Ryan Skinner

Improving Compressible BC Robustness in PHASTA

1 Proposal

For the final project, I propose a detailed investigation and improvement of boundary condition implementation in the compressible version of PHASTA. Currently, the incompressible version's non-linear convergence is much more robust than that of the compressible version; it often takes considerable manual effort to push a compressible case through its initial transient. By studying the incompressible version's implementation of BCs, I will identify aspects of the compressible code for improvement. The final deliverable will be a detailed algorithmic outline of both codes' BC implementation, a discussion of improvements selected for study, and benchmarks of the degree to which they affect non-linear convergence of a familiar compressible case: that of the aggressive subsonic diffuser.

2 Initial Outline of PHASTA: Incompressible

Though the final deliverable will include outlines of the BC implementation in both compressible and incompressible versions of PHASTA, the initial outline will be presented here using the latter version due to its robustness. In the following outline, routines followed by an asterisk (*) are expanded and described in their own separate block. Important integers and arrays are listed in Table 2. Furthermore, we ignore cardiovascular impedance and RCR boundary condition statements.

PHASTA begins execution at `main`, located in `phSolver/[in]compressible`, depending on which branch is desired. This function initializes MPI, and then calls `phasta`, located in `/phSolver/common`. Here, inputs are read and computed in `input`, and then the solver is run by calling `proces`.

- `main`
  - initialize MPI
  - `phasta`
    - initialize PETSc
    - `input_fform` — read ASCII data from `input.config` and `solver.inp`
    - `input`* — populate data structures with problem set-up and solver parameters
    - `proces`* — generate problem data and calls the solution driver
    - finalize PETSc
  - finalize MPI

Most file input occurs in `input`, which populates data structures with problem and solver parameters.

- `input`
  - `readnblk` — read and block data
    - read `numstart.dat` and finds appropriate `restart.dat` files
    - read geometry from Posix or SyncIO files using `phio_readheader`

- calculate maximum number of boundary element nodes
- initialize constants like `ndof`, `ndofBC`, `ndiBCB`, and `ndBCB`
- `ctypes` — initialize variables for parallel processing
- `genblk` — read and block connectivity for interior elements; there are two other versions of this function, `genblkPosix` and `genblkSyncIO`, which just address idiosyncrasies of the file formats; all call `gensav`
  - `gensav` — save element block data to and return an `ien` array, mapping element and local node numbers to global node number, and a material type flag `mater`
- read BC mapping array into `nBC`
- read temporary boundary condition code into `iBCtmp`
- read BC data into `BCinp`
- read periodic BC data into `iperread`
- `genbkb` — generate boundary element blocks and traces for gather/scatter operations; there are two other versions of this function, `genbkbPosix` and `genbkbSyncIO`, which just address idiosyncrasies of the file formats; all call `gensvb`
  - `gensvb` — save boundary element block data to and return boundary nodal connectivity `ienb`, boundary condition codes `iBCB`, boundary condition values `BCB`, and material type flag `materb`
- read restart data for solution `qold`, displacement `uold`, and accelerations `acold`

- assert valid input constants (e.g. `icoord`, `navier`, `iexec`) defined in `common.h`
- `genint` — generate integration information: number of quadrature points on the interior `nint` and boundary `nintb`, their weights `Qwt` and `Qwtb`, and locations `Qpt` and `Qptb`; the following routines are called to populate these variables depending on type of element and whether it is on the boundary
  - `symtet` — interior tetrahedra
  - `symtri` — boundary tetrahedra, boundary wedges (if boundary face is a triangle)
  - `symhex` — interior hexahedra
  - `symquad` — boundary hexahedra, boundary pyramids (if boundary face is a quadrilateral)
  - `sympyr`— interior pyramids
  - `symtripyr` — boundary pyramids (if boundary face is a triangle)
  - `symwdg` — interior wedges
  - `symquadw` — boundary wedges (if boundary face is a quadrilateral)
- estimate number of global nonzeros `nnz` based on basis function order `ipord`
- compute fluid thermodynamic properties, such as specific heats and the gas constant

The next overarching routine, `proces`, generates problem data and calls the solution driver.

■ `proces`

- `gendat` — generate geometry and BC data
  - `xyzbound` — compute length scales (domain size) of the problem by looking at minima and maxima of the `x` array
  - `genshp` — generate interior element shape functions and shape function derivatives
    - loop through element topologies, getting their coordinate system and element type, and then generate the parent element shape functions and their derivatives by calling either `shpTet` (tets), `shphex` (hexes), `shp6w` (wedges), or `shppyr` (pyramids)

    - these are all Fortran wrappers (located in `phasta/common/*.c`) for the C routines called `TetShapeAndDrv` (in `shapeFunction/src/uniformP.c`), `HexShapeAndDrv`, `WedgeShapeAndDrv`, and `PyrShapeAndDrv` (these three in `phasta/common/newshape.cc`) respectively
    - return `shp(a,i,j,p)` and `shgl(a,i,p)`, which are indexed by topology index `a`, spatial dimension(s) `i` and `j`, and the integration point index `p` $= 1, \ldots,$ `nint`
  - `geniBC` — generate boundary condition codes, stored as a bitmap for each global node number (bitmap described just before description of `iBC` routine in this document)
    - set `iBC(:)= iBCtmp(nBC(:))` if this partition has boundary nodes; `nBC` is used to map from the `iBCtmp` data read in `readnblk` to `iBC`, which is indexed by the full global node number
  - `genBC` — generate the essential boundary conditions
    - set `BCtmp = BCinp` if this partition has boundary nodes; `BCtmp(nshg,6+5*I3nsd)` has a second index of ρ, T, p, velocities, scalars (?)
    - `genwnm` — calculate wall normals and modify `BCtmp` with the appropriate constraints
    - `genotwn` — determine first "off-the-wall-node" for each node, store result in `otwn(nshg)`
    - `genBC1` — account for arbitrarily-oriented velocity constraints u_r, u_s, and u_t, finally storing the simplified boundary condition constraint result in `BC`; note that second index of `BC` holds ρ, T, p, u_1, u_2, u_3, and scalars
  - `genshpb` — generate boundary element shape functions and derivatives, storing results in `shpb` and `shglb`; this routine is analogous to `genshp`
  - LES: call `setfilt`, `filtprep`, and depending on `iLES`' value, `setave` and `aveprep`
  - `genini` — generate initial values of solution variables
    - `restar` — sort initial values into `y` (called `q` inside `restar`) and `ac` from `qold` and `acold`, respectively, that were read in `readnblk`
    - `itrBC`* and `itrBCSclr`* — satisfy BCs, outlined later in this document
- `setper` and `perprep` — allocate and store inverse of sum of one and number of slaves in `rcount`, in preparation for dealing with periodic boundaries
- LES: `keeplhsG` and `setrls`
- `initStats` — allocate arrays to store flow statistics
- RANS: `initTurb`
- `itrdrv`* — iterate the discrete solution using the predictor multi-corrector algorithm

Numerical solution of the time-integrated unsteady Navier-Stokes equations occurs within `itrdrv`.

■ `itrdrv`

- `initTimeSeries` — initialize time series collection to `varts.*.dat` files using `xyzts.dat` input
- initialize `istep` and `ifuncs(:)` to zero
- set `yold = y` and `acold = ac`, that is, populate $\underline{\overset{n}{Y}}$ and $\underline{\overset{n}{Y}}_{,t}$ with their converged solutions from the previous time step, which came from a restart file
- `initEQS` — create the `rowp` and `colm` maps to facilitate sparse storage of the tangent matrix
  - determine how many scalar equations need solution, `nsclrsol` (# scalars + 1 if temperature)
  - determine whether we are solving the flow
  - `genadj` — pre-process the adjacency list
    - `do iblk = 1, nelblk` — loop over element blocks (groups of elements with the same topology)

        - ▸ `Asadj` — generate adjacency data structures `row_fill_list` and `adjcnt`
          - ▹ declare `row_fill_list` to have dimension `(nshg, 15*nnz)`, where `15*nnz` is a high estimate of the maximum number of adjacent nodes
          - ▹ `row_fill_list(A,:)` holds global nodes that share local support with global node `A`
          - ▹ `adjcnt(A)` holds the number of nodes adjacent to `A`, that is, how many entries `row_fill_list(A,:)` was populated with on purpose
          - ▹ *note*: some operations here are $\mathcal{O}(n^2)$, but n is relatively small on each processor, and this process only needs to be done once for a given mesh connectivity
      - ○ build the `colm` array (which is trivial to do at this point)
      - ○ sort `rowp`, because we binary search it when computing the sparse $\underline{\underline{A}}\,\underline{p}$-product, and also compute the number of non-zero element blocks `icnt` on this partition
    - set `nnz_tot = icnt`
    - depending on `nsolflow` and `nsclrsol` (whether this is a flow or scalar solve), initialize certain constants, such as `equType`, `nDofs`, `nPermDims`, `nTmpDims`, and allocate certain arrays, such as `apermS` and `atempS`
  - ▫ `genlmass` — generates lumped mass matrix `gmass` if we are using a lumped mass fraction on either the LHS or RHS
    - `AsImass` assembles the interior lumped mass matrix within a loop over element blocks
      - ○ `localx` — gather node coordinates to local frame
      - ○ `do intp = 1, ngauss` — loop over Gauss points
        - ▸ `getshp` — returns the shape functions evaluated at this point, `shape` and `shdrv`
        - ▸ `e3metric` — compute the deformation gradient ($dx_i/d\xi_j$ or `dxdxi(npro,i,j)`) and its inverse (`dxidx`), as well as the quadrature-weighted Jacobian determinant `WdetJ` and the global shape function gradient `shg`
        - ▸ add contribution of this Gauss point to the local mass matrix
      - ○ compute the trace and scale the diagonal, operating on local mass matrices
      - ○ `local` — assemble the global residual, `gmass`
  - ▫ initialize `lstep0 = lstep + 1` to hold the first time step solved by the current run
  - ▫ `do itsq = 1, ntseq` — loop over time sequences; as far as I can tell `ntseq = 1` is the default in `input.config`, and time sequences are not often used
    - set `itseq = itsq`
    - set iteration-specific variables `nstp = nstep`, `nitr = niter`, `LCtime = loctim`, and `dtol(:)= deltol(:)`, where all of the longer-named variables are indexed by `itseq`
    - `itrSetup` — set up time integration parameters
      - ○ calculate α_m, α_f, and γ as functions of ρ_∞ (`almi`, `alfi`, and `gami` as functions of `rhoinf`)
      - ○ set inverse of global time step `Dtgl` and CFL data `CFLfl`
    - calculate number of flow solves per time step, store in `nitr` (IC), `niter` (C)
    - initialize `istop = 0`; flag can be set to stop the solver based on statistics of the residual
    - `do istp = 1, nstp` — main loop over time steps
      - ○ LES: `lesmodels`
      - ○ `asbwmod` — set traction BCs if turbulence wall model is set (`itwmod`)
      - ○ `itrPredict`* — predict primitive variables at time $n+1$
      - ○ `itrBC`* — satisfy BCs on primitive variables; return a modified `y`
      - ○ `itrBCSclr` — satisfy BCs on scalar `isclr`; return a modified `y`

        - `do istepc = 1, seqsize` — loop over individual solves of flow and scalar
          - `icode = stepseq(istepc)` — get sequence code
          - `if` this is a flow solve
            - `SolFlow`* — perform a flow solve
          - `else if` this is a scalar solve
            - `SolSclr` — perform a scalar solve
          - `else` this is an update
            - `itrCorrect`* and `itrBC`* — update flow if desired
            - `itrCorrectSclr` and `itrBCSclr` — update scalar if desired
        - `stsGetStats` — obtain time averaged statistics
        - find solution at end of time step and move it to old solution variables
        - increment `istep` and `lstep`
        - `Bflux` — compute the consistent boundary flux if desired
      - deallocate variables and close files
  - deallocate variables and close files

The following routines specific to the generalized-alpha method are used primarily in `itdrv`, but some are called (above) to prepare input data, such as `itrBC` and `itrPredict`.

- `itrPredict` — predict solution variables at time $n+1$
  - `if (ipred .eq. 1)` — we are using same-velocity prediction, as discussed in class
    - set $\underline{Y}^{n+1(i)} = \underline{Y}^{n}$ with `y = yold`
    - set $\underline{Y}^{n+1(i)}_{,t} = (1-1/\gamma)\underline{Y}^{n}_{,t}$ with `ac = acold * (gami-one)/gami`
  - other prediction methods (zero-acceleration, same-acceleration, and same-delta) are also supported with different values of `ipred`

Boundary conditions are set with the `iBC` and `BC` arrays. The bits of `iBC`, in increasing order, indicate whether the following BCs are set: ρ, T, p, u_1, u_2, u_3, scalars 1–4, periodicity, scaled plane extraction (SPEBC), axisymmetry, and deformable wall (for cardiovascular cases). This means for each global node, `iBC` has at least 14 bits. Note that `ibits(i,a,l)` extracts bits `a+1` through `a+l` of the integer `i`, and returns the base-10 integer. This routine is used to help identify and process boundary condition flags held in `iBC`. For example, if `ibits(iBC,3,3).eq. 1` then u_1 is the only velocity component specified essential BC.

- `itrBC` — satisfy BCs on the primitive variables
  - impose limits on flow variables in `y`, using the `ylimit` data structure of dimension `(3, nflow)`, whose first index contains the limit flag, lower limit, and upper limit for each flow variable
  - velocity
  - pressure
  - local periodic
  - global periodic

In the event of an update, `itrCorrect` is called (followed by `itrBC`) to update the solution at time $n+1$, such that it is consistent with the most recent solve.

- `itrCorrect`

- set `fct1` $= \gamma \Delta t$
- set `fct2` $= \gamma \alpha_f \Delta t$
- update velocity: $\underline{Y}^{n+1(i)}_{\text{velocity}}$ `+= fct1 * solinc(:,1:3)`, where `solinc(:,1:3)` $= \underline{Y}^{n+1(i)}_{\text{velocity},t}$
- update pressure: $\underline{Y}^{n+1(i)}_{\text{pressure}}$ `+= fct2 * solinc(:,4)`, where `solinc(:,4)` $= \underline{Y}^{n+1(i)}_{\text{pressure},t}$
- update acceleration: $\underline{Y}^{n+1(i)}_{\text{velocity},t}$ `+= solinc(:,1:3)`

Once boundary conditions have been satisfied at the beginning of an iteration, `SolFlow` is called from within `itrdrv` to perform a flow solve. The general form of the system we seek a solution to is

$$\begin{bmatrix} \underline{\underline{K}} & \underline{\underline{G}} \\ -\underline{\underline{G}}^T & \underline{\underline{C}} \end{bmatrix} \begin{bmatrix} \Delta \underline{u}_{j,t} \\ \Delta \underline{p}_{,t} \end{bmatrix} = \begin{bmatrix} \underline{R}^{\text{mom}} \\ \underline{R}^{\text{cont}} \end{bmatrix} \tag{1}$$

Because we are looking at the incompressible code, the temperature equation is not included in the flow solve matrix system, and is instead solved separately as a scalar. Further note that in code, $\underline{\underline{K}}$ = `xKebe`, and $\underline{\underline{G}}$ on top of $\underline{\underline{C}}$ is `xGoC`.

- `SolFlow` — perform a flow solve; output `res` preconditioned residual,
  - `itrYAlpha` — compute $\underline{Y}^{n+\alpha_f}{}^{(i)}$ and $\underline{Y}^{n+\alpha_m(i)}_{,t}$, store respectively in `yAlpha` and `acAlpha`
  - `ElmGMR` — compute tangent matrix, residual vector, and preconditioning matrix for GMRES
    - `if` using a global reconstruction approach
      - `do iblk = 1, nelblk` — loop over element blocks
        - `if` this is the last time step and we need to compute vorticity, call `AsIqGradV`
        - `AsIq`* — compute and assemble the diffusive flux residual vector `qres` and the lumped mass matrix `rmass`
      - `qpbc` — satisfy periodic BCs on `rmass` and `qres`
    - initialize `res = 0`, which is the full system residual $\underline{G}_b$
    - `do iblk = 1, nelblk` — loop over element blocks
      - `AsIGMR`* — compute and assemble residual and tangent matrix
      - `bc3lhs` — satisfy boundary conditions on the tangent matrix
        - `do` loop over elements `iel` and local shape functions `inod`
          ▷
      - `fillsparseI` — fill the sparse tangent matrix data structures,
    - `lmassadd` — add lumped mass matrix contributions if we are lumping
    - compute time averaged statistics
    - `do iblk = 1, nelblb` — loop over boundary element blocks
      - `AsBMFG` —
      - `bc3lhs` —
      - `fillsparseI` —
    - `rotabc` — rotate the residual vector before cross-processor communication for efficiency
    - `commu` — communicate with other processors
    - `bc3Res` — satisfy boundary conditions on the residual vector

- `usrNew` — set up GMRES solver
- `myfLesSolve` —

The diffusive flux residual vector `qres` and the lumped mass matrix `rmass` are compred and assembed within `AsIq`:

- `AsIq`
  - gather `y` → `yl` and `x` → `xl` via `localy` and `localx`
  - initialize `ql = 0` and `rmassl = 0`
  - `e3q` — compute the element-wise residuals `ql` and `rmassl`
    - `do intp = 1, ngauss` — loop over Gauss points
      - `getshp` — get shape functions and derivatives at Gauss points
      - `e3qvar` — compute integration variables necessary for formation of `ql`; this routine is straight-forward, and returns `dxdxi`, `dxidx`, `WdetJ`, `shg`, and gradients of `y` in the usual manner
      - `getdiff` — compute the viscosity at this point
      - compute diffusive fluxes, stored in `qdi`
      - add local node contributions to `ql`
      - compute local contribution to the lumped mass matrix `rmassl`
      - `if isurf .eq. 1` — if surface tension is being computed, compute and fill the extra three indices of `ql`
    - normalize the mass matrix if desired (if `idiff == 3`)
  - scatter `ql` → `qres` and `rmassl` → `rmass`

Going through lectures 26 and 27.

First in the compressible code. Data structures are used in `solgmr` -> `elmgmrs` (sparse). Section on diffusive flux reconstruction, set up some arrays for interior elements. call `asigmr`, took care of volume integrals. now come to boundary elements, which is where integral over gamma takes place. block boundary elements as a separate list of elements with separate connectivity. as `asbmfg` is called, `mienb` holds boundary elements. computing normal gradients requires nodes off of the boundary. solution goes into `asbmfg` (no time derivatives are input, unlike `asigmr`), out comes a modified solution. in `asibmfg`: working with a block of elements; solution and coordinates are localized, local residual is zeroed, call `e3b`, assemble local residual. in `e3b`: loop over quadrature points (`ngaussb`), `getshpb` to get boundary shape functions, `e3bvar` called with surface normals, need `Fv{2,3,4}` to evaluate the floating flux, let `e3bvar` compute `Fv` values and fluxes, then test if we should use computed value or value from prescribed boundary condition. in `e3bvar`: interpolate nodal values to quadrature points, call `getthm` to compute thermodynamic state, compute element metrics for mapping physical space to get `wdetj`, compute `rou`,`p` and `tau*n`,`heat` (normal flux, pressure, traction vector, heat flux) that is, `rou` takes $h^m(\xi_l)$, `p` takes $h^p(\xi_l)$, `tau*n` takes $h^v_*(\xi_l)$, etc. what's passed out is these things and the raw variables, their gradients, and the derived thermodynamic state. Back to `e3b`, do convective pressure, n-s, heat terms. after return from `e3bvar`, if no natural bc flag is set, we overwrite `rou`,`p` with floating values. compute euler stuff; then compute viscous stuff. get floating flux `tau*n`, overwrite where bits are not set. be careful what's passed out and in. also compute aerodynamic forces and heat flux in `e3b`, since we're doing surface integrals anyway.

incompressible is different, slightly. don't interpolate temperature at the outset; compute normal via cross-product; compute deformation gradient, local and global variable gradients; `unm` has the floating value of $\underline{u} \cdot \underline{n}$. eventually compute `tau*n`, which has the total stress floating value. skip over a bunch of

deforming-wall stuff. iBCB did not come in to e3bvar; only floating flux stuff is computed in `e3bvar` for incompressible. all nodal interpolation is now done in `e3b`:

for Dirichlet bcs, `iBC` was a bitmap to boundary conditions `BC`

for natural bcs, `ibcb`(1:$\text{nel}_{\text{in block (npro)}}$,...) is a bitmap to boundary conditions `BCB`(:,...), where ... takes normal flux, pressure, traction vector, and heat flux take values 1–4.

going through more code... (2016-03-30)

this stuff is used in `irdrv` when it calls `solgmrs`, calls `elmgmrs` to populate lshk with current iteration's tangent values (same for `res`). allows us to start `gmres`; factorize matrix; precondition rhs with `i3lu`; `spsi3pre` sparse matrix preconditioning of `lhsk`; copy preconditioned residual into `uBrg`, which is a collection or Krylov vectors; calculate it's norm, make orthonormal; outer gmres loop `do 2000` can be skipped, which is the gmres restart; actual start of GMRES discussed in class is `uBrg` statement just before `do 1000`; `sumgat` does off-processor (communication).

Symbol	Dimension	Description
`nshg`		# global shape functions (`nshg` = `nnp` if piecewise linear)
`nnp`		# global nodal points
`npro`		# elements in a block of same-topology elements, indexed by e
`nshl`		# nodes per element, indexed by a
`ndof`		# degrees of freedom, including scalars for turbulence models
`nflow`		# flow variables (4 incompressible, 5 compressible)
`ntseq`		# time sequences, which seems seldom used and defaults to 1
`nstep`		# time steps requested per sequence
`nelblk`		# element blocks
`ipord`		order of basis functions
`lstep`		current time step
`lstep0`		first time step solved by current run, initialized to `lstep+1`
`istep`		step number relative to start of run
`iter`		iteration number
`niter`	`(MAXTS)`	# multi-corrector iterations per time step
`loctim`	`(MAXTS)`	local time stepping flag (?)
`deltol`	`(MAXTS, 2)`	velocity and pressure delta ratios
`impl`	`(MAXTS)`	heat, flow, and scalar solver flags (1's, 10's and 100's places)
`iturb`		indicates which turbulence model to use
`ifunc`		function evaluation counter, `niter*(lstep-lstep0)+iter`
`ifuncs`	`(6)`	function evaluation counter (?)
`y`	`(nshg, ndof)`	$\underline{Y}_A^{n+\alpha_f\,(i)}$ (meaning changes throughout)
`ac`	`(nshg, ndof)`	$\underline{Y}_{A,t}^{n+\alpha_m\,(i)}$ (meaning changes throughout)
`yold`	`(nshg, ndof)`	$\underline{Y}_A^{n\,(i)}$ (meaning changes throughout)
`acold`	`(nshg, ndof)`	$\underline{Y}_{A,t}^{n\,(i)}$ (meaning changes throughout)
`x`	`(nshg, nsd)`	node coordinates
`iBC`	`(nshg)`	BC codes
`BC`	`(nshg, ndofBC)`	BC constraint parameters
`iper`	`(nshg)`	periodicity table
`mien`	`(nelblk)`	pointer to IEN array (interior): has dimension `(nshg, 15*nnz)`
`mienb`	`(nelblk)`	pointer to IEN array (boundary): has dimension `(nshg, 15*nnz)`
`shp`	`(nshape, ngauss)`	physical shape functions at Gauss points (interior)
`shb`	`(nshapeb, ngaussb)`	physical shape functions at Gauss points (boundary)
`shgl`	`(nsd, nshape, ngauss)`	parent shape function gradients at Gauss points (interior)
`shglb`	`(nsd, nshapeb, nguassb)`	parent shape function gradients at Gauss points (boundary)

3 General Notes

- In PHASTA, a block contains elements of the same topology.

4 Life's Persistent PHASTA Questions

- Is `qold`, allocated in `readnblk.f` ever deallocated? Can't find it.
- Why do most of the time step parameters have dimension `MAXTS`?
  - It also seems that some parameters are indexed by `itseq`, but don't change from step to step.
- When is it the case that `ndof` $\neq$ `nflow`? For example during its limit-imposing stage, `itrbc` loops over `nflow` when indexing `y`'s dimension of size `ndof`.
- Often the value of `datmat(1,2,1)` is assigned to a variable like `rmu`; where it it calculated? Kinematic viscosity? See `getdiff`.
- In `genBC1`, `BC(:,1)` gets assigned both density and pressure, so what goes in `BC(:,6)`???