of the sparse stiffness matrix corresponding to prescribed displacements. The subroutine *fullreduceALE* is used to accomplish this. With the corresponding displacement boundary condition force vector formed and the system of equations fully reduced, the correct prescribed displacement boundary condition solution can be solved for.

3.5 Interfaces

The creation of several interfaces for the communication between the fluid solver, the structural solver, and the mesh corrector was necessary. These interfaces all require knowledge of the nodes which apply to the respective physics. This was accomplished by implementing a new subroutine, *divvyibc*. This subroutine loops through the global nodes and identifies nodes that belong to both the structural and fluid solver. These nodes are placed in the array FSmap, which identifies the relation of the fluid node number to the corresponding structure node number, and is zero when an interface is not present. The subroutine *flux* is used to determine aerodynamic forces in a given time step, and a new subroutine *forcesxtoforcesxs* was used to map these forces to the appropriate structure nodes. After completion of the structural solver, the new subroutine *itrdbcale* uses the structural displacements as prescribed displacement boundary conditions for the mesh, and passes these values to the mesh corrector.

4 Results

4.1 Verification

To complete the overall solution of any FSI problem, the solver must solve the fluid equations, apply aerodynamic forces to the structural body, solve the structural dynamics equations, apply displacements to the mesh, correct the mesh to maintain positive volumes throughout the displacement, and adapt the fluid equations for mesh displacement. This means there are three solvers that must work, and three interfaces between solvers. For this purpose, a series of unit tests were performed to ensure that each solver and interface were working as expected. To verify the fluid solution, a simple unit problem was compared after modification of the code to ensure that the fluid solution was the same within machine tolerance to an unmodified version of PHASTA.

The other unit tests are discussed in more detail.

4.2 Verification of ALE Fluid Solution

Two unit tests were performed to verify the ALE adaptation of the fluid equations. One unit test was performed to determine if the solution of a simple Poiselle flow problem in a 0.1m cube could be performed with prescribed mesh displacements and velocities on interior nodes to match the solution of the same problem with no mesh displacement or velocity. This ensures that the fluid solver correctly accounts for mesh velocity in determining the fluid solution. The test case is shown in Figure 16.

For the test, the Poiselle flow test case was run with no mesh velocity to produce a baseline solution. Next, mesh velocities were added to the interior nodes in a direction parallel to the flow, with the displacement highest in the center. The displacement was assigned using a sin function varying with time step number:

$$-(0.1-x)xsin((0.1)n-1) \tag{79}$$

The fluid solver was able to correctly resolve the solution with these mesh velocities, confirming that the ALE modification allows PHASTA to account for non-zero mesh velocity and displacement.

The second unit test for the ALE modification was to perform displacement of the boundary to determine if fluid velocity was correctly altered by a moving boundary. The code is designed only to implement moving boundaries on the fluid structure interface, where a no slip boundary condition will be present. This means that fluid velocity that is non-zero relative to the frame of the fluid-structure boundary will be non-zero relative to the fluid frame when the structure is moving. This velocity must be imparted to the fluid.

To verify that this is occurring, a test case of static flow within a 1m cube was created. Within the center of the cube is a cylinder with a 10cm diameter. This cylinder is displaced to simulate a moving structure boundary. The case produces a flow velocity in the same

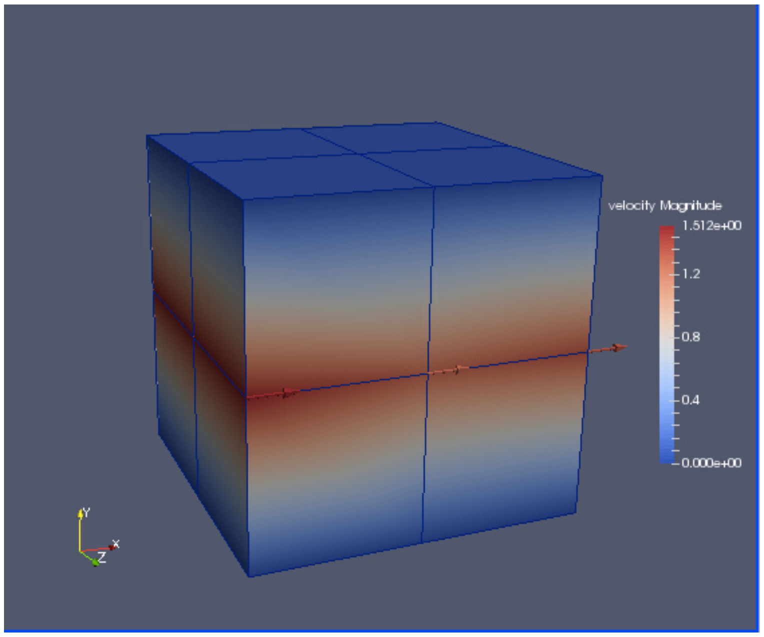


Figure 16: Poiselle Test Case for Prescribed Mesh Motion

direction of the displacement of the cylinder, verifying that no slip boundary velocities are imparted to the flow.

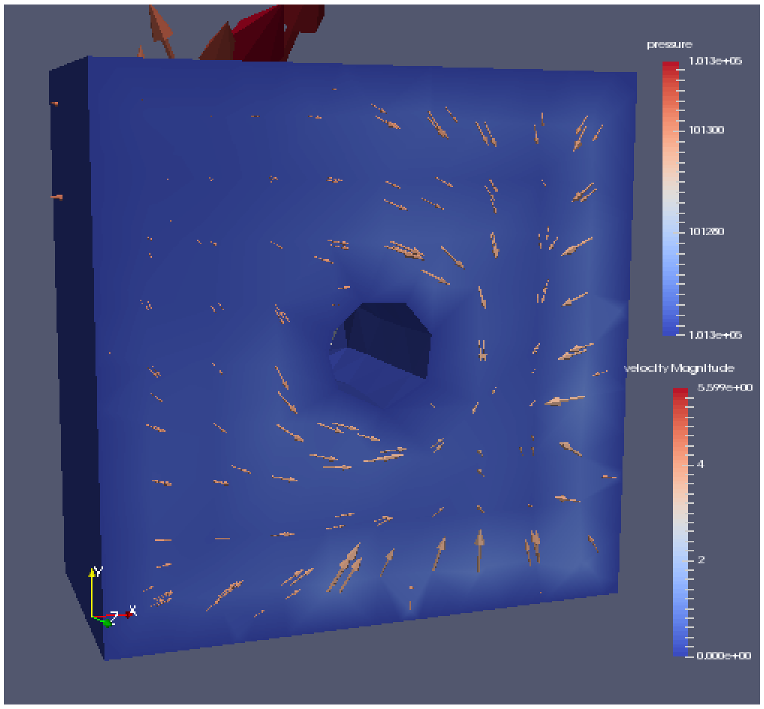


Figure 17: Demonstration of Velocity Imparted to Flow from Moving Boundary

4.3 Verification of Structural Model and Solutions

The first step in verification of the structural solution was verification that the mass and stiffness matrices formed by the solver were correct. This was done.

A simple test case of a 0.1m cube under compression was used to verify the solution of the structural model. These results were compared to the static solution of the same case and the dynamic results from FEAPpv[11]. A two dimensional representation of the problem rendered by FEAPpv is shown in Figure 18.

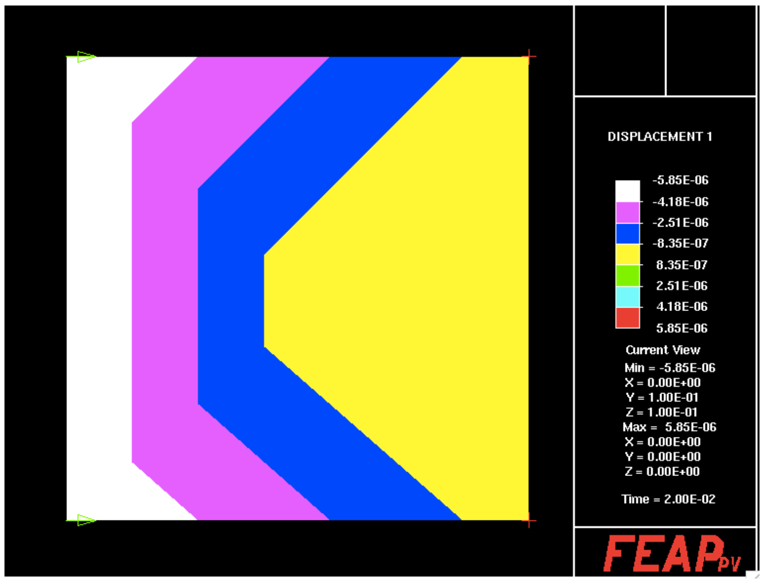


Figure 18: Test Problem with Boundary Conditions from FEAPpv

The cube is given a modulus of elasticity of 195000 Pa with a Poisson ratio of 0.265. A force of the cube in the positive direction of 0.1N is placed on each corner of the cube on the face x=0. The face x=0.1 is constrained with a zero displacement boundary condition. The static solution of the displacement in one dimension can be found from:

$$\delta = \frac{FL}{AE} \tag{80}$$

The maximum displacement from the dynamic results will be larger, and oscillate near this static equilibrium value. Time varying displacement of the solution returned by the structural solver are compared with time varying solutions of displacement returned from FEAPpv. The structural solver implemented in PHASTA uses the generalized alpha method with time integration constants α_f and $\alpha_m$equal to one, which is equivalent to the Newmark method used by FEAPpv. These results are on the same order of magnitude and exhibit similar behavior. The time step was 100 micro-seconds for both solvers.

The volume was meshed with one hexahedra in fear and four hexahedra in PHASTA. The maximum displacement returned by FEAPpv was 109.6 micrometers, while the maximum displacement returned by PHASTA was 89.2 micrometers. Differences are the result of variation in the mesh formed by FEAPpv and the mesh formed for PHASTA, and are within the second order of accuracy expected. The frequency of oscillation of the structure returned by FEAPpv is approximately 47.4 Hz, while the frequency of oscillation from PHASTA results is approximately 50 Hz. Results are similar enough to grant confidence that the structural solver implemented in PHASA is working as expected. The results from PHASTA and FEAPpv are shown in Figure 19.

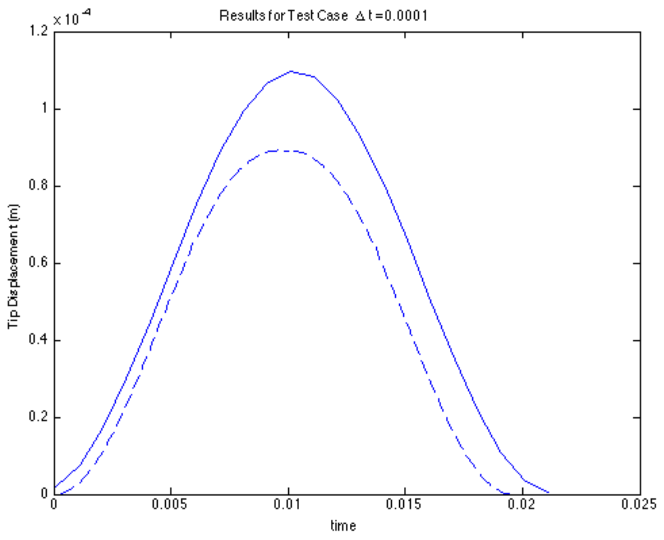


Figure 19: Structural Dynamics Test Case Results from FEAPpv (solid) and Phasta (dashed)

These results verify the structural solver is able to resolve a simple test problem with small displacements.

4.4 Verification of Mesh Correction

The mesh must correct due to deformation caused by structural displacements to prevent inversion of volume elements, which would invalidate the solution. This is verified in a unit test in which the wall of a 0.1m cube is displaced in the x direction. The nodal displacements calculated by the mesh correction must not result in inversion of volume elements while accounting for prescribed displacements. Further, the mesh displacement must maintain prescribed zero displacement boundary conditions, which for this test case was all other faces of the cube.

The test was successful in meeting these criteria for the test case until a deformation of approximately 0.025m was enforced. This is approximately the length of the largest elements on the deforming boundary. Beyond this point, boundary conditions could no longer be satisfied. The test was successful in demonstrating successful mesh correction for small displacements. The mesh corrections calculated up to the point of failure are shown in the Figure 20

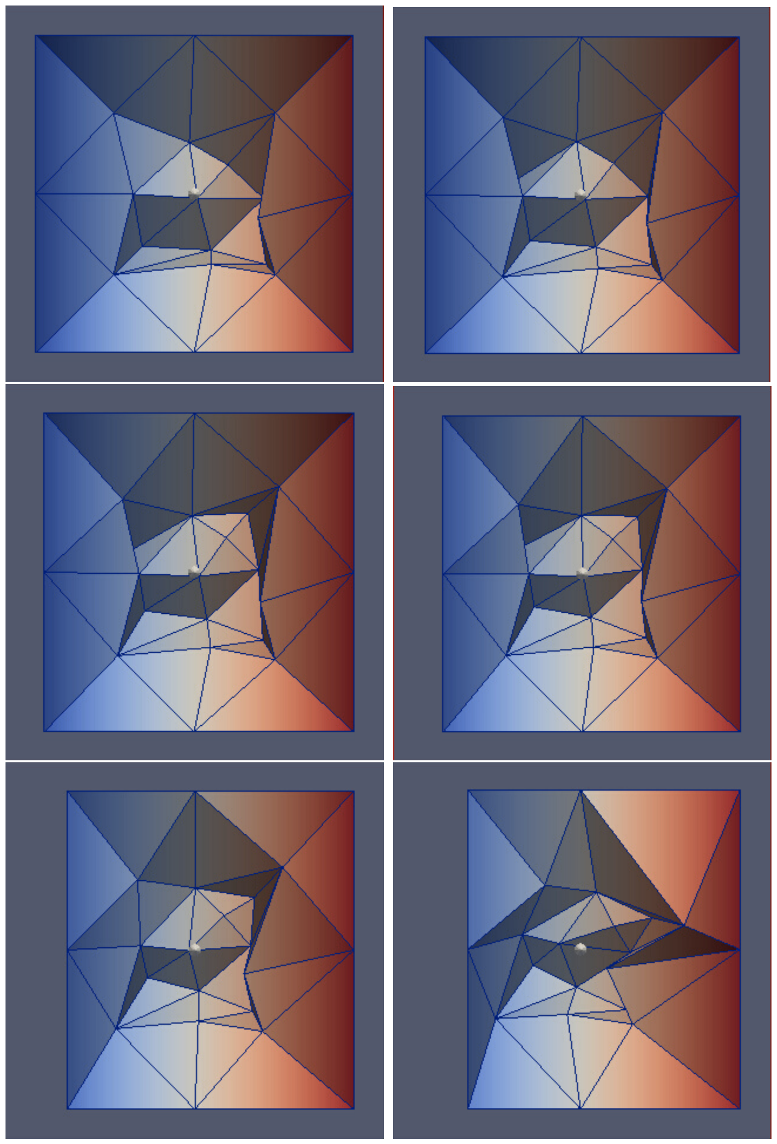


Figure 20: Mesh Correction for Increasing Deformation

4.5 Verfication of Fluid Structure Interaction Behavior

A simple unit test was created to verify fluid and structure sections of the mesh were separated correctly and assigned to the correct solver. This test was also used to verify the interface of the solvers between each other. These interfaces consist of the passing of the aerodynamic forces from the structural solver to the fluid solver, the passing of the structural displacements to the mesh correction, and the passing of mesh velocities to the fluid solver. The geometry of the model is shown in Figure 21.

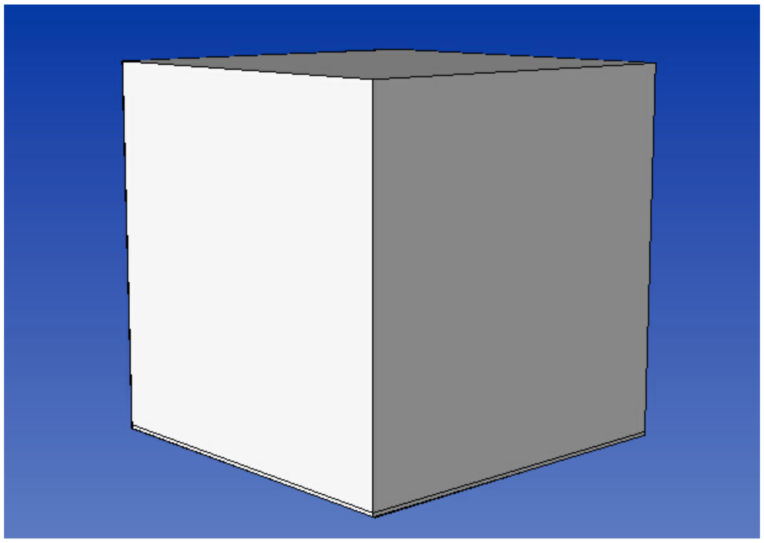


Figure 21: Interface Verification Test Geometry

The test consists of a simple 1m box with stagnant flow for the fluid. On the face of the box where y=0, a structure consisting of a 1mx1mx1cm plate is attached. The fluid interface was confirmed to be working by verifying a total force of 101300 Pa, the pressure in the fluid section, was passed to the structure nodes. The mesh interface was confirmed to be working when the resolved structural displacements were passed to the mesh. This test demonstrates that all the interfaces are working correctly and that the solvers are able to operate in conjunction.

A second verification test for fluid structure interaction behavior was to place a bluff structural body in a cross flow. For this test, a square cylinder with a length of 10cm and height of 20cm was placed in a crossflow with 1m/s velocity. Zero displacement boundary conditions were placed on the nodes on the square edges of the cylinder. The reaction of the center of the cylinder to aerodynamic forces present is shown in Figure 22. In the figure, the center of the cylinder is the small square section enclosed by zero velocity four elements high by four elements wide.

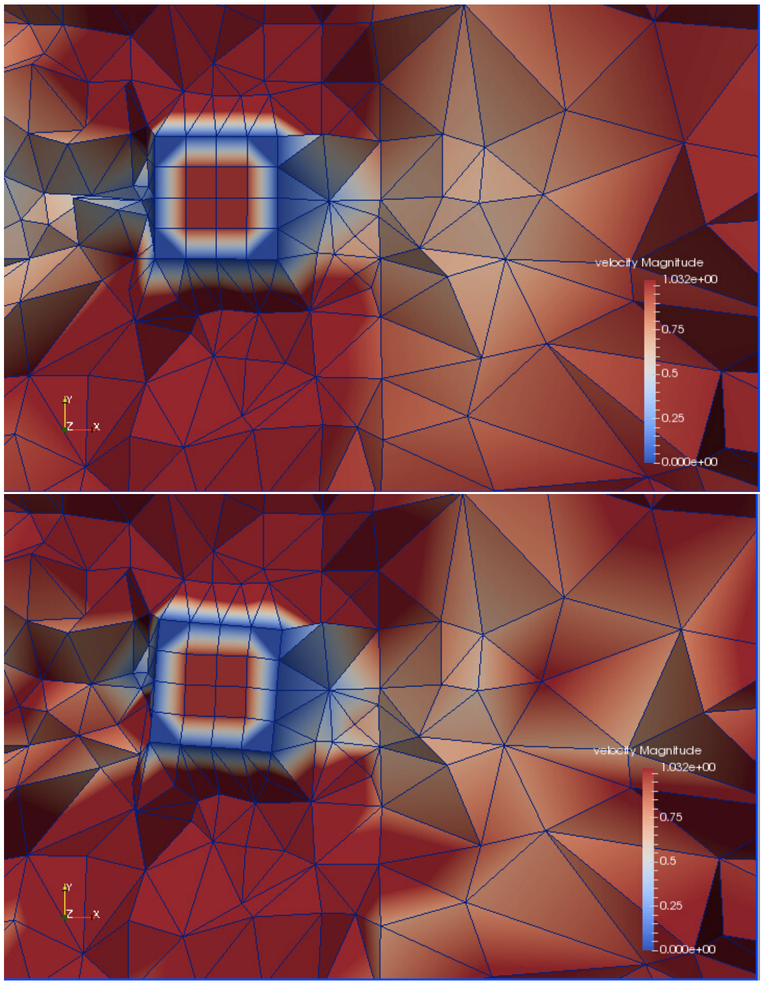


Figure 22: Deflection of Square Cylinder due to Aerodynamic Forces

5 Conclusions

The data structures necessary to implement fluid structure interaction in PHASTA using an added structural solver and mesh correction are detailed in this document. On a broad level, this consists of creating a structural dynamics solver, creating a mesh corrector, and modifying PHASTA to use an ALE formulation for fluid dynamics. These routines were implemented and tested. These new subroutines are able to add limited capability to PHASTA to solve linear dynamic structural problems. The mesh is capable of correcting for small displacements of boundaries to prevent inversion of volume elements. The high fidelity computational fluid dynamics capabilities of PHASTA are maintained. This approach is limited in scope so that problems of engineering interest concerning fluid structure may not be resolved adequately. This is largely due to the simplicity of the structural solver and the mesh correction approach limiting the regime of structural dynamics in which valid results will be returned. However, the data structure created are valid for a more robust structural system, and demonstrate that the capability of PHASTA to solve loosely coupled fluid structure interaction problems. Additional work on the implementation of PHASTA for FSI should include adding capability to the pre-processor to allow material flagging so that structural elements are not constrained to linear hexahedra, as well as adding structural and mesh boundary conditions to the input. Additionally, the subroutines should allow parallelization. Since the subroutines closely follow the hierarchy and data structures of PHASTA, this implementation can be accomplished. Issues such as appropriate load balancing and iteration behavior still need to be examined. With further work to include a more robust generalized dynamics structural solver and mesh correction routine, as well as parallelizing the data structures present, application of engineering interest concerning fluid structure interaction could be resolved with high fidelity using this approach.

References

[1] Bazllevs, Y., Cal, Y.M., Hughes, T.J.R., Zhang, Y. ”Isogeometric fluid-structure interaction: theory, algorithms, and computations.” Computational Mechanics. Vol 43. p3-37. 2008.

[2] Felippa, C. ”Aero- and Hydrodynamic Loading.” Nonlinear Finite Elements Course Notes. Chapter 36. Retrieved from http://www.colorado.edu/engineering/CAS/courses.d/NFEM.d/NFEM.Ch36.d/NFEM.Ch36.pdf

[3] Jansen, K.E., Christian, H.W., Hulbert, G.M. ”A generalized-α method for integrating the filtered Navier-Stokes equations with a stabilized finite element method.” Computer Methods in Applied Mechanics and Engineering. Vol 190. p. 305-319. 200.

[4] Hughes, T.J.R. ”The Finite Element Method.” Dover Publications Inc. New York. 1987.

[5] Chowdhury, I., Dasgupta, S.P. ”Computation of Rayleigh Damping Coefficients for Large Systems.” Retrieved from http://www.ejge.com/2003/Ppr0318/Ppr0318.pdf

[6] Felippa, C. ”Dynamic Stability Formulation.” Nonlinear Finite Elements Course Notes. Chapter 37. Retrieved from http://www.colorado.edu/engineering/CAS/courses.d/NFEM.d/NFEM.Ch37.d/NFEM.Ch37.pdf

[7] Chung, J. Hulbert, G.M. ”A Time Integration Algorithm for Structural Dynamics with Improved Numerical Dissipation: The Generalized- α Method.” Journal of Applied Mechanics. Vol 60. p. 371-375. 1993.

[8] Felippa, C. ”The Direct Stiffness Method II.” Introduction to Finite Elements Course Notes. Chapter 3. Retrieved from http://www.colorado.edu/engineering/CAS/courses.d/IFEM.d/IFEM.Ch03.d/IFEM.Ch03.pdf

[9] Shakib, F. ”Finite Element Analysis of the Compressible Euler And Navier-Stokes Equations.” Unpublished doctoral thesis/dissertation, Stanford University. 1988.

[10] Saad, Y., Schultz, M.H. ”GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems.” SIAM J. Sci. Stat. Comput., Vol. 7 p. 856-869. 1986.

[11] FEAPpv. A Finite Element Analysis Program: Personal Version. Retrieved from http://www.ce.berkeley.edu/projects/feap/feappv/

6 Appendix A: Stiffness Matrix Verification

This simple test program for the formulation of the element stiffness matrix was written for a square hexahedra element with length 0.5 on each edge.

```
5
%hex stiffness test


mu= 7.617e+10
lambda= 9.695e+10
WdetJ=1.5625E−05

D=[lambda+2*mu, lambda , lambda ,0 ,0 ,0; lambda , lambda+2*mu, lambda
    ,0 ,0 ,0;...
     lambda , lambda , lambda+2*mu,0 ,0 ,0;0 ,0 ,0 ,mu,0 ,0; ,0 ,0 ,0 ,0 ,mu
         ,0;0 ,0 ,0 ,0 ,0 ,mu];

X=[0.1 , −0.1 ,0;...
     0.05 , −0.1 ,0;...
     0.05 , −0.1 ,.05;...
     0.1 , −0.1 ,0.05;...
     0.1 , −0.05 ,0;...
     0.05 , −0.05 ,0;...
     0.05 , −0.05 ,0.05;...
     0.1 , −0.05 ,0.05];


%int 1

xi=−1/sqrt(3);
eta=−1/sqrt(3);
mu=−1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
         eta ,mu,X)

[xxi , xeta , xmu, yxi , yeta , ymu, zxi , zeta , zmu]=shapexderivs(X, N11,N12,
    N13,...
     N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83, xi , eta ,mu);
```

```
[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi1 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi1t=transpose(Bi1);

k1=WdetJ*Bi1t*D*Bi1;

%int 2
xi=-1/sqrt(3);
eta=-1/sqrt(3);
mu=1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
        eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X,  N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi2 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
    NR41,NR42,NR43,NR51,...
     NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);

Bi2t=transpose(Bi2);

k2=WdetJ*Bi2t*D*Bi2;
%int 3
xi=-1/sqrt(3);
eta=1/sqrt(3);
mu=-1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
         eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X, N11,N12,
    N13,...
     N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu);

[ NR11,NR12,NR13,...
     NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
         ,...
     NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
         (X, N11,N12,N13,...
     N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu,xxi,xeta,xmu
         ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi3 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
    NR41,NR42,NR43,NR51,...
     NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi3t=transpose(Bi3);
k3=WdetJ*Bi3t*D*Bi3;
%int 4

xi=-1/sqrt(3);
eta=1/sqrt(3);
mu=1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
```

```
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
        eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X, N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X, N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi4 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi4t=transpose(Bi4);
k4=WdetJ*Bi4t*D*Bi4;
%int 5

xi=1/sqrt(3);
eta=-1/sqrt(3);
mu=-1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
        eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X, N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
```

```
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi5 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi5t=transpose(Bi5);
k5=WdetJ*Bi5t*D*Bi5;
%int 6

xi=1/sqrt(3);
eta=-1/sqrt(3);
mu=1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
        eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X,  N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi6 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi6t=transpose(Bi6);
k6=WdetJ*Bi6t*D*Bi6;
%int 7
```

```
 xi=1/sqrt(3);
 eta=1/sqrt(3);
mu=-1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
       eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X, N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
       ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
       (X, N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu,xxi,xeta,xmu
       ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi7 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi7t=transpose(Bi7);
k7=WdetJ*Bi7t*D*Bi7;
%int 8


 xi=1/sqrt(3);
 eta=1/sqrt(3);
mu=1/sqrt(3);

[N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
       eta,mu,X)

[xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X, N11,N12,
   N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu);

[ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu);

Bi8 = makestiffB(NR11,NR12,NR13,NR21,NR22,NR23,NR31,NR32,NR33,
   NR41,NR42,NR43,NR51,...
    NR52,NR53,NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83);
Bi8t=transpose(Bi8);

k8=WdetJ*Bi8t*D*Bi8;

KH=k1+k2+k3+k4+k5+k6+k7+k8;

\begin{lstlisting}
function [ NR11,NR12,NR13,...
    NR21,NR22,NR23,NR31,NR32,NR33,NR41,NR42,NR43,NR51,NR52,NR53
        ,...
    NR61,NR62,NR63,NR71,NR72,NR73,NR81,NR82,NR83]=shapederivsreal
        (X,  N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83,  xi,  eta  ,mu,xxi,xeta,xmu
        ,yxi,yeta,ymu,zxi,zeta,zmu)

J=[xxi,xeta,xmu;yxi,yeta,ymu;zxi,zeta,zmu];
j=det(J);


cof11=yeta*zmu-ymu*zeta;
cof12=ymu*zxi-yxi*zmu;
cof13=yxi*zeta-zxi*yeta;
cof21=zeta*xmu-zmu*xeta;
cof22=zmu*xxi-zxi*xmu;
cof23=zxi*xeta-xxi*zeta;
cof31=xeta*ymu-xmu*yeta;
```

```
cof32=xmu*yxi-xxi*ymu;
cof33=xxi*yeta-yxi*xeta;

%j=xxi*cof11+xeta*cof12+xmu*cof13;

NR11=(N11*cof11+N12*cof12+N13*cof13)/j;
NR12=(N11*cof21+N12*cof22+N13*cof23)/j;
NR13=(N11*cof31+N12*cof32+N13*cof33)/j;

NR21=(N21*cof11+N22*cof12+N23*cof13)/j;
NR22=(N21*cof21+N22*cof22+N23*cof23)/j;
NR23=(N21*cof31+N22*cof32+N23*cof33)/j;

NR31=(N31*cof11+N32*cof12+N33*cof13)/j;
NR32=(N31*cof21+N32*cof22+N33*cof23)/j;
NR33=(N31*cof31+N32*cof32+N33*cof33)/j;

NR41=(N41*cof11+N42*cof12+N43*cof13)/j;
NR42=(N41*cof21+N42*cof22+N43*cof23)/j;
NR43=(N41*cof31+N42*cof32+N43*cof33)/j;

NR51=(N51*cof11+N52*cof12+N53*cof13)/j;
NR52=(N51*cof21+N52*cof22+N53*cof23)/j;
NR53=(N51*cof31+N52*cof32+N53*cof33)/j;

NR61=(N61*cof11+N62*cof12+N63*cof13)/j;
NR62=(N61*cof21+N62*cof22+N63*cof23)/j;
NR63=(N61*cof31+N62*cof32+N63*cof33)/j;

NR71=(N71*cof11+N72*cof12+N73*cof13)/j;
NR72=(N71*cof21+N72*cof22+N73*cof23)/j;
NR73=(N71*cof31+N72*cof32+N73*cof33)/j;

NR81=(N81*cof11+N82*cof12+N83*cof13)/j;
NR82=(N81*cof21+N82*cof22+N83*cof23)/j;
NR83=(N81*cof31+N82*cof32+N83*cof33)/j;
```

5

```
function [xxi,xeta,xmu,yxi,yeta,ymu,zxi,zeta,zmu]=shapexderivs(X,
    N11,N12,N13,...
    N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52,N53,...
    N61,N62,N63,N71,N72,N73,N81,N82,N83, xi, eta ,mu)

x1=X(1,1);
x2=X(2,1);
x3=X(3,1);
x4=X(4,1);
x5=X(5,1);
x6=X(6,1);
x7=X(7,1);
x8=X(8,1);

y1=X(1,2);
y2=X(2,2);
y3=X(3,2);
y4=X(4,2);
y5=X(5,2);
y6=X(6,2);
y7=X(7,2);
y8=X(8,2);

z1=X(1,3);
z2=X(2,3);
z3=X(3,3);
z4=X(4,3);
z5=X(5,3);
z6=X(6,3);
z7=X(7,3);
z8=X(8,3);

xxi = x1*N11+x2*N21+x3*N31+x4*N41+x5*N51+x6*N61+x7*N71+x8*N81;
xeta = x1*N12+x2*N22+x3*N32+x4*N42+x5*N52+x6*N62+x7*N72+x8*N82;
xmu = x1*N13+x2*N23+x3*N33+x4*N43+x5*N53+x6*N63+x7*N73+x8*N83;

yxi = y1*N11+y2*N21+y3*N31+y4*N41+y5*N51+y6*N61+y7*N71+y8*N81;
yeta = y1*N12+y2*N22+y3*N32+y4*N42+y5*N52+y6*N62+y7*N72+y8*N82;
ymu = y1*N13+y2*N23+y3*N33+y4*N43+y5*N53+y6*N63+y7*N73+y8*N83;

zxi = z1*N11+z2*N21+z3*N31+z4*N41+z5*N51+z6*N61+z7*N71+z8*N81;
zeta = z1*N12+z2*N22+z3*N32+z4*N42+z5*N52+z6*N62+z7*N72+z8*N82;
zmu = z1*N13+z2*N23+z3*N33+z4*N43+z5*N53+z6*N63+z7*N73+z8*N83;
```

5

```
function [N11,N12,N13,N21,N22,N23,N31,N32,N33,N41,N42,N43,N51,N52
    ,N53,...
     N61,N62,N63,N71,N72,N73,N81,N82,N83]=shapefunctionderiv8(xi,
         eta,mu,X)

%calculate the shape function derivatives at an integration point
      in weight
%space

%ex N11 is dN1/dxi, N12 is DN1/deta, N13 is dN1/dmu

N11=-1/8*(1-eta)*(1-mu);
N12=-1/8*(1-xi)*(1-mu);
N13=-1/8*(1-xi)*(1-eta);
N21=1/8*(1-eta)*(1-mu);
N22=-1/8*(1+xi)*(1-mu);
N23=-1/8*(1+xi)*(1-eta);
N31=1/8*(1+eta)*(1-mu);
N32=1/8*(1+xi)*(1-mu);
N33=-1/8*(1+xi)*(1+eta);
N41=-1/8*(1+eta)*(1-mu);
N42=1/8*(1-xi)*(1-mu);
N43=-1/8*(1-xi)*(1+eta);
N51=-1/8*(1-eta)*(1+mu);
N52=-1/8*(1-xi)*(1+mu);
N53=1/8*(1-xi)*(1-eta);
N61=1/8*(1-eta)*(1+mu);
N62=-1/8*(1+xi)*(1+mu);
N63=1/8*(1+xi)*(1-eta);
N71=1/8*(1+eta)*(1+mu);
N72=1/8*(1+xi)*(1+mu);
N73=1/8*(1+xi)*(1+eta);
N81=-1/8*(1+eta)*(1+mu);
N82=1/8*(1-xi)*(1+mu);
N83=1/8*(1-xi)*(1+eta);
```

5

```
function B = makestiffB(n11,n12,n13,n21,n22,n23,n31,n32,n33,n41,
    n42,n43,n51,n52,n53,n61,n62,n63,n71,n72,n73,n81,n82,n83)


B=[n11,0,0,n21,0,0,n31,0,0,n41,0,0,n51,0,0,n61,0,0,n71,0,0,n81
    ,0,0;...
     0,n12,0,0,n22,0,0,n32,0,0,n42,0,0,n52,0,0,n62,0,0,n72,0,0,n82
         ,0;...
     0,0,n13,0,0,n23,0,0,n33,0,0,n43,0,0,n53,0,0,n63,0,0,n73,0,0,
         n83;...
     0,n13,n12,0,n23,n22,0,n33,n32,0,n43,n42,0,n53,n52,0,n63,n62
         ,0,n73,n72,0,n83,n82;...
     n13,0,n11,n23,0,n21,n33,0,n31,n43,0,n41,n53,0,n51,n63,0,n61,
         n73,0,n71,n83,0,n81;...
     n12,n11,0,n22,n21,0,n32,n31,0,n42,n41,0,n52,n51,0,n62,n61,0,
         n72,n71,0,n82,n81,0];
```

7 Appendix B: ALE Modified Transformation Matrices

Subroutine e3mtrx modified for ALE

Summary of changes: ui replaced by (ui - xdoti) in the advective term.

5

```
        subroutine e3mtrx (rho,     pres,    T,
     &                     ei,      h,       alfap,
     &                     betaT,   cp,      rk,
     &                     u1,      u2,    u3,
     &                     A0,      A1,
     &                     A2,      A3,
     &                     e2p,     e3p,     e4p,
     &                     drdp,    drdT,    A0DC,
     &                     A0inv,   dVdY,xdotl,xdot1,xdot2,xdot3)
c
c
    ------------------------------------------------------------------------

c
c This routine sets up the necessary matrices at the integration
    point.
c
c input:
```

```
c  rho    (npro)          : density
c  pres   (npro)          : pressure
c  T      (npro)          : temperature
c  ei     (npro)          : internal energy
c  h      (npro)          : enthalpy
c  alfap  (npro)          : expansivity
c  betaT  (npro)          : isothermal compressibility
c  cp     (npro)          : specific heat at constant pressure
c  c      (npro)          : speed of sound
c  rk     (npro)          : kinetic energy
c  u1     (npro)          : x1-velocity component
c  u2     (npro)          : x2-velocity component
c  u3     (npro)          : x3-velocity component
c
c output:
c  A0     (npro,nflow,nflow)   : A0 matrix
c  A1    (npro,nflow,nflow)   : A_1 matrix
c  A2    (npro,nflow,nflow)   : A_2 matrix
c  A3    (npro,nflow,nflow)   : A_3 matrix
c
c Note: the definition of the matrices can be found in
c       thesis by Hauke.
c
c Zdenek Johan, Summer 1990.  (Modified from e2mtrx.f)
c Zdenek Johan, Winter 1991.  (Fortran 90)
c Kenneth Jansen, Winter 1997 Primitive Variables
c
   ----------------------------------------------------------------------

c
        include "common.h"
c
c
c  passed arrays
c
        dimension rho(npro),                   pres(npro),
     &            T(npro),                     ei(npro),
     &            h(npro),                     alfap(npro),
     &            betaT(npro),
     &            cp(npro),
     &            rk(npro),
     &            u1(npro),                   u2(npro),
     &            u3(npro),                   fact1(npro),
     &            A0(npro,nflow,nflow),       dVdY(npro,15),
     &            A1(npro,nflow,nflow),       A2(npro,nflow,nflow),
     &            A3(npro,nflow,nflow),       A0DC(npro,4),
     &            A0inv(npro,15),             d(npro),
     &            fact2(npro),                 s1(npro),
     &            e1bar(npro),                e2bar(npro),
     &            e3bar(npro),                e4bar(npro),
     &            e5bar(npro),                c1bar(npro),
     &            c2bar(npro),                cv(npro),
     &            c3bar(npro),                u12(npro),
     &            u31(npro),                  u23(npro)
c
c  local work arrays that are passed shared space
c
        dimension e2p(npro),
     &            e3p(npro),                   e4p(npro),
     &            drdp(npro),                  drdT(npro)
c DEP added variables
        dimension xdotl(npro, nenl,nsd), xdot1(npro),xdot2(npro),
           xdot3(npro)
c end DEP added

        ttim(21) = ttim(21) - secs(0.0)
c
c.... initialize
c
        A0 = zero
        A1 = zero
        A2 = zero
        A3 = zero
c
c.... set up the constants
c
c
        drdp = rho * betaT
        drdT = -rho * alfap
        A0(:,5,1) = drdp * (h + rk)  - alfap * T     ! e1p
c        A0(:,5,1) = drdp * (ei + rk) + betaT * pres - alfap * T
      ! e1p
```

```
               e2p   = A0(:,5,1) + one
               e3p   = rho * ( h + rk)
               e4p   = drdT * (h + rk) + rho * cp
c
c
c....  Calculate A0
c
            A0(:,1,1) = drdp
c           A0(:,1,2) = zero
c           A0(:,1,3) = zero
c           A0(:,1,4) = zero
            A0(:,1,5) = drdT
c

            A0(:,2,1) = drdp * u1
            A0(:,2,2) = rho
c           A0(:,2,3) = zero
c           A0(:,2,4) = zero
            A0(:,2,5) = drdT * u1
c
            A0(:,3,1) = drdp * u2
c           A0(:,3,2) = zero
            A0(:,3,3) = rho
c           A0(:,3,4) = zero
            A0(:,3,5) = drdT * u2
c
            A0(:,4,1) = drdp * u3
c           A0(:,4,2) = zero
c           A0(:,4,3) = zero
            A0(:,4,4) = rho
            A0(:,4,5) = drdT * u3
c
covered above         A0(:,5,1) = drdp * u1
            A0(:,5,2) = rho * u1
            A0(:,5,3) = rho * u2
            A0(:,5,4) = rho * u3
            A0(:,5,5) = e4p
c
            flops = flops + 67*npro
c
c....  Calculate A-tilde-1, A-tilde-2 and A-tilde-3
```

```
c
          A1(:,1,1) = drdp * (u1 - xdot1)
          A1(:,1,2) = rho
c         A1(:,1,3) = zero
c         A1(:,1,4) = zero
          A1(:,1,5) = drdT * (u1 - xdot1)
c
          A1(:,2,1) = drdp * (u1 - xdot1) * u1 +1
          A1(:,2,2) = rho  * (u1 - xdot1) + rho * u1
c         A1(:,2,3) = zero
c         A1(:,2,4) = zero
          A1(:,2,5) = drdT * (u1 - xdot1) * u1
c
          A1(:,3,1) = drdp * (u1 - xdot1) * u2
          A1(:,3,2) = rho  * u2
          A1(:,3,3) = rho  * (u1 - xdot1)
c         A1(:,3,4) = zero
          A1(:,3,5) = drdT * (u1 - xdot1) * u2
c
          A1(:,4,1) = drdp * (u1 - xdot1) * u3
          A1(:,4,2) = rho  * u3
c         A1(:,4,3) = zero
          A1(:,4,4) = rho  * (u1 - xdot1)
          A1(:,4,5) = drdT * u1 * u3
c
          A1(:,5,1) = (u1 - xdot1) * e2p  +xdot1 !extra -xdot must
             be nullified from P derivative
          A1(:,5,2) = e3p + rho * (u1) * u1
          A1(:,5,3) = rho * (u1) * u2
          A1(:,5,4) = rho * (u1) * u3
          A1(:,5,5) = (u1 - xdot1) * e4p
c
          flops = flops + 35*npro
c
          A2(:,1,1) = drdp * (u2 - xdot2)
c         A2(:,1,2) = zero
          A2(:,1,3) = rho
c         A2(:,1,4) = zero
          A2(:,1,5) = drdT * (u2 - xdot2)
c
          A2(:,2,1) = drdp * u1 * (u2-xdot2)
          A2(:,2,2) = rho   * (u2-xdot2)
          A2(:,2,3) = rho   * u1
c         A2(:,2,4) = zero
          A2(:,2,5) = drdT * u1 * (u2-xdot2)
c
          A2(:,3,1) = drdp * (u2-xdot2) * u2 +1
c         A2(:,3,2) = zero
          A2(:,3,3) = rho   * (u2-xdot2) +rho*u2
c         A2(:,3,4) = zero
          A2(:,3,5) = drdT * (u2-xdot2) * u2
c
          A2(:,4,1) = drdp * (u2-xdot2) * u3
c         A2(:,4,2) = zero
          A2(:,4,3) = rho   * u3
          A2(:,4,4) = rho   * (u2-xdot2)
          A2(:,4,5) = drdT * (u2-xdot2) * u3
c
          A2(:,5,1) = (u2-xdot2) * e2p + xdot2
          A2(:,5,2) = rho * u1 * (u2)
          A2(:,5,3) = e3p + rho * (u2) * u2
          A2(:,5,4) = rho * (u2) * u3
          A2(:,5,5) = (u2-xdot2) * e4p
c
          flops = flops + 35*npro
c
          A3(:,1,1) = drdp * (u3-xdot3)
c         A3(:,1,2) = zero
c         A3(:,1,3) = zero
          A3(:,1,4) = rho
          A3(:,1,5) = drdT * (u3 -xdot3)
c
          A3(:,2,1) = drdp * u1 * (u3 - xdot3)
          A3(:,2,2) = rho   * (u3 - xdot3)
c         A3(:,2,3) = zero
          A3(:,2,4) = rho   * u1
          A3(:,2,5) = drdT * u1 * (u3 - xdot3)
c
          A3(:,3,1) = drdp * (u3 - xdot3) * u2
c         A3(:,3,2) = zero
          A3(:,3,3) = rho   * (u3 - xdot3)
          A3(:,3,4) = rho   * u2
          A3(:,3,5) = drdT * (u3 - xdot3) * u2
c
          A3(:,4,1) = drdp * (u3 - xdot3) * u3 +1
c         A3(:,4,2) = zero
c         A3(:,4,3) = zero
          A3(:,4,4) = rho  * (u3 - xdot3) + rho*u3
          A3(:,4,5) = drdT * (u3 - xdot3) * u3
c
          A3(:,5,1) = (u3 - xdot3) * e2p +xdot3
          A3(:,5,2) = rho * u1 * (u3)
          A3(:,5,3) = rho * u2 * (u3)
          A3(:,5,4) = e3p + rho * (u3) * u3
          A3(:,5,5) = (u3-xdot3) * e4p
c
          flops = flops + 35*npro
          ttim(21) = ttim(21) + secs(0.0)

c
c.... return
c
      if (idc .ne. 0) then
c.... for Discountinuity Capturing Term
c
c.... calculation of A0^DC matrix
c
c.... Ref P-163 of the handout
c
        s1 = one/(rho**2 * betaT * T)
        cv = cp - (alfap**2 * T/rho/betaT)
        A0DC(:,1) = (rho * betaT)**2*s1
        A0DC(:,2) = -rho*alfap*rho*betaT*s1
        A0DC(:,3) = rho/T
        A0DC(:,4) = (-rho*alfap)**2 * s1 + (rho*cv/T**2)
c
c.... calculation of A0^tilda^inverse matrix
c.... ref P-169 of the hand out
c
        fact1 = one/(rho*cv*T**2)
        d = alfap*T/rho/betaT
        e1bar = h - rk
        e2bar = e1bar - d
        e3bar = e2bar - cv * T
        e4bar = e2bar - 2* cv * T
        e5bar = e1bar**2 - 2*e1bar*d + 2*rk*cv*T + cp*T/rho/betaT
        c1bar = u1**2 + cv * T
        c2bar = u2**2 + cv * T
        c3bar = u3**2 + cv * T
        u12 = u1 * u2
        u31 = u3 * u1
        u23 = u2 * u3
        A0inv( :,1) = e5bar*fact1
        A0inv( :,2) = c1bar*fact1
        A0inv( :,3) = c2bar*fact1
        A0inv( :,4) = c3bar*fact1
        A0inv( :,5) = 1*fact1
        A0inv( :,6) = u1*e3bar*fact1
        A0inv( :,7) = u2*e3bar*fact1
        A0inv( :,8) = u3*e3bar*fact1
        A0inv( :,9) = -e2bar*fact1
        A0inv(:,10) = u12*fact1
        A0inv(:,11) = u31*fact1
        A0inv(:,12) = -u1*fact1
        A0inv(:,13) = u23*fact1
        A0inv(:,14) = -u2*fact1
        A0inv(:,15) = -u3*fact1
c
c.....calculation of dV/dY (derivative of entropy variables w.r.
   to primitive
c
       fact1 = 1/T
       fact2 = fact1/T
       dVdY( :,1) = fact1/rho
       dVdY( :,2) =-fact1*u1
       dVdY( :,3) =  fact1
       dVdY( :,4) =  -fact1*u2
       dVdY( :,5) =  zero
       dVdY( :,6) =  fact1
       dVdY( :,7) =  -fact1*u3
       dVdY( :,8) =  zero
       dVdY( :,9) =  zero
       dVdY(:,10) =  fact1
       dVdY(:,11) =  -(h-rk)*fact2
```

```
      dVdY(:,12) =  -fact2*u1
      dVdY(:,13) =  -fact2*u2
      dVdY(:,14) =  -fact2*u3
      dVdY(:,15) =   fact2

      endif  !end of idc.ne.0

        return
        end
c
c
        subroutine e3mtrxSclr (rho,
     &                         u1,      u2,    u3,
     &                         A0t,     A1t,
     &                         A2t,     A3t   )
c
c
 ----------------------------------------------------------------------

c
c This routine sets up the necessary matrices at the integration
  point.
c
c input:
c  rho    (npro)          : density
c  u1     (npro)          : x1-velocity component
c  u2     (npro)          : x2-velocity component
c  u3     (npro)          : x3-velocity component
c
c output:
c  A0t     (npro) : A_0 "matrix"
c  A1t    (npro)  : A_1 "matrix"
c  A2t    (npro)  : A_2 "matrix"
c  A3t    (npro)  : A_3 "matrix"
c
c Note: the definition of the matrices can be found in
c       thesis by Hauke.
c
c Zdenek Johan, Summer 1990.  (Modified from e2mtrx.f)
c Zdenek Johan, Winter 1991.  (Fortran 90)
c Kenneth Jansen, Winter 1997 Primitive Variables
```

```
c
   ----------------------------------------------------------------------
c
         include "common.h"
c
c
c   passed arrays
c
         dimension rho(npro),
     &              u1(npro),          u2(npro),
     &              u3(npro),
     &              A0t(npro),
     &              A1t(npro),         A2t(npro),
     &              A3t(npro)
c
         if (iconvsclr.eq.2) then   !convective form
            A0t(:) = one
            A1t(:) = u1(:)
            A2t(:) = u2(:)
            A3t(:) = u3(:)
         else                       !conservative form
            A0t(:) = rho(:)
            A1t(:) = rho(:)*u1(:)
            A2t(:) = rho(:)*u2(:)
            A3t(:) = rho(:)*u3(:)
         endif

c
c.... return
c
         return
         end
```