Ryan Skinner
Due 2016/??/??

Improving Boundary Condition Stability in PHASTA

1 Introduction

2 Initial Outline of PHASTA

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`, a Fortran routine. Within `proces`, `gendat` generates geometry and BC data.

Routines followed by an asterisk (*) are outlined in further detail separately.

INCOMPRESSIBLE ONLY, and we ignore cardiovascular impedance and RCR boundary stuff.

- `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

Inputs:

- `input` — populate data structures with problem set-up and solver parameters
  - `readnblk` — read and blocks 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`
    - `genblk` — read and block connectivity
    - 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
    - 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`
- estimate number of global nonzeros `nnz` based on basis function order `ipord`
- compute fluid thermodynamic properties

Process:

- `proces` — generate problem data and call the solution driver
  - `gendat` — generate geometry and BC data
    - `genshp` — generate interior element shape functions and 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 C routines, and some are in `phasta/shapeFunction/`, whereas others are in `phasta/phSolver`
      - 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
      - set `iBC = iBCtmp` if this partition has boundary nodes; see code for `BCinp` description
    - `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 (like `genshp`), storing results in `shpb` and `shglb`
    - 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
  - `setper` and `perprep` — store inverse of sum of one and number of slaves in `rcount`
  - 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`. Working arrays are listed in Table 2.

- `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 $\overset{n}{\underline{Y}}$ and $\overset{n}{\underline{Y}}_{,t}$ 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 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 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`
- 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

Iteration routines...

- `itrPredict` — predict solution variables at time $n+1$
  - `if (ipred .eq. 1)` — we are using same-velocity prediction, as discussed in class
    - set $\overset{n+1}{\underline{Y}}{}^{(i)} = \overset{n}{\underline{Y}}$ with `y = yold`
    - set $\overset{n+1}{\underline{Y}}{}^{(i)}_{,t} = (1 - 1/\gamma)\overset{n}{\underline{Y}}_{,t}$ 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

Once boundary conditions have been satisfied, `SolFlow` is called to perform a flow solve:

- `SolFlow` — perform a flow solve; output `res` preconditioned residual,
  - `itrYAlpha` — compute $\overset{n+\alpha_f}{\underline{Y}}{}^{(i)}$ and $\overset{n+\alpha_m}{\underline{Y}}{}^{(i)}_{,t}$, store respectively in `yAlpha` and `acAlpha`
  - `ElmGMR` — compute tangent matrix, residual vector, and preconditioning matrix for GMRES

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: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`		# 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)`	$\overset{n+\alpha_f}{\underline{Y}}{}_A^{(i)}$ (meaning changes throughout)
`ac`	`(nshg, ndof)`	$\overset{n+\alpha_m}{\underline{Y}}{}_{A,t}^{(i)}$ (meaning changes throughout)
`yold`	`(nshg, ndof)`	$\overset{n}{\underline{Y}}{}_A^{(i)}$ (meaning changes throughout)
`acold`	`(nshg, ndof)`	$\overset{n}{\underline{Y}}{}_{A,t}^{(i)}$ (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)`	element shape functions at Gauss points (interior)
`shb`	`(nshapeb, ngaussb)`	element shape functions at Gauss points (boundary)
`shgl`	`(nsd, nshape, ngauss)`	local shape function gradients at Gauss points (interior)
`shglb`	`(nsd, nshapeb, nguassb)`	local 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)`???