University of Colorado
Boulder

Ann and H.J. Smead Department of Aerospace Engineering Sciences

---

ASEN 5331 CFD Unstructured Grid

Final Project Report

Pol Mesalles Ripoll

Boulder, Colorado

June 2021

1 Introduction

The main goal of this project is to implement ***slip* boundary conditions** into the PHASTA codebase in order to improve the accuracy with which it predicts the flow behavior in the continuum-transition or transitional regime (see Figure 1). Both velocity slip and temperature jump occur when the flow is rarefied enough that the no-slip condition —taken for granted in most continuum problems— stops being applicable.

According to previous studies [11, 7], adding this new capability should increase the fidelity when running simulations at higher Knudsen numbers, as is common in both hypersonics during spacecraft planetary (re)entry, for example, as well as microfluidics, giving results closer to experimental data and simulations conducted with particulate methods, such as the Direct Simulation Monte Carlo (DSMC), which has traditionally been the default choice when studying the continuum-transition. The latter is based on the more generally applicable Boltzmann equation, and thus takes into account non-equilibrium effects. However, especially at lower altitudes, or more generally the lower end of the transitional regime, DSMC simulations can be significantly more computationally expensive than their continuum counterparts, since the number of particles to be simulated increases as Knudsen decreases.

Therefore, there is still a place for CFD solvers based on solving the continuum Navier–Stokes transport equations in the study of the transitional regime. With this project, we seek to implement features that can help PHASTA deliver the same quality of results expected from DSMC, but at a lower cost.

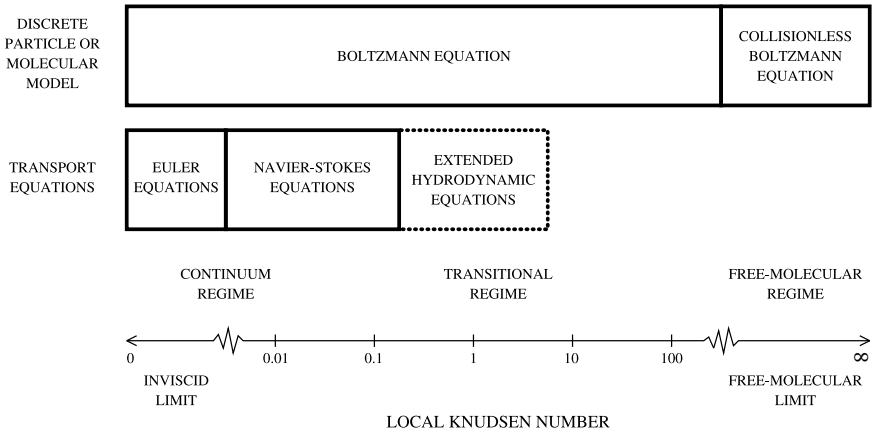


Figure 1: The Knudsen number limits on the mathematical models (reproduced from [1])

In order to complete this project, this document starts by introducing the different slip models. This is followed by a comprehensive overview of the finite element formulation implemented in PHASTA, along with details about each subroutine in the code. Then, we move on to the specifics about the present implementation of slip boundary conditions. Finally, the results obtained are presented and future relevant work is discussed.

The described changes in the code are available on GitHub, published under the `slip` branch of my fork of the `phasta` repository: `https://github.com/polmes/phasta/tree/slip`.

2 Slip boundary conditions

Several slip boundary conditions have been proposed over the years. However, the most common form found in the literature is Maxwell's 1D velocity slip [13, 11]:

$$u_{\text{slip}} \equiv u_r = A\left(\frac{2-\sigma}{\sigma}\right)\lambda\left.\frac{\partial u_r}{\partial x_s}\right|_w + \frac{3}{4}\frac{\mu}{\rho T}\frac{\partial T}{\partial x_r} \tag{1}$$

where, in natural coordinates $\{r, s, t\}$:

- u_{slip} is the fluid velocity at the surface, assuming a fixed wall.
- A is a constant of proportionality, usually taken as 1.
- σ is the tangential momentum accommodation coefficient, which basically relates the amount of molecules that are reflected diffusively ($\sigma = 1$) and specularly ($\sigma = 0$).
- λ is the mean free path, which can be computed from macroscopic properties of the flow as $\lambda = \frac{\mu}{\rho}\sqrt{\frac{\pi}{2RT}}$.
- u_r is the velocity in the tangential direction of the wall.
- s is the direction normal to the wall.

There is also a generalized version of Equation (1), which directly relates the slip velocity with the stress tensor and the heat flux [9]:

$$\boldsymbol{u}_{\text{slip}} - \boldsymbol{u}_{\text{wall}} = A\left(\frac{2-\sigma}{\sigma}\right)\frac{\lambda}{\mu}\boldsymbol{\tau}_{ws} - \frac{3}{4}\frac{\mu}{\rho T}\boldsymbol{q}_{ws} \tag{2}$$

where the subscript ws indicates the shear stress and heat fluxes at the wall in its tangential directions. Relating the stress tensor and heat flux with the velocity and temperature through the constitutive relations used in Navier–Stokes allows us to find more complete expressions for the velocity slip that take into account the surface curvature, which is what we want for a 3D CFD solver [7, 6].

Besides the velocity slip, another effect related to the rarefied slip regime is the so-called *temperature jump* boundary condition. The most common form is that proposed by Smoluchowski [7, 2]:

$$T_{\text{slip}} - T_{\text{wall}} = \frac{2-\alpha}{\alpha}\frac{2\gamma}{\gamma+1}\frac{\lambda}{\text{Pr}}\left.\frac{\partial T}{\partial x_s}\right|_w \tag{3}$$

where:

- α is the thermal accommodation coefficient, often assumed to be the same as σ.
- Pr is the Prandtl number, which can be computed as $\text{Pr} = \frac{c_p \mu}{\kappa}$, taking κ as the thermal conductivity.

The idea behind this project is to build the foundation upon which these new boundary conditions can be incorporated into PHASTA, and then see how the results compare to the original code.

3 Formulation of the finite element method for CFD

PHASTA solves the compressible Navier–Stokes equations. Starting with the conservative form:

$$\boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i} = \boldsymbol{\mathcal{F}} \tag{4}$$

where $\boldsymbol{U}$ is the solution vector, $\boldsymbol{F}_i$ is the flux vector, and $\boldsymbol{\mathcal{F}}$ represents the source term, such that:

$$\boldsymbol{U} = \begin{Bmatrix} U_1 \\ U_{j+1} \\ U_5 \end{Bmatrix} = \begin{Bmatrix} \rho \\ \rho u_j \\ \rho e_{\text{tot}} \end{Bmatrix} \qquad \boldsymbol{F}_i = \underbrace{\begin{Bmatrix} \rho u_i \\ \rho u_i u_j \\ \rho u_i e_{\text{tot}} \end{Bmatrix} + \begin{Bmatrix} 0 \\ p\,\delta_{ij} \\ u_i p \end{Bmatrix}}_{\boldsymbol{F}_i^{\text{adv}}} + \underbrace{\begin{Bmatrix} 0 \\ \tau_{ij} \\ u_j \tau_{ij} \end{Bmatrix} + \begin{Bmatrix} 0 \\ 0_j \\ q_i \end{Bmatrix}}_{\boldsymbol{F}_i^{\text{diff}}} \tag{5}$$

with $e_{\text{tot}} = e + \frac{1}{2} u_l u_l$, the total energy, $e = c_v T$, the internal energy, $\tau_{ij} = \mu \left(u_{i,j} + u_{j,i} \right) + \lambda u_{l,l} \delta_{ij}$, the viscous stress tensor, and $q_i = -\kappa T_{,i}$, the heat flux according to Fourier's law.

Since the code uses the finite element method, the next step is to find the weak form of Equation (4):

$$\int_\Omega \left[\boldsymbol{W} \cdot \boldsymbol{U}_{,t} - \boldsymbol{W}_{,i} \cdot \boldsymbol{F}_i - \boldsymbol{W} \cdot \boldsymbol{\mathcal{F}} \right] + \int_\Gamma \boldsymbol{W} \cdot \boldsymbol{F}_i n_i \, d\Gamma = 0 \tag{6}$$

which we get after moving everything to one side (residual form), integrating by parts (over the whole domain Ω and its boundary Γ), and multiplying by the weight space functions.

Since we seek a discrete form of the equations with n_n nodes in the domain, in the Galerkin formulation we use the same shape functions (or basis functions, typically piecewise polynomials) to interpolate both the solution and the weights from nodal values (at $\boldsymbol{x}$ locations) as follows:

$$\boldsymbol{Y}(\boldsymbol{x}) = \sum_{A=1}^{n_n} N_A(\boldsymbol{x}) \, \boldsymbol{Y}_A \qquad \boldsymbol{Y}_{,i}(\boldsymbol{x}) = \sum_{A=1}^{n_n} N_{A,i}(\boldsymbol{x}) \, \boldsymbol{Y}_A \qquad \boldsymbol{Y}_{,t}(\boldsymbol{x}) = \sum_{A=1}^{n_n} N_A(\boldsymbol{x}) \, \boldsymbol{Y}_{A,t} \tag{7a}$$

$$\boldsymbol{W}(\boldsymbol{x}) = \sum_{B=1}^{n_n} N_B(\boldsymbol{x}) \, \boldsymbol{W}_B \qquad \boldsymbol{W}_{,i}(\boldsymbol{x}) = \sum_{B=1}^{n_n} N_{B,i}(\boldsymbol{x}) \, \boldsymbol{W}_B \tag{7b}$$

where we have replaced $\boldsymbol{U}$, the conservative variables, with the array of primitive variables:

$$\boldsymbol{Y} = \begin{Bmatrix} Y_1 \\ Y_{j+1} \\ Y_5 \end{Bmatrix} = \begin{Bmatrix} p \\ u_j \\ T \end{Bmatrix} \tag{8}$$

In order to evaluate the global integral in the residual above (6), we will break it down into a sum of local or element-level integrals, where the solution is now expressed as:

$$\boldsymbol{Y}(\boldsymbol{x}) = \sum_{a=1}^{n_{en}} N_a(\boldsymbol{x}) \, \boldsymbol{Y}_a \qquad \boldsymbol{Y}_{,i}(\boldsymbol{x}) = \sum_{a=1}^{n_{en}} N_{a,i}(\boldsymbol{x}) \, \boldsymbol{Y}_a \qquad \boldsymbol{Y}_{,t}(\boldsymbol{x}) = \sum_{a=1}^{n_{en}} N_a(\boldsymbol{x}) \, \boldsymbol{Y}_{a,t} \tag{9}$$

where n_{en} counts the number of nodes on each element (a).It is worth noting that instead of computing the shape functions in physical or real space ($\boldsymbol{x}$ coordinates), a better approach is to localize them to the parent or parametric space:

$$\boldsymbol{x}(\boldsymbol{\xi}) = \sum_{a=1}^{n_{en}} N_a(\boldsymbol{\xi}) \, \boldsymbol{x}_a^e \qquad \boldsymbol{x}_{\boldsymbol{\xi}} = \sum_{a=1}^{n_{en}} N_{a,\boldsymbol{\xi}}(\boldsymbol{\xi}) \, \boldsymbol{x}_a^e \tag{10}$$

where $\boldsymbol{\xi}$ is a reference coordinate that takes values the same value for every element (from now on indicated by superscript e).

The real space gradient of the shape functions can now be evaluated with the chain rule:

$$N_{a,i}(\boldsymbol{\xi}) = N_{a,\xi_j}\xi_{j,i} \tag{11}$$

Putting all these results together, we can rewrite the residual equation (6) at the element level:

$$\boldsymbol{G}_b^e \equiv \int_{\square} \left\{ N_b(\boldsymbol{\xi}) \left[\boldsymbol{A}_0 \sum_{a=1}^{n_{en}} N_a(\boldsymbol{\xi})\, \boldsymbol{Y}_{a,t} - \boldsymbol{\mathcal{F}} \right] - N_{b,i}(\boldsymbol{\xi})\, \boldsymbol{F}_i \right\} D(\boldsymbol{\xi})\, d\square + \int_{\square_\Gamma} N_b(\boldsymbol{\xi})\, \boldsymbol{F}_i n_i\, D_\Gamma(\boldsymbol{\xi})\, d\square_\Gamma \tag{12}$$

where $\boldsymbol{A}_0$ relates the primitive and conservative variables, $D(\boldsymbol{\xi}) \equiv \det\left(\boldsymbol{x}_{\boldsymbol{\xi}}\right)$ is the Jacobian of the coordinate transformation, and the symbol $\square$ is used to denote the reference space.

Since we want to evaluate the integrals in Equation (12) in an efficient manner, we will make use of Gauss quadrature in the parent domain:

$$\int_{\square} f(\boldsymbol{\xi})\, d\square \approx \sum_{k=1}^{n_{\text{int}}} f(\boldsymbol{\xi}^k)\, w^k \tag{13}$$

where $\boldsymbol{\xi}^k$ and w^k are, respectively, the position and weight —in parametric coordinates— of the n_{int} quadrature points, which can change depending on the order of integration.

Finally, the n_e elemental residuals computed with (12) are assembled together to recover the global residual we seek:

$$\boldsymbol{G}_B = \mathop{\mathbf{A}}_{e=1}^{n_e} \boldsymbol{G}_b^e = 0 \tag{14}$$

The Galerkin method that has been presented so far is shown to be unstable for convection-dominated flows. Hence, we add an additional stabilization term to the residual:

$$\hat{\boldsymbol{G}}_b^e = \underbrace{\boldsymbol{G}_b^e}_{\text{Galerkin}} + \underbrace{\int_{\square} \hat{\mathcal{L}}\, N_b(\boldsymbol{\xi}) \cdot \boldsymbol{\tau} \left[\mathcal{L}\boldsymbol{Y} - \boldsymbol{\mathcal{F}} \right] D(\boldsymbol{\xi})\, d\square}_{\text{Stabilization}} \tag{15}$$

where the following "least squares" linear operator is defined:

$$\mathcal{L} \equiv \boldsymbol{A}_0 \frac{\partial}{\partial t} + \boldsymbol{A}_i \frac{\partial}{\partial x_i} - \frac{\partial}{\partial x_i} \left[\boldsymbol{K}_{ij} \frac{\partial}{\partial x_j} \right] \tag{16}$$

with the matrices above being such that the following relations hold true:

$$\boldsymbol{A}_0 \equiv \boldsymbol{U}_{,\boldsymbol{Y}} \qquad A_i \equiv \boldsymbol{F}_{i,\boldsymbol{Y}}^{\text{adv}} \qquad \boldsymbol{F}_i^{\text{diff}} = -\boldsymbol{K}_{ij}\boldsymbol{Y}_{,j} \tag{17}$$

and hence when the operator is applied to the solution vector, the result is simply the original LHS of the Navier–Stokes strong form (4):

$$\mathcal{L}\boldsymbol{Y} = \boldsymbol{A}_0\boldsymbol{Y}_{,t} + \boldsymbol{A}_i\boldsymbol{Y}_{,i} - \left[\boldsymbol{K}_{ij}\boldsymbol{Y}_{,j}\right]_{,i} = \boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i}^{\text{adv}} + \boldsymbol{F}_{i,i}^{\text{diff}} \tag{18}$$

showing that the added stabilization term is just a residual that disappears when the solution converges.

Additionally, the operator $\hat{\mathcal{L}}$, which together with the $\boldsymbol{\tau}$ matrix of timescales act as a weight to this additional residual, can take different forms. In Streamline Upwind Petrov–Galerkin (SUPG), the scheme used by PHASTA, it is defined to only consider advection:

$$\hat{\mathcal{L}} \equiv \boldsymbol{A}_i \frac{\partial}{\partial x_i} \tag{19}$$

At this point, the FEM spatial discretization has allowed us to transform the original partial differential equation (4) into the residual $\hat{\boldsymbol{G}}_B(\boldsymbol{Y}, \boldsymbol{Y}_{,t})$ of a nonlinear system of ordinary differential equations. Next, we will use a generealized-α time integrator to turn this into a nonlinear system of algebraic equations:

$$\overset{n+1}{\boldsymbol{Y}}{}_A^{(i)} = \overset{n}{\boldsymbol{Y}}_A + \Delta t\, \overset{n}{\boldsymbol{Y}}_{A,t} + \Delta t \left[\overset{n+1}{\boldsymbol{Y}}{}_{A,t}^{(i)} - \overset{n}{\boldsymbol{Y}}_{A,t} \right] \qquad \forall\ A \tag{20a}$$

$$\hat{\boldsymbol{G}}_B \left(\overset{n+\alpha_m}{\boldsymbol{Y}}{}_{A,t}^{(i)}, \overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)} \right) = 0 \qquad \forall\ B \tag{20b}$$

$$\overset{n+\alpha_m}{\boldsymbol{Y}}{}_{A,t}^{(i)} = \overset{n}{\boldsymbol{Y}}_{A,t} + \alpha_m \left[\overset{n+1}{\boldsymbol{Y}}{}_{A,t}^{(i)} - \overset{n}{\boldsymbol{Y}}_{A,t} \right] \qquad \forall\ A \tag{20c}$$

$$\overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)} = \overset{n}{\boldsymbol{Y}}_A + \alpha_f \left[\overset{n+1}{\boldsymbol{Y}}{}_A^{(i)} - \overset{n}{\boldsymbol{Y}}_A \right] \qquad \forall\ A \tag{20d}$$

where i is the time iteration and counter, and the parameters α_f and α_m are not necessarily equal. Given a ρ_∞ amplification factor, for second order accuracy in time these can be selected as follows:

$$\alpha_m = \frac{1}{2}\left[\frac{3-\rho_\infty}{1+\rho_\infty}\right] \qquad \alpha_f = \frac{1}{1+\rho_\infty} \qquad \gamma = \frac{1}{2} + \alpha_m - \alpha_f \tag{21}$$

Newton's method is used to linearize the algebraic system of equations such that in each iteration we can find the difference in our solution needed for the residual to become zero. We choose to linearize about $\overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)}$, resulting in the followin linear system of equations:

$$\underbrace{\hat{\boldsymbol{G}}_B \left(\overset{n+\alpha_m}{\boldsymbol{Y}}{}_{A,t}^{(i)}, \overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)} \right)}_{\boldsymbol{R}_B} + \underbrace{\sum_{A=1}^{n_n} \frac{\partial \overset{\text{LHS}}{\boldsymbol{G}}_B}{\partial \overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)}}}_{\boldsymbol{M}_{BA}} \underbrace{\Delta \overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)}}_{\Delta \boldsymbol{Y}_A} = \boldsymbol{0} \tag{22}$$

where $\boldsymbol{R}_B$ is the RHS residual, $\boldsymbol{M}_{BA}$ is called the LHS tangent or mass matrix, and $\Delta \overset{n+\alpha_f}{\boldsymbol{Y}}{}_A^{(i)}$ is the difference we seek. Note that the tangent matrix is computed at the element level and then assembled globally.

Furthermore, the LHS residual is computed differently for better convergence, reversing integration by parts on the advective term and dropping the boundary integral:

$$\overset{\text{LHS}}{\boldsymbol{G}}_b = \int_\square \left\{ N_b \left[\boldsymbol{A}_0\, \overset{n+\alpha_m}{\boldsymbol{Y}}{}_{a,t}^{(i)} + \boldsymbol{A}_i\, \overset{n+\alpha_f}{\boldsymbol{Y}}{}_{a,i}^{(i)} - \boldsymbol{\mathcal{F}} \right] + N_{b,i}\, \boldsymbol{K}_{ij}\, \overset{n+\alpha_f}{\boldsymbol{Y}}{}_{a,j}^{(i)} + \hat{\mathcal{L}}\, N_b \cdot \boldsymbol{\tau} \left[\boldsymbol{A}_0\, \overset{n+\alpha_m}{\boldsymbol{Y}}{}_{a,t}^{(i)} + \boldsymbol{A}_i\, \overset{n+\alpha_f}{\boldsymbol{Y}}{}_{a,i}^{(i)} - \boldsymbol{\mathcal{F}} \right] \right\} D(\boldsymbol{\xi})\, d\square \tag{23}$$

Using a predictor/multi-correct iteration process, the following algorithm is applied inside a loop over timesteps:

1. Set initial values for $\overset{n+1}{\boldsymbol{Y}_A^{(i)}}$ with data from the previous timestep or initial conditions.
2. Solve Equation (20a) to get a consistent prediction of $\overset{n+1}{\boldsymbol{Y}_{A,t}^{(i)}}$.
3. Interpolate the solution vector $\overset{n+\alpha_f}{\boldsymbol{Y}_A^{(i)}}$ and its derivative $\overset{n+\alpha_m}{\boldsymbol{Y}_{A,t}^{(i)}}$ with the selected parameters using (20c) and (20d).
4. Compute the residual $\hat{\boldsymbol{G}}_B$ with Equation (20b).
5. Using Newton's method (22), find the difference $\Delta\overset{n+\alpha_f}{\boldsymbol{Y}_A^{(i)}}$ that makes the residual zero.
6. Update our solution vectors: $\overset{n+\alpha_f}{\boldsymbol{Y}_A^{(i+1)}} = \overset{n+\alpha_f}{\boldsymbol{Y}_A^{(i)}} + \Delta\overset{n+\alpha_f}{\boldsymbol{Y}_A^{(i)}}$, and find $\overset{n+\alpha_m}{\boldsymbol{Y}_{A,t}^{(i+1)}}$ with (20a) and (20d).
7. Keep iterating 4–6 until the residual drops below a specified tolerance, the solution difference goes below a secondary tolerance, or the maximum number of iterations per timestep is reached.
8. Move to the next timestep, starting back again at 1.

The compressible version of PHASTA has multiple ways to approach the solution of the linear system of equations presented in Equation (22), all of which are based on the iterative method known as generalized minimal residual (GMRES), since for large problems direct solvers are not possible.

Before we discuss the possible approaches, we must first introduce preconditioning. CFD problems are notoriously ill-conditioned, an issue that originates with the scales of the different terms in the Navier–Stokes equations (p, u, and T all have different orders of magnitude). In PHASTA, before solving the system we will make use of symmetric block-diagonal preconditioning. Start by defining:

$$\boldsymbol{B} \equiv \operatorname{diag}\left(\boldsymbol{M}_{BA}\right) \tag{24}$$

and then, using the Cholesky method, we can decompose the above into the product of a lower triangular matrix $\boldsymbol{L}$ and an upper triangular matrix $\boldsymbol{U}$. With these, we see better results by modifying the the original system of equations as follows:

$$\underbrace{\boldsymbol{L}^{-1}\boldsymbol{M}_{BA}\boldsymbol{U}^{-1}}_{\boldsymbol{A}}\ \underbrace{\boldsymbol{U}\Delta\boldsymbol{Y}_A}_{\boldsymbol{x}} = \underbrace{-\boldsymbol{L}^{-1}\boldsymbol{R}_B}_{\boldsymbol{b}} \tag{25}$$

and after finding $\boldsymbol{x}$, we can recover the original solution we seek as $\Delta\boldsymbol{Y}_A = \boldsymbol{U}^{-1}\boldsymbol{x}$.

Iterative solvers approximate the solution $\boldsymbol{x}$ as $\boldsymbol{x}_0+\boldsymbol{z}$, where $\boldsymbol{x}_0$ is an initial guess and $\boldsymbol{z}$ is iteratively constructed such that the norm $\left\|\boldsymbol{A}\left(\boldsymbol{x}_0+\boldsymbol{z}\right)-\boldsymbol{b}\right\|$ is minimized. GMRES considers $\boldsymbol{z}$ to be a member —that is, a linear combination— of the Krylov subspace:

$$\mathcal{K} = \operatorname{span}\left\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^k\boldsymbol{r}_0\right\} \tag{26}$$

with $\boldsymbol{r}_0 = \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b}$, the error of the initial guess.

Note that each subsequent Krylov vector is the product of the previous one:

$$\boldsymbol{v}_k = \boldsymbol{A}^{k-1}\boldsymbol{r}_0 = \boldsymbol{A}\boldsymbol{v}_{k-1} \equiv \boldsymbol{A}\boldsymbol{p} = \boldsymbol{q} \tag{27}$$

and is then made orthonormal with respect to all the previous ones.

Recall that so far we have not computed the global $\boldsymbol{A}$ (or $\boldsymbol{M}_{BA}$) matrix. It is not practical, but we still need to compute the matrix-vector product $\boldsymbol{Ap}$. As mentioned earlier, in PHASTA there are three different approaches:

1. Elemen-by-Element GMRES: instead of assembling the elemental $\boldsymbol{A}^e$ matrices, we localize the global Krylov vectors:

$$\boldsymbol{p}^e = \mathop{\mathsf{L}}_{e=1}^{n_e} \boldsymbol{p} \tag{28}$$

   and then perform the matrix-vector product at the element level, and assemble the result:

$$\boldsymbol{q}^e = \boldsymbol{A}^e \boldsymbol{p}^e \implies \boldsymbol{q} = \mathop{\mathsf{A}}_{e=1}^{n_e} \boldsymbol{q}^e \tag{29}$$

2. Matrix-Free GMRES: recall that $\boldsymbol{A} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{Y}}$ (dropped indices), and thus can be approximated as:

$$\boldsymbol{A} \approx \frac{\boldsymbol{G}(\boldsymbol{Y} + \varepsilon \boldsymbol{v}) - \boldsymbol{G}(\boldsymbol{Y})}{\varepsilon \boldsymbol{v}} \tag{30}$$

   where ε is a perturbation with an arbitrary variable $\boldsymbol{v}$ that represents an increment $\Delta \boldsymbol{Y}$. Since what we seek is a matrix-vector product, replacing $\boldsymbol{v}$ above by $\boldsymbol{p}$, we have:

$$\boldsymbol{Ap} \approx \frac{\boldsymbol{G}(\boldsymbol{Y} + \varepsilon \boldsymbol{p}) - \boldsymbol{G}(\boldsymbol{Y})}{\varepsilon \cancel{\boldsymbol{p}}} \cancel{\boldsymbol{p}} \tag{31}$$

   and we see that there is no need to form the $\boldsymbol{A}$ matrix, greatly reducing the memory use required.

3. Sparse GMRES: in this case, the $\boldsymbol{A}$ matrix is sparsely assembled, meaning that we only store the nonzeros in a stacked array. In this way, when performing the $\boldsymbol{Ap}$ product, the number of floating point operations is minimized.

Finally, we need to discuss how boundary conditions are imposed. These can be of either essential (Dirichlet) or natural (Neumann) types.

When dealing with essential boundary conditions, the value of the unknown is directly specified on the boundary Γ_g. PHASTA supports the following list of v^g variables to be prescribed at any point:

$$\begin{Bmatrix} p^g \\ u_r^g \\ u_s^g \\ u_t^g \\ T^g \\ \rho^g \end{Bmatrix} = \begin{Bmatrix} Y_1 \\ c_i^r Y_{i+1} \\ c_i^s Y_{i+1} \\ c_i^t Y_{i+1} \\ Y_5 \\ \frac{Y_1}{RY_5} \end{Bmatrix} \tag{32}$$

where c_i^r, c_i^s, and c_i^t represent the direction cosines of the velocity vector $\left(u_r^g, u_s^g, u_t^g\right)$, components usually considered to be aligned with the wall (non-cartesian). Rearranging Equation (32), we can find the $\hat{\boldsymbol{q}}$ vector, which modified the values of our solution vector $\boldsymbol{Y}$ as a function of the Dirichlet boundary conditions where necessary:

$$\begin{Bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_1 \end{Bmatrix} = \begin{Bmatrix} p^g \\ \frac{1}{c_1^r}\left(u_r^g - c_2^r Y_3 - c_3^r Y_4\right) \\ \frac{1}{c_2^s}\left(u_s^g - c_1^s Y_2 - c_3^s Y_4\right) \\ \frac{1}{c_3^t}\left(u_t^g - c_1^t Y_2 - c_2^t Y_3\right) \\ T^g \\ \rho^g R Y_5 \end{Bmatrix} \equiv \hat{\boldsymbol{q}} \tag{33}$$

PHASTA also has support for periodic boundary conditions, which are similar to Dirichlet in the sense that they directly modify the value of the solution vector. In this case, though, the value is taken to be the same as the solution in a partner node.

Besides modifying the solution vector as shown above, in order to impose essential boundary conditions, we will also need to constraint the weight space such that it fulfills the relations above. For each possible variable k, a new weight is derived as follows:

$$W'_k = \frac{\partial \hat{q}_k}{\partial Y_j} W_j \tag{34}$$

In PHASTA, this is implemented by modifying the corresponding $\boldsymbol{M}_{BA}$ matrix and $\boldsymbol{R}_B$ vector from Equation (22) with a matrix $\boldsymbol{S}_v$ that relates the old weights with the new ones, $\boldsymbol{W'} = \boldsymbol{S}_v \boldsymbol{W}$ for each possible variable v^g. The new system of equations becomes:

$$\underbrace{\boldsymbol{S}_v^T \boldsymbol{M}_{BA} \boldsymbol{S}_v}_{\tilde{\boldsymbol{M}}_{BA}} \Delta \boldsymbol{Y}_A = - \underbrace{\boldsymbol{S}_v^T \boldsymbol{R}_B}_{\tilde{\boldsymbol{R}}_B} \tag{35}$$

which, again, is modified at the element level and then assembled to obtain the global matrix.

As a notable example, when setting the three components of velocity at a given point, the transformation of the weight space is:

$$\boldsymbol{S}_{rst} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{36}$$

On the other hand, for natural boundary conditions, the flux of the variable is prescribed on a boundary Γ_h. In this case, the constraints are imposed during the integration of the boundary elements when computing the residual (12), as follows:

$$\begin{aligned}
\int_{\square_\Gamma} N_b \left[\boldsymbol{F}_i^{\text{adv}} + \boldsymbol{F}_i^{\text{diff}} \right] n_i \, D \, d\square_\Gamma &= \int_{\square_\Gamma} N_b \left[u_n \boldsymbol{U} + p \begin{Bmatrix} 0 \\ \delta_{jn} \\ u_n \end{Bmatrix} - \begin{Bmatrix} 0 \\ \tau_{jn} \\ \tau_{jn} u_j \end{Bmatrix} + \begin{Bmatrix} 0 \\ 0_j \\ q_n \end{Bmatrix} \right] D_\Gamma \, d\square_\Gamma \\
&= \int_{\square_{\Gamma_h^m}} N_b \, h^m \, \boldsymbol{U} \, D_\Gamma \, d\square_\Gamma + \int_{\square_{\Gamma - \Gamma_h^m}} N_b \, u_n \, \boldsymbol{U} \, D_\Gamma \, d\square_\Gamma \\
&+ \int_{\square_{\Gamma_h^p}} N_b \, h^p \begin{Bmatrix} 0 \\ \delta_{jn} \\ u_n \end{Bmatrix} D_\Gamma \, d\square_\Gamma + \int_{\square_{\Gamma - \Gamma_h^p}} N_b \, p \begin{Bmatrix} 0 \\ \delta_{jn} \\ u_n \end{Bmatrix} D_\Gamma \, d\square_\Gamma \\
&- \int_{\square_{\Gamma_h^v}} N_b \begin{Bmatrix} 0 \\ h_j^v \\ h_j^v u_j \end{Bmatrix} D_\Gamma \, d\square_\Gamma - \int_{\square_{\Gamma - \Gamma_h^v}} N_b \begin{Bmatrix} 0 \\ \tau_{jn} \\ \tau_{jn} u_j \end{Bmatrix} D_\Gamma \, d\square_\Gamma \\
&+ \int_{\square_{\Gamma_h^h}} N_b \begin{Bmatrix} 0 \\ 0_j \\ h_h^h \end{Bmatrix} D_\Gamma \, d\square_\Gamma + \int_{\square_{\Gamma - \Gamma_h^h}} N_b \begin{Bmatrix} 0 \\ 0_j \\ q_n \end{Bmatrix} D_\Gamma \, d\square_\Gamma
\end{aligned} \tag{37}$$

where h^k are the prescribed flux values on the boundary Γ_h^k, and then we have a floating flux on $\Gamma - \Gamma_h^k$, which is computed using the current value of the solution.

4 PHASTA codebase

In this section, we describe how the PHASTA code (compressible version) is implemented, including the main variables and functions used and how they relate to the FEM theory presented so far.

It is worth noting that, since PHASTA is designed to run in parallel, the whole domain is first divided into parts, and each of these is then further divided into blocks of elements. These blocks have a fixed size designed to optimize cache performance in Fortran implicit loops, which are used throughout the code.

4.1 Summary of variables

First of all, as many of the variables defined in PHASTA are vectors or matrices, below is a reference of all the dimensions mentioned throughout this document:

- `nsd` is the number of space dimensions (generally 3).
- `npro` is the number of elements to process in a single block call (n_e in the notes).
- `nenl` is the number of vertices per element.
- `nshl` is the local number of shape functions per element (local nodes, $a = 1 \dots n_{en}$ in the notes).
- `nshg` the global number of shape functions (total number of nodes, $A = 1 \dots n_n$ in the notes).
- `ngauss` is the number of Gauss quadrature integration points in an element (n_{int} in the notes).
- `nflow` is the number of flow variables in the solution vector (5 in the compressible 3D case).
- `ndof` is the number of degrees of freedom per node (basically, `ndof = nflow + scalars`).
- `nedof = nflow * nshl` is the total number of degrees of freedom at each element node.
- `nnz_tot` is the total number of nonzeros in the LHS matrix.
- `nshlb` is like `nshl`, but for the boundary elements.
- `ngaussb` is like `ngauss`, but for the boundary elements.
- `nenbl` is the number of nodes in a boundary face for the boundary elements.
- `Kspace` is the dimension of the Krylov subspace.
- `nlwork` is the size of the array local to the processor.

Some of these variables are stored per each `iblk` block of elements for ease of access in the `lcblk` array, as detailed below:

- `lcblk(1,iblk) = iel` is the element counter, and is used to compute the number of elements in the block as `npro = lcblk(1,iblk+1) - iel`.
- `lcblk(3,iblk) = lcsyst` represents the element topology in the block (`1` for tetrahedrons, `2` for hexahedrons, `3` for wedges, `5` for pyramids).
- `lcblk(4,iblk) = iorder` is the element order
- `lcblk(5,iblk) = nenl`.
- `lcblk(10,iblk) = nshl`.

Table 1 summarizes the most important variables used throughout the PHASTA code and their correspondence with the theory covered in section 3.

Table 1: List of the main variables (and their dimensions) used in PHASTA

In code	In the notes	Description
`y(nshg,ndof)`	$\overset{n+\alpha_f}{\boldsymbol{Y}^{(i)}}$	Solution array (primitive global variables) at current timestep. Their order is $\{u_j, p, T\}$.
`x(nshg,nsd)`	$\boldsymbol{x}$	Global node (vertex) coordinates.
`ac(nshg,ndof)`	$\overset{n+\alpha_m}{\boldsymbol{Y}^{(i)}_{,t}}$	Acceleration array (time derivative of solution vector).
`yold(nshg,ndof)`	$\overset{n}{\boldsymbol{Y}}$	Solution vector at previous (or initial) timestep.
`acold(nshg,ndof)`	$\overset{n}{\boldsymbol{Y}_{,t}}$	Acceleration vector at previous (or initial) timestep.
`shp(nshl,ngauss)`	N_a	Element shape functions evaluated at the quadrature points. In the code, this is sometimes referred to as `shape`, and is then reshaped to be of the form `shp(npro,nshl)` for each Gauss point.
`shgl(nsd,nshl,ngauss)`	$N_{a,\boldsymbol{\xi}}$	Element local shape function gradients (in the parent domain) evaluated at the quadrature points. In the code, this is sometimes referred to as `shdrv`, and is then reshaped to be of the form `shgl(npro,nsd,nshl)` for each Gauss point.
`shg(npro,nshl,nsd)`	$N_{a,i}$	Global gradient of element shape functions.
`dxdxi(npro,nsd,nsd)`	$\boldsymbol{x}_{,\boldsymbol{\xi}}$	Jacobian of the map from real domain to parent domain.
`dxidx(npro,nsd,nsd)`	$\boldsymbol{\xi}_{,i} = \xi_{j,i}$	Inverse of the above, the metric tensor that maps the parent domain to the real domain.
`WdetJ(npro)`	$w^k D(\boldsymbol{\xi}^k)$	Weighted Jacobian for spatial (quadrature) integration.
`ien(npro,nshl)`	–	Nodal connectivity matrix that given `element #, local node #` returns the `global node #`.
`yl(npro,nshl,nflow)`	$\overset{n+\alpha_f}{\boldsymbol{Y}_a}$	Solution array (primitive variables) at current timestep. Their order is $\{p, u_j, T\}$.
`ycl(npro,nshl,ndof)`	–	Corresponds to a vector with $\overset{n+\alpha_f}{\boldsymbol{Y}_a}$ and scalars.
`acl(npro,nshl,ndof)`	$\overset{n+\alpha_m}{\boldsymbol{Y}_{a,t}}$	Acceleration vector at current timestep.
`xl(npro,nshl,nsd)`	$\boldsymbol{x}^e_a$	Local node coordinates.
`dui(npro,nflow)`	$\overset{n}{\Delta \boldsymbol{U}}$	Delta of conservative variables at previous timestep.
`pres`	p	Pressure at the quadrature point.
`u1`, `u2`, `u3`	u_j	Velocity (in the j direction) at the quadrature point.

In code	In the notes	Description
`T`	T	Temperature at the quadrature point.
`g1yi(npro,nflow)`, `g2yi(npro,nflow)`, `g3yi(npro,nflow)`	$\boldsymbol{Y}_{,i}$	Gradient of the solution vector.
`A0(npro,nflow,nflow)`	$\boldsymbol{A}_0 = \boldsymbol{U}_{,\boldsymbol{Y}}$	Matrix that maps conservative variables to primitive variables.
`A1(npro,nflow,nflow)`, `A2(npro,nflow,nflow)`, `A3(npro,nflow,nflow)`	$\boldsymbol{A}_i = \boldsymbol{F}^{\text{adv}}_{i,\boldsymbol{Y}}$	Matrices that relate the divergence of the advective flux such that $\boldsymbol{F}^{\text{adv}}_{i,i} = \boldsymbol{A}_i \boldsymbol{Y}_{,i}$.
`stiff(npro,nsd*nflow, nsd*nflow)`	$\boldsymbol{K}_{ij}$	Diffusive tangent matrix such that the viscous flux can be computed as $\boldsymbol{F}^{\text{diff}}_i = -\boldsymbol{K}_{ij}\boldsymbol{Y}_{,j}$.
`tau(npro,3)`	$\boldsymbol{\tau}$	Stabilization matrices (continuity, momentum, energy).
`src(npro,nflow)`	$\mathbf{F}$	Body force vector.
`ri(npro,nflow*(nsd+1)`	–	Partial residual. `ri(:,1:15` represents $\boldsymbol{F}_i = \boldsymbol{F}^{\text{adv}}_i + \boldsymbol{F}^{\text{diff}}_i$, while `ri(16:20)` collects $\boldsymbol{A}_0\boldsymbol{Y}_{,t} - \boldsymbol{\mathcal{F}}$.
`rLyi(npro,nflow)`	$\mathcal{L}\boldsymbol{Y} - \boldsymbol{\mathcal{F}}$	Least-squares residual vector, where the operator $\mathcal{L} \equiv \boldsymbol{A}_0 \frac{\partial}{\partial t} + \boldsymbol{A}_i \frac{\partial}{\partial x_i} - \frac{\partial}{\partial x_i}\left(\boldsymbol{K}_{ij}\frac{\partial}{\partial x_j}\right)$ is used.
`rl(npro,nshl,nflow)`	$\hat{\boldsymbol{G}}^e_b$	Element-level residual.
`res(nshg,nflow)`	$\hat{\boldsymbol{G}}_B$	Global residual.
`EGmass(npro,nedof,nedof)`	$\boldsymbol{M}_{ba} = \hat{\boldsymbol{G}}_{b,\boldsymbol{Y}_a}$	Element-level LHS tangent mass matrix.
`lhsK(nflow*nflow,nnz_tot)`	$\boldsymbol{M}_{BA}$	Sparsely assembled LHS mass matrix.
`iBC(nshg)`	–	Essential boundary condition binary code. The code currently goes up to 2^{13}, where each digit represents whether a condition is prescribed or not (e.g., 10101010101010), and they are ordered as follows: density (2^0), temperature (2^1), pressure (2^2), velocity components (x_1, 2^3; x_2, 2^4; x_3, 2^5), scalars (2^6, 2^7, 2^8, 2^9), periodicity (2^{10}), SPEBC (2^{11}), axisymmetric (2^{12}), and wall deformation (2^{13}).
`BC(nshg,ndof+1)`	$\hat{\boldsymbol{q}} = f(p^g, u^g_j, T^g, \rho^g)$	Essential boundary condition constraint parameters. Includes prescribed values for density, pressure, temperature, and velocity, as well as the corresponding direction consine terms.

In code	In the notes	Description
`iBCB(npro,2)`	–	Natural boundary condition code, equivalent to `iBC`. In this case, the order of `iBCB(:,1)` is: convective flux (2^0), pressure flux (2^1), viscous flux (2^2), heat flux (2^3), turbulence wall (2^4), and the next are scalar fluxes. `iBCB(:,2)` is the `srfID` given by the user for which the integrated fluxes will be colleted.
`BCB(npro,nshlb,ndof+1)`	$h^k(\boldsymbol{x})$	Natural boundary condition prescribed fluxes.
`xlb(npro,nenl,nsd)`	x_b^e	Like `xl`, but for boundary elements.
`shpb(nshl,ngaussb)`	N_b	Like `shp`, but for boundary elements. Reshaped to `shpb(npro,nshl)` for each Gauss point.
`shglb(nsd,nshl,ngaussb)`	$N_{b,\boldsymbol{\xi}}$	Like `shgl`, but for boundary elements. Reshaped to `shglb(npro,nsd,nshl)` for each Gauss point.
`dxdxib(npro,nsd,nsd)`	$\boldsymbol{x}_{,\boldsymbol{\xi}}$	Like `dxdxi`, but for boundary elements.
`dxidxb(npro,nsd,nsd)`	$\boldsymbol{\xi}_{,i} = \xi_{j,i}$	Like `dxidx`, but for boundary elements.
`WdetJb(npro)`	$w^k D_\Gamma(\boldsymbol{\xi}^k)$	Like `WdetJ`, but for boundary elements.
`bnorm(npro,nsd)`	$\hat{\boldsymbol{n}}$	Outward normal vector (to the surface).
`rou(npro)`	$h^m \boldsymbol{U}$	Boundary condition mass flux.
`p(npro)`	$h^p \begin{Bmatrix} 0 \\ \delta_{jn} \\ u_n \end{Bmatrix}$	Boundary condition natural pressure.
`tau1n(npro)`, `tau2n(npro)`, `tau3n(npro)`	$\begin{Bmatrix} 0 \\ h_j^v \\ h_j^v u_j \end{Bmatrix}$	Boundary condition viscous flux.
`heat(npro)`	$\begin{Bmatrix} 0 \\ 0_j \\ h^h \end{Bmatrix}$	Boundary condition heat flux.
`F1(npro)`, `F2(npro)`, `F3(npro)`, `F4(npro)`, `F5(npro)`	$\boldsymbol{F}_i n_i$	Total flux normal to the boundary.
`Force(3)`	–	3 components of the integrated aerodynamic forces on the body.
`Hflux`	–	Total integrated heat flux on the body.
`iper(nshg)`	–	Partners of periodic boundary conditions,relates the master node to the slave node so that they end up having the same values.
`BDiag(nshg,nflow,nflow)`	$\boldsymbol{B}$	Block-diagonal preconditioner matrix.
`Dy(nshg,nflow)`	$\Delta \boldsymbol{Y}_A^{(i)}$	Difference in solution vector.

In code	In the notes	Description
`uBrg(nsh,nflow,Kspace+1)`	$\boldsymbol{v}_k$	Krylov vector.
`HBrg(Kspace+1,Kspace)`	–	Upper Hessenberg matrix.
`eBrg(Kspace+1)`	$\left\|\boldsymbol{A}\left(\boldsymbol{x}_0+\boldsymbol{z}\right)-\boldsymbol{b}\right\|$	Error of the minimization problem in GMRES.
`yBrg(Kspace)`	$\boldsymbol{x}_0+\boldsymbol{z}$	Solution of the minimization problem in GMRES.
`row(nnz*nshg)`	–	Helper array that collects in a stacked structure all the nonzeros in a given row (according to the links between nodes).
`col(nshg+1)`	–	Helper array that determines what on what index of `row` each new row starts. Its size is `nshg+1` to indicate the end point.
`ilwork(nlwork)`	–	Local processor MPI communication array.

4.2 Summary of functions

Figure 2 shows a general outline of the main routines called when running the PHASTA code and how they are connected. Then, each of these functions is described in more detail. Note that all file paths in this document are referred to the *phSolver/* parent folder. Also note that subroutines called multiple times are only shown the first time they appear.

This description further assumes the PHASTA is run with the sparse solver, the default option and the one used in this work, and thus it calls the `SolGMRs` and `ElmGMRs` routines. Switching to the element-by-element GMRES solver would result in calls to `SolGMRe` and `ElmGMRe` instead. If the matrix-free GMRES solver were used, the calls would be to `SolMFG` and `ElmMFG`. In each case there would be additional changes, but the general outline of the code would remain the same.

Even though not shown in the aforementioned stack trace, several routines in PHASTA call the function `commu` *(common/commu.f)*, which is in charge of dealing with the communication between processes when running PHASTA in parallel (through MPI). As input parameters it takes a global and a local array (`ilwork`), along with a code string. If the latter is set to `'in'`, the lower rank processes communicate local results (for instance, the residual of each part) to their peers (Type I communication). If instead the code is `'out'`, the global result is backpropagated to each element, from higher to lower rank (Type II communication).

As an additional note, simulations in PHASTA require that one first generates the appropriate preprocessing files with the input data (mesh and boundary conditions, *geombc.dat*, and initial conditions, *restart.0.1*). These are generated by Chef from a given SimModeler case.

Finally, in terms of postprocessing, PHASTA writes the solution in binary *restart.%d.%d* files that can be read with Paraview through an XML file typically named `post.pht`, which describes the number of solution variables and timesteps available.

Figure 2: Stack trace of the main functions called when running PHASTA

Input

`main` File that starts the code: initializes MPI for parallel execution, calls the main `phasta` function, and ends the program.

`phasta` Sets up some parameters and calls a `run` function. There, the code initializes some common variables with a call to `initPhastaCommonVars`, which is a wrapper to the Fortran subroutine `common` *(common/common.f)*. After that, it takes all the input data (see below) and calls `proces`, starting the solver.

`input_fform` C++ function that reads the solver parameters to use for the simulation. The default value are listed in the *input.config* file, while the user-supplied changes to these defaults are specified in *solver.inp*. This subroutine reads both files and sets the appropiate variables, including the number of timesteps, the timestep size, the turbulence model (if any), the fluid viscosity, the tolerance for solution convergence, the integration rules order, etc.

`input` After reading the simulation parameters, this Fortran subroutine allocates the required array storage and handles the geometry input data. It does that by calling `readnblk` and `genint`.

`readnblk` Reads the binary files *restart.%d.%d* for initial conditions (or restart information from last simulation), *geombc.dat* for mesh information and boundary conditions, as well as *numstart.dat* to select the timestep with which to start the simulation. It calls `genblk`.

`genblk` Routine that generates the blocks of interior elements depending on the number of processors. Creates the `lcblk` data structure for each block.

`genint` Generates the spatial quadrature integration rules for the different types of elements (tetrahedrons, hexahedrons, wedges, pyramids). Creates structures with the quadrature points and weights for interior and boundary elements.

Preprocessing

`proces` Subroutine that generates the problem data and then calls the solution driver.

`gendat` First computes the domain size with `xyzbound`, and then handles the geometry and boundary conditions with the appropiate subroutines (see below).

`genshp` Generates the shape functions for triangular, quadrilateral, tetrahedron, wedge and brick elements, and pyramids in the interior blocks. That is, it creates the `shp` (shape functions) and `shgl` (local shape function gradients) data structures. These shape functions are not in the Fortran code, though, they are written in C and then called via Fortran wrappers. The source files can be found in `common/shptet.c`, `common/shphex.c`, etc., which at the same time make use of the files in `../shapeFunction/` directory.

`geniBC` Creates the binary encoding for essential boundary condition, that is, `iBC`. The code currently goes up to 2^{14}, as has been detailed in Table 1.

`genBC` Reads and generates the essential boundary conditions, that is, `BC`, which includes the constraint parameters depending on which variables are restricted (e.g., `BC` holds the corresponding combinations of u_r^g, u_s^g, u_t^g, C_i^r, C_i^s, C_i^t for prescribed u_i, u_i and u_j, and all three u_i components). It includes functions to compute the wall normal, `genwnm`, and the first node off the wall, `genotwn`.

`genshpb` Just like `genshp`, `genshpb` generates the shape functions and corresponding derivatives, but in this case it does so for the boundary elements.

`genini` Reads the initial conditions and the specified boundary conditions, and outputs the initial values for $\boldsymbol{Y}$ (primitive) variables (density, velocity, temperature). It calls `itrBC` to satisfy the boundary conditions.

Nonlinear equation marcher

`itrdrv` The semi-discrete predictor multi-corrector iterative driver that contains the generalized-α method to integrate the solution in time. Before calling `itrdrv`, there are other calls in `proces` that set up periodicity conditions, turbulence models, etc. The call to `itrdrv` is the main function of PHASTA.

After setting up several parameters that depend on the simulation configuration, with `itrSetup` being the main of these routines, the loop over time steps starts. Then, the predictor phase begins by calling `itrPredict` and `itrBC`, and then we move to the multi-corrector phase, which calls `SolGMRs`, and finishes by calling `itrCorrect` and again `itrBC`. As expected, before starting the next loop iteration, calls to `itrUpdate` and once more to `itrBC` take place.

`itrSetup` Sets up the integration parameters, i.e., α_f, α_m, and γ. By setting all the values to 1, one would be using a backward Euler integrator.

`itrPredict` Predicts the solution when starting a new timestep iteration by setting $\overset{n+\alpha_f}{\boldsymbol{Y}^{(i)}}$ and $\overset{n+\alpha_m}{\boldsymbol{Y}_{,t}^{(i)}}$ with the previous values.

`SolGMRs` The main solution driver. Calls `ElmGMRs` to form the LHS matrix and the RHS residual, applies periodic boundary conditions with `bc3per`, and solves the system of equations (detailed later). In the end, it outputs $\overset{n+\alpha_f}{\Delta\boldsymbol{Y}^{(i)}}$.

`itrCorrect` Corrects the solution vector $\overset{n+\alpha_f}{\boldsymbol{Y}^{(i)}}$ and the acceleration $\overset{n+\alpha_f}{\boldsymbol{Y}_{,t}^{(i)}}$.

`itrUpdate` Finally, this routine updates the actual solution at the timestep we are seeking $(n+1)$: $\overset{n+1}{\boldsymbol{Y}}$ and $\overset{n+1}{\boldsymbol{Y}_{,t}}$. It also updates `yold` and `acold` to prepare for the next iteration.

Linear equation formation

`ElmGMRs` Main function that computes the LHS mass matrix and the RHS residual vector. After some preprocessing, it starts with a loop over the interior element "blocks", and inside that calls `AsIGMR`, where the residual and tangent matrix are assembled. Then, LHS boundary conditions are satisfied with `bc3LHS` and the mass matrix is filled with `fillsparseC`. Next, over a loop over the boundary element "blocks", calls `AsBMFG` to assemble the boundary data. Finally, `bc3Res` modifies the residual to satisfy boundary conditions.

`AsIGMR` Computes the data corresponding to the interior elements. Starts by localizing the global variables making use of the `local` functions (`localy`, etc.), calling them with the `'gather'` parameter. Then, `e3` computes the element-level residual and mass matrix. After that, it calls `local` again, but now with the `'scatter'` parameter, in order to assemble the global residual, `res`, from the local one, `rl`. Finally, it assembles the block-diagonal preconditioner `BDiag` from the element-level `EGmass`.

`e3` Responsible for finding the 3D element-level Navier–Stokes RHS residual and LHS matrix. It does so through a series of subroutines that assemble each component individually, as traced in Figure 2, all of which are inside a loop over the integration points.

`getshp` Creates a matrix of shape functions and derivatives at the current quadrature point, i.e., `shp` and `shgl`.

`e3ivar` Evaluates local variables at the integration points, including the solution and its time derivative (from previous timestep). It then gets thermodynamic and fluid material properties, the element metrics, and from all of that the global gradient of the solution vector, i.e., `g1yi`, `g2yi`, `g3yi`.

`getthm` Calculates the thermodynamic properties. Depending on the input, it can compute multiple parameters, like the internal energy, specific heat, entropy, etc. under different assumptions.

`getDiff` Calculates fluid material properties. This includes μ, λ, etc.

`e3metric` Computes the metrics of the mapping from global to local, and vice versa. The results are the Jacobian, `dxdxi`, its inverse, `dxidx`, the determinant of the Jacobian weighted by the quadrature weight, `WdetJ`, and the global gradient of shape functions, `shg`.

`e3mtrx` Generates the $\boldsymbol{A}_0$ and $\boldsymbol{A}_i$ matrices at the quadrature point.

`e3conv` Computes the convective term to be added to both the RHS and the LHS, that is, $\boldsymbol{F}_i^{\text{adv}}$ stored in `ri(:,1:15)`. It also computes $\boldsymbol{A}_i\boldsymbol{Y}_{,i}$ as `rLyi` for stabilization (part of the $\mathcal{L}$ operator) and LHS, as well as $N_a\boldsymbol{A}_iN_{b,i}$, which is added to `EGmass`.

`e3visc` Computes the contribution of viscous and heat fluxes to both the RHS and LHS. Calculates the $\boldsymbol{K}_{ij}$ matrix and stores it in `stiff`. Computes the diffusive flux vector $\boldsymbol{F}_i^{\text{diff}}$ and adds it to `ri(:,1:15)`, resulting in the total flux $\boldsymbol{F}_i$.

`e3source` Adds the body forces term by substracting it from `ri(:,16:20)`.

`e3LS` First, it adds the least-squares operator, $\boldsymbol{A}_0\boldsymbol{Y}_{,t}$ to `rLyi`. Then, it calls `e3tau`, and handles the stabilization for both RHS and LHS, which involves modifying `ri` and `stiff`.

`e3tau` Computes the stabilization $\boldsymbol{\tau}$ matrix of timescales for the least-squares operator.

`e3DC` Handles the computation of the discontinuity capture operator in RHS.

`e3massr` Adds the $\boldsymbol{A}_0\boldsymbol{Y}_{,t}$ contribution to `ri(:,16:20)` (RHS).

`e3massl` Adds the $\boldsymbol{A}_0$ equivalent term to `EGmass` (LHS).

`e3wmlt` Takes care of multiplying each of the residual components by the shape functions or their derivative. N_b multiplies $\boldsymbol{A}_0\boldsymbol{Y}_{,t} - \boldsymbol{\mathcal{F}}$. $N_{b,i}$ multiplies $\boldsymbol{F}_i$. They are added together and form the elemental residual, `rl`. On the other hand, it adds the stifffness contribution to `EGmass`.

`fillsparseC` Fills up the sparsely stored LHS mass matrix. It generates `lhsK` from the element-level `EGmass`.

Boundary condition prescription

`itrBC` This routine has been referenced several times as part of the time integration, and it is responsible for satisfying essential-type boundary conditions on the $\boldsymbol{Y}$ variables. It directly modifies the `y` array with the information istored in `iBC` and `BC`. For each node, it checks whether a certain boundary condition is prescribed in `iBC` using Fortran's built-in bit testing functions (`btest` and `ibits`), and if that is true it will replace the existing value by the corresponding one set in the $\hat{\boldsymbol{q}}$ vector (see Equation (33)).

`bc3LHS` Satisfies the boundary conditions on the element-level LHS mass matrix. It modifies $\boldsymbol{M}_{BA}$ (`EGmass`) as $\boldsymbol{S}_v^T\boldsymbol{M}_{BA}\boldsymbol{S}_v$, where $\boldsymbol{S}_v$ is the linearized transformation of the weight space to account for each possible combination of boundary conditions.

`AsBMFG` Handles the computation and assembly of all the data corresponding to the boundary elements. Just like `AsiGMR`, it starts by localizing `y` and `x` with calls to the `local('gather')` subroutines. Then, passes this information to `e3b`, which computes the RHS residual, and finally assembles the modified residual with `local('scatter')`.

`e3b` It first finds the boundary nodes with `getbnodes` and then starts a loop over integration points. Inside that, it calls `getshpb`, `e3bvar`, and `getDiff`, and with all that ends up computing the natural fluxes `F1`, `F2`, `F3`, `F4`, `F5` for each variable. These take the prescribed values from `BCB` on the elements defined by `iBCB`, while the floating flux is computed with the current solution vector and the normal `bnorm` on the rest.

In this way, all the fluxes are added together and then multiplied by the shape function (`shpb`) and the determinant of the Jacobian and quadrature weight (`WdetJb`) to be numerically integrated and added to the local residual `rl`, thus satisfying such boundary conditions. Note that this routine is also responsible for computing the aerodynamic forces on the body by integrating the stresses, which are stored in `Force` and `HFlux`.

`getbnodes` Simple function to find the element nodes (or modes, in higher order elements) that lie on the boundaries of the element. It does so for the different types of elements that have been coded.

`getshpb` Equivalent to `getshp`, creates a matrix of shape functions and derivatives at the current quadrature point, i.e., `shpb` and `shglb` for the boundary elements.

`e3bvar` Like `e3ivar`, evaluates local variables at integration points. It also calls `getthm` to get thermodynamic properties. It computes the element metrics (`dxdxib` and `dxidxb`), and then the local (`gl1yi`, `gl2yi`, `gl3yi`) and global (`g1yi`, `g2yi`, `g3yi`) gradients. Finally, it computes the mass flux, pressure, heat flux, and viscous flux that are used when integrating the boundary natural fluxes.

`bc3Res` Adjusts the global residual $\boldsymbol{G}_B$ to satisfy essential boundary conditions, doing $\boldsymbol{S}_v^T \boldsymbol{G}_B$ for each different possible condition, zeroing out the residual of all terms where the solution is directly set.

`bc3Bdg` Modifies the block-diagonal preconditioner $\boldsymbol{B}$ to satisfy Dirichlet boundary conditions by zeroing out all terms different from the one prescribed, and setting that one to one.

`bc3per` Adjusts the global residual to account for periodic boundary conditions, adding the residual of one partner to the other, and zeroing out the first one.

Linear equation solution

`SolGMRs` After returning from `ElmGMRs`, the driver routine performs LU decomposition by calling `i3LU` and calls `Spsi3pre` as well. Then, it starts looping through GMRES cycles, and for each one loops over GMRES iterations. There, it computes a new Krylov vecctor, `uBrg`, by performing the $\boldsymbol{Ap}$ product with `SparseAp`, and orthonormalizes the result with respect to the previous vectors.

With these, the Hessenberg matrix `HBrg` is constructed, and then used to minimize the residual, `eBrg`, until it is below a specified tolerance. After the loop of iterations, solve for `yBrg`, and then update `Dy` adding to it the dot product of all Krylov vectors `uBrg` with their corresponding solution `yBrg`.

`i3LU` Handles LU factorization of the $\boldsymbol{B}$ preconditioner. The resulting upper and lower matrices are saved overwriting `BDiag`.

`Spsi3pre` Preconditions the LHS sparsely assembled matrix, `lhsK`, making use of `BDiag`.

`SparseAp` Performs the matrix-vector $\boldsymbol{Ap}$ product, where $\boldsymbol{A}$ (`lhsK`) is a sparsely stored matrix. To do that, it uses the helper arrays `row` and `col`.

Output

`checkpoint` The main purpose of this routine is to write the output files. It does so by calling `restar`.

`restar` Finally, this function writes the specified solution variables to the *`restart.%d.%d`* binary files. These can then be used in postprocessing to check the results of the simulation.

5 Implementation of slip boundary conditions

5.1 Formulation

From the definitions in section 2, we can see that both velocity slip and temperature jump are essential boundary conditions, since they are directly setting values of the solution vector $\boldsymbol{Y}$ at the wall (boundary). However, instead of prescribing a constant value, the velocity or temperature that they set is dependant on the velocity and temperature gradients at the wall, and so they are a function of the current solution.

Therefore, in order to carry on the implementation of slip-regime boundary conditions, we need to be able to compute these gradients at the wall. The existing code in `e3bvar` responsible for computing the fluxes in the formulation of natural boundary conditions already computes the solution gradients. However, these are evaluated at the integration or quadrature points, not at the element nodes in direct contact with the boundary or wall, as we seek.

As a starting point, we will focus on the 1D Maxwell velocity slip condition (1), simplified for the case of an isothermal wall and a wall parallel to the $x - z$ plane:

$$u_{\text{slip}} \equiv u_r = u_1 = A \left(\frac{2 - \sigma}{\sigma} \right) \lambda \left. \frac{\partial u_1}{\partial x_2} \right|_w \tag{38}$$

Recall from section 3 that the gradient of the solution can be evaluated as:

$$\boldsymbol{Y}_{a,i}(\boldsymbol{x}) = \sum_{a=1}^{n_{en}} N_{a,i}(\boldsymbol{x})\, \boldsymbol{Y}_a \tag{39}$$

and this is exactly what we will follow to compute the velocity derivative in Equation (38).

In the whole code, the arrays `shp`, `shgl`, `shg`, along with their boundary counterparts, only keep track of the shape functions at the quadrature points, and they are never evaluated at the element nodes. Hence, our first goal is to find the shape functions and their gradients, in parent space, for the boundary nodes on each element.

PHASTA arranges the boundary elements in such a way that the first nodes are always in the boundary face. In the case of hexahedrons, this means that the first 4 nodes are boundary nodes, or the first 3 for tetrahedrons. The current implementation only supports first order shape functions on meshes with just tetrahedron or hexahedron elements, but it could be easily extended to more elements. Making use of PHASTA's built-in shape function generation routines, we can find $N_a(\boldsymbol{\xi})$ and $N_{a,\boldsymbol{\xi}}(\boldsymbol{\xi})$ by passing the reference coordinates ($\boldsymbol{\xi}$) of the corresponding element boundary nodes. For hexahedrons, these are:

$$\boldsymbol{\xi}_1 = (-1, -1, -1) \qquad \boldsymbol{\xi}_2 = (+1, -1, -1) \qquad \boldsymbol{\xi}_3 = (+1, +1, -1) \qquad \boldsymbol{\xi}_4 = (-1, +1, -1) \tag{40}$$

Next, we need to transform the shape function gradients in the parent domain to the physical space, as detailed in Equation (11) and is reproduced below:

$$N_{a,i}(\boldsymbol{\xi}) = N_{a,\xi_j}\xi_{j,i} \tag{41}$$

where $\xi_{j,i}$ is the inverse of the deformation gradient $\boldsymbol{x}_{,\boldsymbol{\xi}}$.

Once we have the shape function gradients in real space, we need to localize the solution vector (`yl` from `y`). With the elemental or local variables, we can now compute $\frac{\partial u_1}{\partial x_2}$ following the template of Equation (39):

$$\left.\frac{\partial u_1}{\partial x_2}\right|_w^b = \sum_{a=1}^{n_{en}} N_{a,2}^b\, Y_2^a \tag{42}$$

where b denotes the boundary node of a given element and a each node of the element.

With the velocity gradient in our hands, we can finally compute the slip velocity according to (38). As explained in section 2, A and σ are constants (specified as input by the user), but the mean free path is a function of the current solution vector too:

$$\lambda = \frac{\mu}{\rho}\sqrt{\frac{\pi}{2RT}} \tag{43}$$

where $T = Y_5^a$, $\rho = \frac{Y_1^a}{RY_5^a}$, and μ and R are considered to be constants read from the input.

Finally, the nodal values of slip velocity found above are assembled into a global array. This way, the value of the $\boldsymbol{Y}$ vector can be directly modified with the computed slip on the global nodes where a boundary condition of type slip has been prescribed.

Besides the changes to the solution vector, as Dirichlet boundary conditions, we must also update the LHS tangent mass matrix and the RHS residual accordingly. In this case, we will reuse the same $\boldsymbol{S}_{rst}$ matrix derived when setting all 3 velocity components, since we are considering $\boldsymbol{u}_{\text{slip}} = \big(u_{\text{slip}}, 0, 0\big)$. Therefore, the same changes to $\boldsymbol{M}_{ba}$ and $\hat{\boldsymbol{G}}_b$ made for the three-component case will be applied.

Essential boundary conditions can also be imposed weakly. In this case, they are usually called Nitsche boundary conditions, and they can be implemented as an additional residual term:

$$\int_\Gamma N_b \left(\boldsymbol{Y}_{\text{current}} - \boldsymbol{Y}_{\text{target}}\right) d\Gamma \tag{44}$$

where $\boldsymbol{Y}_{\text{current}}$ is the current solution vector and, $\boldsymbol{Y}_{\text{target}}$, the modified value with —in this case— the computed slip velocity. This change is performed at the element level, inside a $\square_\Gamma$ integral, and added to the $\boldsymbol{G}_b^e$ residual as a new term. Since in this case we are performing an integration, we can reuse the solution gradient `g2yi` already available in PHASTA.

5.2 Changes in the code

Before describing the specific changes to existing subroutines (described in section 4) and what the newly implemented functions do, we must consider the preprocessing and changes to the input.

Since adding full support for a new type of boundary condition would require changes in SimModeler and Chef, the current implementation of slip boundary condition uses `scalar_1` (in SimModeler) to set —by prescribing any value— on which boundary faces (or walls) the new boundary condition will be imposed. In PHASTA, `scalar_1` corresponds to the bit 2^6 in the `iBC` array.

Additionally, there are four new parameters defined in *input.config*, which can then be changed by the user in *solver.inp*:

- `Slip BC Enabled`: takes the value `True` or `False`, and enables the use of the slip model on the boundaries where `scalar_1` is set. `True` by default. The value is kept in the `isSlipBC` integer (`1` when enabled, `0` otherwise).
- `Slip BC Type`: takes the value `Dirichlet` or `Nitsche`, and switches between the Dirichlet or the Nitsche implementation of slip boundary conditions. `Dirichlet` by default. The value is kept in the `slipNitsche` integer (`1` for Nitsche, `0` otherwise).
- `Slip Momentum Accommodation Coefficient`: sets the value of the σ parameter in Equation (38). 1.0 by default. The value is stored in the `slipSigma` variable.
- `Slip Proportionality Coefficient`: sets the value of the A parameter in Equation (38). 1.0 by default. The value is stored in the `slipConst` variable.

In order to implement these changes in the input, the new parameters had to be defined as global variables in both *common/common.h* (Fortran) and *common/common_c.h* (C). `input_fform` *(common/input_fform.cc)* was also modified to be able to read the new parameters.

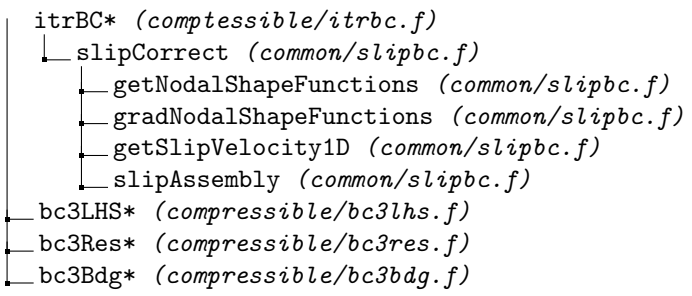
It is worth noting that a new module named `slipGeometry` is defined (in *common/slipbc.f*). It is used in `gendat`, before the first call to `itrBC`, to store the value of the global `x` node positions array in a separate variable, `xs`, and thus avoid having to add `x` as an argument to all the functions leading to `itrBC`, since $\boldsymbol{x}$ is necessary to compute the Jacobian and the velocity gradient.

The main file where all the new functions live is *common/slipbc.f*. Below is a short description of the main changes in the routines already described in section 4, now denoted with a symbol `*` at the end, and the new ones, for both Dirichlet and Nitsche types.

Dirichlet slip boundary conditions

Figure 3 summarizes the main calls to new functions and changes in previously described subroutines. After that, a brief description is given for each of them in line with the theory presented in subsection 5.1.

Figure 3: Stack trace of the main functions called when with Dirichlet slip boundary conditions



`itrBC*` Checks whether slip boundary conditions are enabled, whether they are of Dirichlet type, and whether there is any node where these are set. If all of these are true, then it calls `slipCorrect` to modify the `y` vector according to the provided `iBC` array.

`slipCorrect` Is the main function for essential-type slip boundary conditions. After initializing variables, it starts a loop over boundary element blocks, inside of which allocates arrays that are dependant on `npro` and `nshl`, for example. Then, it calls `getNodalShapeFunctions` to compute the shape functions and their gradients, and evaluate them at the nodes of the element topology in the current block. After that, it localizes `xs` (global `x`) and `y` to the element level, and gets the value of the viscosity μ from input data.

Next, it starts a loop over each boundary node. Just like the main PHASTA routines do, the shape functions found earlier `shpnodtmp(nenbl,nshl)` and their parent space gradients `shglnodtmp(nenbl,nsd,nshl)` are reshaped —inside the loop, for each boundary node— to be `shpnod(npro,nshl)` and `shglnod(npro,nshl,nsd)`. By calling `gradNodalShapeFunctions`, the gradients are converted to the physical domain, and with this result —saved in `shgnod(npro,nshl,nsd)`— the velocity gradient is found and then the slip velocity is computed by calling `getSlipVelocity1D`.

Once the loop over boundary nodes ends, `slipAssembly` is called to move the local slip values into a global array, which we call `uslip(nshg)`.

Finally, once the loop over boundary element blocks ends, using a Fortran `where` clause, combined with the bit-testing function `btest`, it checks on which nodes `scalar_1` is set, and on those it sets the velocity components to be $\big(u_{\text{slip}}, 0, 0\big)$.

`getNodalShapeFunctions` Returns the shape functions `shpnodtmp(nenbl,nshl)` and their parent space gradients `shglnodtmp(nenbl,nsd,nshl)` for the boundary nodes of an element topology. Supports first-order tetrahedron and hexahedron elements. It checks both `lcsyst` for the topology and `ipord` for the order, and errors out if the selected combination is not supported.

`gradNodalShapeFunctions` Given the shape function gradient in parent space and the local nodal coordinates, it computes the deformation gradient, its inverse, and with the latter the real space shape function gradient.

`getSlipVelocity1D` Basic routine that implements Equation (38). Takes the viscosity, density, temperature, and velocity gradient of a block of elements, and computes the corresponding slip for all of them.

`slipAssembly` Assembles the local array of slip velocities, `ul(npro,nenbl)`, into its global counterpart, `uslip(nshg)`.

`bc3LHS` Checks whether slip boundary conditions are enabled, whether they are of Dirichlet type, and whether there is any node where these are set. If all of these are true, then it modifies `EGmass` according to $\boldsymbol{S}_{rst}$.

`bc3Res` Checks whether slip boundary conditions are enabled, whether they are of Dirichlet type, and whether there is any node where these are set. If all of these are true, then it zeroes out the corresponding entries of the residual `res`.

`bc3Bdg` Checks whether slip boundary conditions are enabled, whether they are of Dirichlet type, and whether there is any node where these are set. If all of these are true, then it modifies `BDiag` like in the case where $\{r, s, t\}$ velocity components are set.

Nitsche slip boundary conditions

Just like in the previous case, Figure 4 first summarizes the new function calls. This is followed by a brief description of each subroutine.

Figure 4: Stack trace of the main functions called when with Nitsche slip boundary conditions

```
geniBC* (common/genibc.f)
  └─ setSlipiBC (common/slipbc.f)
e3b* (compressible/e3b.f)
  └─ getSlipVelocity1D (common/slipbc.f)
```

`geniBC` Checks whether slip boundary conditions are enabled, whether they are of Nitsche type, and whether there is any node where these are set. If all of these are true, then it calls `setSlipiBC` and passes references to both the `iBC` and `iBCB` arrays.

`setSlipiBC` Loops over all the elements in a block, and for each element tests whether one or more nodes have the `scalar_1` condition set in `iBC`. If so, the `iBCB` array is updated to reflect that `scalar_1 flux` is also set. In this way, when we are at the element level we can quickly check whether to add the Nitsche residual to an element or not.

`e3b` Checks whether slip boundary conditions are enabled, whether they are of Nitsche type, and whether there is any node where these are set. If all of these are true, then it calls `getSlipVelocity1D` (same as above) to compute the slip velocity for current block of elements, returning a local array `uslip(npro)`.

Then, it performs bit-testing and updates the flux `F2` —which corresponds to the first component of velocity, $\boldsymbol{Y}_2$— with the difference `(u1 - uslip)` for the elements where `scalar_1 flux` is set. Recall that, as described in section 4, these flux arrays are then multiplied by the undifferentiated shape functions and integrated over the element boundaries, performing the operation defined in Equation (44).

5.3 Results

The case setup for validation consists in flow over a flat plate. In the results presented below, a linear profile (from $u_1 = 0$ to $u_1 = 100\,\mathrm{m/s}$) is specified at the inlet face. The mesh, shown in Figure 5, is very coarse as is designed to resolve laminar boundary layers (low Reynolds cases).

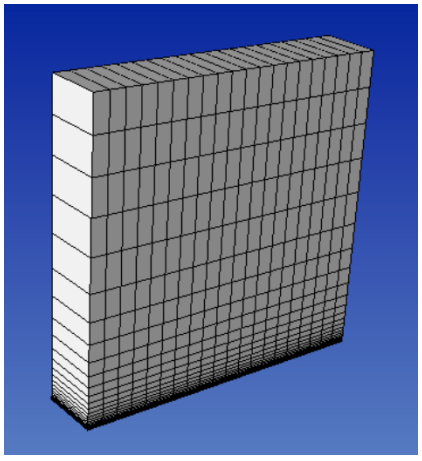


Figure 5: Coarse mesh setup

As mentioned, these first analysis are run in subsonic, but the rest of parameters (pressure, temperature) are already configured for a rarefied flow scenario, resulting in a very low density, for example. The proportionality constant and momentum accommodation coefficient are left with their default values, $A = 1.0$ and $\sigma = 1.0$ (diffusive reflection).

The same case is simulated for both slip boundary conditions (Dirichlet type) and no-slip. The simulation is left to run for a large number of timesteps, such that the solution barely changes between the last ones. The resulting velocity magnitude contour plot is basically the same for both cases, and is presented in Figure 6.

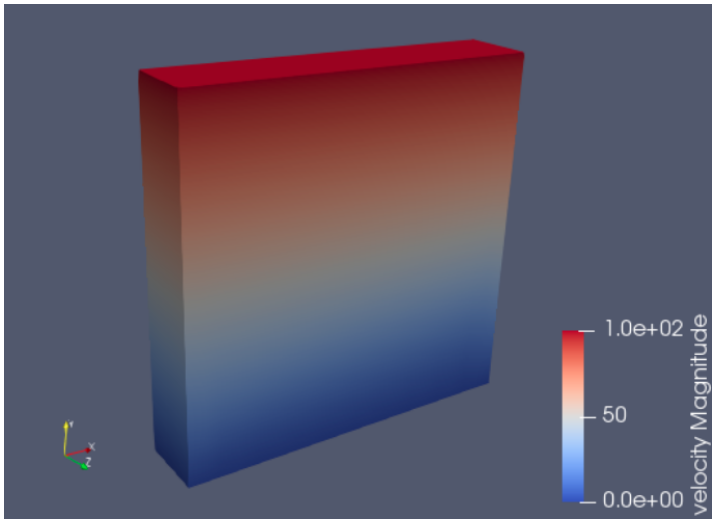
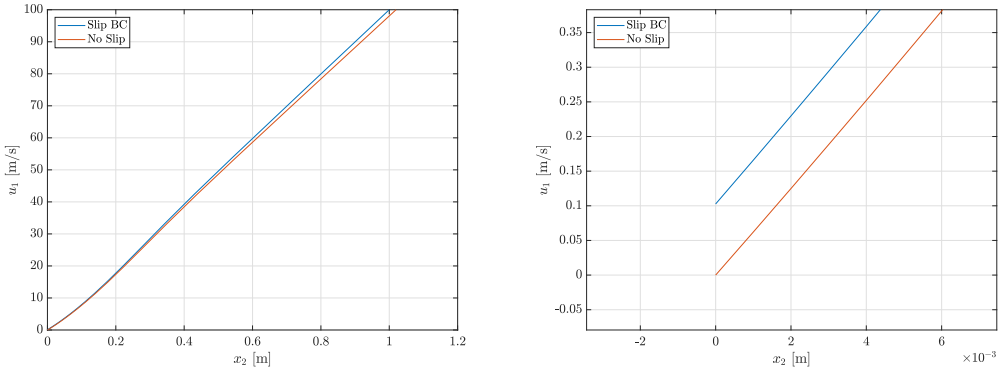


Figure 6: Contour plot of the velocity magnitude

To analyze the slip behavior near the wall, we have chosen to plot the velocity in the x direction over a vertical line in the middle of the domain. The results can be found in Figure 7. When zooming in (see 7b), we can clearly see that when using slip boundary conditions, u_1 at the wall is nonzero, as we were expecting. In this case, the slip value is very small, but that is just a consequence of the velocity magnitude being relatively small as well.



(a) Full velocity profile

(b) Zoom of the velocity profile near the wall

Figure 7: Plot of the u_1 velocity component over a flat plate with and without slip boundary conditions

6 Conclusions and future work

This project has shown the successful implementation of slip boundary conditions in PHASTA, as was intended. Additionally, both Dirichlet-type and Nitsche-type implementations of the 1D Maxwell model have been presented.

Having set the foundation with the subsonic case, the next step is to validate the results under hypersonic conditions corresponding to a spacecraft reentering Earth's atmosphere. To do that, the idea is to start simulations with a simple uniform flow over a flat plate, allowing for the shock to form at the edge (depending on the mesh, this might require the use of discontinuity capturing, available in PHASTA on the `e3DC` *`(compressible/e3dc.f)`* routine), and then studying the behavior throughout the rest of the plate.

This scenario is to be reproduced at multiple flight conditions equivalent to a spacecraft reentering Earth's atmosphere; that is, multiple Knudsen numbers (up to $\mathrm{Kn} \sim 1$), multiple Mach numbers, and multiple wall temperatures. At least some of these conditions will be selected in order to match other available data.

For the same range of conditions, simulations will also be run with the original PHASTA codebase in order to determine when and where the continuum breakdown happens. For that, the gradient-length local (GLL) Knudsen number [11] will be used.

In order to validate the obtained results, the velocity and temperature profiles at the wall will be compared with both Direct Simulation Monte Carlo and Navier–Stokes simulations [7, 4] and experiments [10, 8] from other sources. Finally, skin friction coefficient and different integrated quantities —such as total drag— will be compared against DSMC simulations to be performed using the SPARTA software [14].

Further work would consist in the implementation of the generalized Maxwell condition, which was presented in Equation (2). Doing this would allow for accurate simulations of bodies other than the basic 2D flat plate (in which case the generalized slip model reduces to the 1D version), including a simple sphere or a blunt cone under uniform flow, as both cases have been tested before and there is available data for validation [5, 11, 12, 3]. These simulations would be interesting to see how the generalized Maxwell condition actually accounts for surface curvature.

References

[1] G. A. Bird and J. Brady. *Molecular gas dynamics and the direct simulation of gas flows*. Vol. 5. Clarendon press Oxford, 1994.

[2] M. Gad-el-Hak. "The fluid mechanics of microdevices—the Freeman scholar lecture". In: *Journal of Fluids Engineering* 121.1 (1999), pp. 5–33.

[3] T. Gökçen and R. MacCormack. "Nonequilibrium effects for hypersonic transitional flows using continuum approach". In: *27th Aerospace Sciences Meeting*. 1989, p. 461.

[4] T. Gökçen, R. MacCormack, and D. Chapman. "Computational fluid dynamics near the continuum limit". In: *8th Computational Fluid Dynamics Conference*. 1987, p. 1115.

[5] G. Huang and R. K. Agarwal. "Computation of Rarefaction Effects on a Blunt Body in Hypersonic Flow". In: *55th AIAA Aerospace Sciences Meeting*. 2017, p. 1148.

[6] N. T. Le, N. A. Vu, and L. T. Loc. "New type of smoluchowski temperature jump condition considering the viscous heat generation". In: *AIAA Journal* 55.2 (2016), pp. 474–483.

[7] N. Le, C. J. Greenshields, and J. Reese. "Evaluation of nonequilibrium boundary conditions for hypersonic rarefied gas flows". In: *Progress in Flight Physics* 3 (2012), pp. 217–230.

[8] D. A. Lockerby, J. M. Reese, and M. A. Gallis. "Capturing the Knudsen layer in continuum-fluid models of nonequilibrium gas flows". In: *AIAA journal* 43.6 (2005), pp. 1391–1393.

[9] D. A. Lockerby et al. "Velocity boundary condition at solid walls in rarefied gas calculations". In: *Physical Review E* 70.1 (2004), p. 017303.

[10] A. Lofthouse and I. Boyd. "Hypersonic flow over a flat plate: CFD comparison with experiment". In: *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*. 2009, p. 1315.

[11] A. J. Lofthouse, L. C. Scalabrin, and I. D. Boyd. "Velocity slip and temperature jump in hypersonic aerothermodynamics". In: *Journal of thermophysics and heat transfer* 22.1 (2008), pp. 38–49.

[12] K. Luo et al. "Numerical Investigation of Hypersonic Slip Flow". In: *2015 International Conference on Automation, Mechanical Control and Computational Engineering*. Atlantis Press. 2015.

[13] J. C. Maxwell. "On stresses in rarified gases arising from inequalities of temperature". In: *Philosophical Transactions of the royal society of London* 170 (1879), pp. 231–256.

[14] *SPARTA Direct Simulation Monte Carlo (DSMC) Simulator*. Sandia National Laboratories. URL: `https://sparta.sandia.gov`.