BESHYDRO User Manual

Lipei Du[a,*]

[a]*Department of Physics, The Ohio State University, Columbus, OH 43210-1117, USA*

Contents

1 Introduction **2**

2 Utilities **2**
2.1 Installation 2
2.2 File structure 2

3 Setup **3**
3.1 Output 3
3.2 Parameters 3

4 Code design **4**
4.1 Allocation and initialization 5
4.2 Runge-Kutta Kurganov-Tadmor algorithm 5
4.3 Free memory and cleanup 6

5 Code validation **6**
5.1 The Riemann problem 7
5.2 Bjorken flow 8
5.3 Gubser flow 11
5.4 Compare to MUSIC 12

6 Dynamical initialization **13**

7 Hydro+ **15**
7.1 Code structure 15
7.2 Code validation 16

*du.458@osu.edu

1. Introduction

BESHYDRO is a (3+1)-dimensional diffusive relativistic hydrodynamic code which solves the equations of motion of second-order Denicol-Niemi-Molnar-Rischke (DNMR) theory, including bulk and shear viscous currents and baryon diffusion currents. This code is based on a CPU version of GPU-VH [1] developed by Ulrich Heinz's group at The Ohio State Uiversity.

GPU-VH code can be downloaded from `https://github.com/bazow/gpu-vh`, and BESHYDRO from `https://github.com/LipeiDu/BEShydro`. Questions and comments can be sent to Lipei Du at `du.458@osu.edu`. If you use BESHYDRO, please cite the following papers:

> [1] Dennis Bazow, Ulrich W. Heinz and Michael Strickland, Massively parallel simulations of relativistic fluid dynamics on graphics processing units with CUDA, Comput.Phys.Commun. 225 (2018) 92-113 [arXiv:1608.06577];
>
> [2] Lipei Du and Ulrich W. Heinz, (3+1)-dimensional dissipative relativistic fluid dynamics at non-zero net baryon density, [arXiv:1906.11181].

2. Utilities

2.1. Installation

The code can be downloaded from GitHub by running

```
git clone https://github.com/LipeiDu/BEShydro.git
```

on the command line. The code can be compiled using the `Makefile`, which may need modification to work on your machine. First simply typing `make clean` can delete object files and executable, and then typing

```
make
```

will compile the code. Before running the code, one should make a new directory called `output/` (where the results go), by runing

```
mkdir output
```

on the command line. Then BESHYDRO should be ready to run. To run the code, one can run

```
./beshydro --config rhic-conf/ -o output -h
```

on the command line. Here the configuration files are located under `rhic-conf/` (explained below), and outputs will be written in folder `output/`.

2.2. File structure

The code package consists of several files (`Makefile`, `README.md` and `BEShydro User Manual.pdf`) and folders (`eos/`, `input/`, `tests/`, `rhic/` and `rhic-conf/`). Folder `eos/` has Equation of State (EoS) tables, EOS3 and EOS4 discussed in Ref. [2]. Folder `input/` consists of a few folders: `coefficients/` with some transport coefficient tables, `profiles/` with some initial profiles. Configuration files are in `rhic-conf/`, where one can set parameters in `hydro.properties`, `ic.properties` and `lattice.properties`

(see Sec. 3). Folder rhic/ contains the .cpp and .h files, including freezeout/, include/ and src/ (see Sec. 4). Folder tests/ has initial profiles for doing Gubser tests or comparing to other codes. A few Mathematica notebooks are in tests/Plotting/, with which one can compare the numerical results to semi-analytical solutions to test the code (see Sec. 5).

3. Setup

Before running the code, one can set up the system by modifying the configuration files in rhic-conf/ and some parameters in the code under rhic/. In this subsection, we explain how to do this.

3.1. Output

In outputDynamicalQuantities() defined in rhic/src/HydroPlugin.cpp, one can comment out the quantities that don't need to be written in output files under output/. There are also FREQ, FOFREQ and FOTEST (explained in the code), with FREQ controlling the code to write the results in output/ every FREQ time steps. Under output/, after running the code, one will get several X_Y.dat files with X being the name of the quantity and Y the time step in the format (x, y, η_s, X). surface.dat is the freezeout surface. As an example:

```
    #define FREQ 100
    void outputDynamicalQuantities(double t, const char *outputDir,
    void * latticeParams)
{
    output(e, t, outputDir, "e", latticeParams);
    output(u->ux, t, outputDir, "ux", latticeParams);
    //output(q->nbn, t, outputDir, "nbn", latticeParams);
}
```

In FileIO.cpp, one can choose if to output distributions at $y = 0$ instead of on (x, y) 2D plane, i.e. in the format (x, η_s, X).

```
    //#define OUTPUT_SLICE
```

3.2. Parameters

Under rhic-conf/, there are 3 files containing important parameters:

- lattice.properties: here one can set number of lattice points (odd int) and lattice spacings (float) in (τ, x, y, η_s);
- ic.properties: initialConditionType is used to choose one initial condition (see the file); sourceType and numberOfSourceFiles is related to DynamicalSources.cpp as the initial condition (for dynamical initialization only, see Sec. 6); other parameters will be explained below when used in the tests;

- `hydro.properties`: `initialProperTimePoint` to set τ_0 (the initial time to start hydrodynamics); `shearViscosityToEntropyDensity` is η/s (the specific viscosity); `freezeoutTemperatureGeV` is the freezeout temperature in GeV; also there are parameters for how to initialize shear stress and bulk pressure.

Under `rhic/include/`, there are some `.h` files, where one can specify some physical or numerical parameters (the ones in `EquationOfState.h` and `DynamicalVariables.h` will be discussed seperately below):

- `FluxLimiter.h`: the parameter `THETA` is the flux limiter parameter θ_f (see Refs. [1, 2]);
- `LatticeParameters.h`: `N_GHOST_CELLS` etc. set the number of ghost cells, which should not be changed;
- `TransportCoefficients.h`: here one can set most of the transport coefficients.

Under `rhic/src/`, there are `.cpp` files, where some parameters lie in.

- `HydroPlugin.cpp`: `FREQ`, `FOFREQ` and `FOTEST` (explained above and in the code);
- `PrimaryVariables.cpp`: `MAX_ITERS` sets the maximum iterations in root-finding, and `Method_Newton` specifies if one uses Newton method to find the root;
- `SourceTerms.cpp`: `USE_CARTESIAN_COORDINATES` sets Christoffel symbols to be 0 (in ideal hydrodynamics), `MINMOD_FOR_U_AND_P` uses the flux limiter to calculate derivatives for flow velocity and pressure;
- `TransportCoefficients.cpp`: here one can set transport coefficients of bulk pressure.

4. Code design

The main structure of the code is programmed in `Run.cpp` and `HydroPlugin.cpp`. The main function `main()` is in `Run.cpp`, where the parameters are set from configuration files under `rhic-conf/`, and the code starts by calling function `run()` in `HydroPlugin.cpp`, which contains all functions needed to run a hydrodynamic simulation. Before we explain all the important functions, here we first briefly mention some `.cpp` files serving as accessories.

Copyright information is in `BEShydroLOGO.cpp`. Parameter setup is in a few files: `InitialConditionParameters.cpp`, `InitialConditionParameterTest.cpp`, `LatticeParameters.cpp`, `LatticeParameterTest.cpp`, `HydroParameters.cpp` and `HydroParameterTest.cpp` (also `CommandLineArguments.cpp` and `Properties.cpp`). Some transport coefficients in `TransportCoefficients.cpp`. Print out result files in `FileIO.cpp` and `HydroAnalysis.cpp`. Initial conditions in `InitialConditions.cpp`, `GlauberModel.cpp` and `MonteCarloGlauberModel.cpp`.

Files `DynamicalSources.cpp`, `HydroPlus.cpp`, `ToyJetClass.cpp` are for some specific undergoing studies by the author's group (please leave them untouched at this point).

4.1. Allocation and initialization

In `HydroPlugin.cpp`, function `run()` sets up the simulation:

- `LatticeParameters, InitialConditionParameters, HydroParameters`: to initialize grid spacing, grid number and physical parameters;
- `allocateHostMemory(nElements)` (in `DynamicalVariables.cpp`): allocate memory for all useful variables, where `nElements` is the total number of grids;
- `getEquationOfStateTable()` (in `EquationOfState.cpp`): read in relevant Equation of State tables when needed;
- `getBaryonDiffusionCoefficientTable()` (in `TransportCoefficients.cpp`): read in tables of baryon diffusion coefficient when necessary;
- `setInitialConditions()` (in `InitialConditions.cpp`): using different physical or simple initial conditions, this function initializes the arrays of energy density, baryon density and pressure with EoS, also shear stress, bulk pressure and baryon diffusion sometimes if necessary;
- `setConservedVariables()` (in `DynamicalVariables.cpp`): using the initialized quantities above, this function initializes energy-momentum tensor, baryon current, temperature, chemical potential and entropy density, with functions defined in `EquationOfState.cpp`; alos quantities of the previous step, like $u^\mu, \mathcal{E}, T$ etc, are initialized.
- `setGhostCells()` (in `DynamicalVariables.cpp`): initialize ghost cells at the edge to take care of derivatives (see Ref. [1]).

4.2. Runge-Kutta Kurganov-Tadmor algorithm

This part runs the loop for `nt` times to evolve the system in time, which is the most important part of the code:

- `outputDynamicalQuantities()` (in `HydroPlugin.cpp`): output dynamical quantities every `FREQ` time steps;
- `rungeKutta2()` (in `FullyDiscreteKurganovTadmorScheme.cpp`): the numerical method "two-step Runge-Kutta Kurganov-Tadmor algorithm" solving the hydrodynamic equations of motion is in this function (explained below);
- `setCurrentConservedVariables()` (in `DynamicalVariables.cpp`): call function `swap(arr1, arr2)` in the same file, mainly to swap `q` and `Q`.

The Runge-Kutta Kurganov-Tadmor algorithm `rungeKutta2()` is carried out in file `FullyDiscreteKurganovTadmorScheme.cpp`, which solves the first-order flux-conserving form,

$$\partial_\tau q + \partial_x(v^x q) + \partial_y(v^y q) + \partial_\eta(v^\eta q) = S_q \,. \tag{1}$$

All the equations of motion evolved in BESHYDRO can be written in the form above and solved in the same way (see Refs. [1, 2]).

In function `rungeKutta2()`, the two-step Runge-Kutta algorithm is carried out as follows. First, run the predicted step to estimate the slope of the variables at the current time step τ_n:

- `eulerStepKernelSource()`, `eulerStepKernelX()` etc. (see Ref. [1,2]);

- `setInferredVariablesKernel()` (in `PrimaryVariables.cpp`): this function is used to do root-finding, getting energy, pressure etc. from `qS`;

- `regulateDissipativeCurrents()` (in `RegulationScheme.cpp`): regulate dissipative currents;

- `setGhostCells()` (in `DynamicalVariables.cpp`, same as above).

Second, run the corrected step to estimate the corrected slope at time $\tau_n + \Delta\tau$:

- `eulerStepKernelSource()`, `eulerStepKernelX()` etc. again;

- `convexCombinationEulerStepKernel()` (in the same file): average predicted and corrected slopes for the conserved variables;

- `swapFluidVelocity()` and `swapPrimaryVariables()` (in `DynamicalVariables.cpp`);

- `setInferredVariablesKernel()`: calculated energy density etc.;

- `regulateDissipativeCurrents()`: regulate dissipative currents;

- `setGhostCells()`;

In `eulerStepKernelSource()`, `eulerStepKernelX()` etc., functions calculating source terms `loadSourceTerms2`, `loadSourceTermsX` etc. are called, which are defined in `SourceTerms.cpp`. These source terms contain terms $\partial_i(v^i q)$ and S_q in Eq. (1) (see Refs. [1,2] for details). In Appendix of BESHYDRO paper, the source terms are listed: $I_2^\tau, I_2^x, I_2^y, I_2^\eta$ and J_2^τ are calculated in `loadSourceTerms2`, $I_x^\tau, I_x^x, I_x^y, I_x^\eta$ and J_x^τ in `loadSourceTermsX`, $I_y^\tau, I_y^x, I_y^y, I_y^\eta$ and J_y^τ in `loadSourceTermsY`, and $I_\eta^\tau, I_\eta^x, I_\eta^y, I_\eta^\eta$ and J_η^τ in `loadSourceTermsZ`. For dissipative components, the souce terms S_2^Π, S_2^n, and S_2^π are all calculated in `setDissipativeSourceTerms` and loaded by `loadSourceTerms2`.

To calculate $\partial_i(v^i q)$ and S_q, function `setNeighborCellsJK2()` is called, where each cell together with its 4 neighbors cells, are stored in some arrays. Besides, `flux()` is called, where several files `FluxFunctions.cpp`, `FluxLimiter.cpp`, `HalfSiteExtrapolation.cpp`, `LocalPropagationSpeed.cpp`, `SpectralRadius.cpp`, `SemiDiscreteKurganovTadmorScheme.cpp` are involved.

4.3. Free memory and cleanup

After the evolution is finished, the code deallocate the memory:

- `freeHostMemory()` (in `DynamicalVariables.cpp`)

5. Code validation

In this section, we show how to do a few tests of the code. The users can rerun these tests after changing the code to make sure nothing is messed up. The Mathematica notebooks that make plots comparing the numerical results and semi-analytical ones are in `tests/Plotting/`.

5.1. The Riemann problem

To set up the code and run the Riemann problem test done in Ref. [2], under `rhic-conf/`, modify `lattice.properties`, `ic.properties` and `hydro.properties` in the following way:

```
numLatticePointsX=401
numLatticePointsY=1
numLatticePointsRapidity=1
numProperTimePoints=2000
latticeSpacingX=0.05
latticeSpacingY=0.05
latticeSpacingRapidity=0.05
latticeSpacingProperTime=0.005
```

```
initialConditionType=5
```

```
initialProperTimePoint=0.5
```

Under `rhic/src/` modify the following `.cpp` files: for `SourceTerms.cpp`, use

```
#define USE_CARTESIAN_COORDINATES
```

for `HydroPlugin.cpp`,

```
#define FREQ 50
output(p, t, outputDir, "p", latticeParams);
output(u->ux, t, outputDir, "ux", latticeParams);
output(u->ut, t, outputDir, "ut", latticeParams);
output(rhob, t, outputDir, "rhob", latticeParams);
```

Under `rhic/include/` modify the following `.h` files: for `EquationOfState.h`, use conformal EoS

```
#define CONFORMAL_EOS
```

for `FluxLimiter.h`

```
#define THETA 1.0
```

for `DynamicalVariables.h`

```
//#define PIMUNU
//#define PI
#define NBMU
//#define VMU
#define RootSolver_with_Baryon
//#define EOS_with_baryon
```

Run the code and run the notebook `tests/Plotting/Test_Riemann.nb`, one should get Fig. 1.

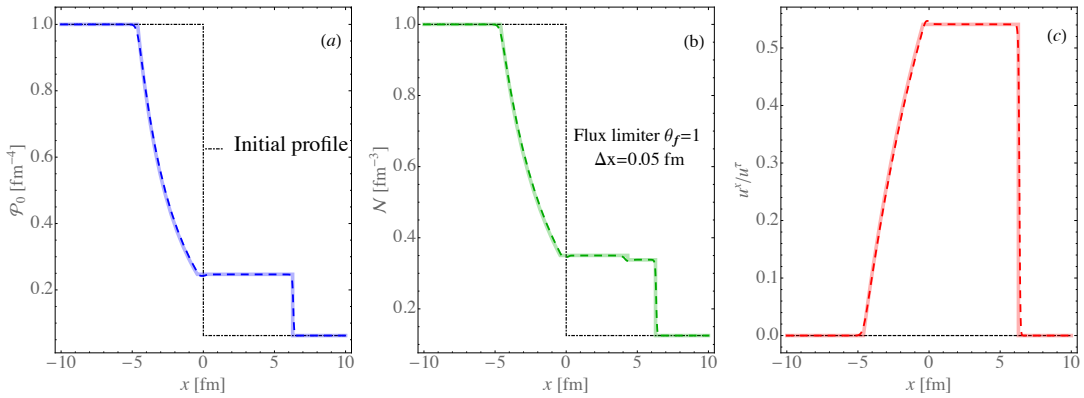

Figure 1: Analytical-numerical results comparison for the relativistic Sod's shock-tube problem in the ideal fluid with conformal EoS.

5.2. Bjorken flow

To set up the code and run the Bjorken test done in Ref. [2], under `rhic-conf/`, please modify `lattice.properties`, `ic.properties` and `hydro.properties` in the following way:

```
numLatticePointsX=2
numLatticePointsY=2
numLatticePointsRapidity=1
numProperTimePoints=2000
latticeSpacingX=0.05
latticeSpacingY=0.05
latticeSpacingRapidity=0.05
latticeSpacingProperTime=0.005
```

```
initialConditionType=15
initialBaryonDensity=500.0
initialEnergyDensity=5391.17
```

```
initialProperTimePoint=0.25
shearViscosityToEntropyDensity=0.2
initializePimunuNavierStokes=1
initializePiNavierStokes=0
```

Under rhic/src/ please modify the following .cpp files: for InitialConditions.cpp

```
    void setNbmuInitialCondition(void * latticeParams ...
    ...

#ifdef VMU
    printf("Initialize \\nb^\\mu to be zero.\n");
    for(int i = 2; i < nx+2; ++i) {
        for(int j = 2; j < ny+2; ++j) {
            for(int k = 2; k < nz+2; ++k) {
                int s = columnMajorLinearIndex(i, j, k, nx+4, ny+4);
                q->nbt[s] = 0.0;
                q->nbx[s] = 0.0;
                q->nby[s] = 0.0;
                q->nbn[s] = 10.0;
            }
        }
    }
#endif
}
```

for SourceTerms.cpp, use

```
    //#define USE_CARTESIAN_COORDINATES
```

for HydroPlugin.cpp,

```
    #define FREQ 2
    output(e, t, outputDir, "e", latticeParams);
    output(q->pinn, t, outputDir, "pinn", latticeParams);
    output(q->Pi, t, outputDir, "Pi", latticeParams);
    output(rhob, t, outputDir, "rhob", latticeParams);
    output(q->nbn, t, outputDir, "nbn", latticeParams);
```

for FullyDiscreteKurganovTadmorScheme.cpp, turn off the regulation

```
    //#ifndef IDEAL
    //        regulateDissipativeCurrents(t, Q, e, p, rhob, u, ncx,
```

```
//  ncy, ncz);
//#endif
```

Under rhic/include/ modify the following .h files: for EquationOfState.h, use EOS2

```
//#define CONFORMAL_EOS
```

for FluxLimiter.h

```
#define THETA 1.0
```

for DynamicalVariables.h

```
#define PIMUNU
#define PI
#define NBMU
#define VMU
#define RootSolver_with_Baryon
```

for TransportCoefficients.h

```
//shear transport coefficients
const PRECISION delta_pipi = 1.33333;
const PRECISION tau_pipi = 1.42857;
const PRECISION delta_PiPi = 0.666667;
const PRECISION lambda_piPi = 1.2;
const PRECISION tau_piw = 1.0;

// baryon transport coefficients
const PRECISION Cb = 4.0;
const PRECISION delta_nn = 1.0;
const PRECISION lambda_nn = 0.6;
const PRECISION tau_nw = 1.0;
```

for SourceTerms.cpp

```
PRECISION kappaB = 0.0;
```

Compile the code and run the notebook tests/Plotting/Test_Bjorken.nb, one can get plots comparing the numerical results with sem-analytical solutions.

5.3. Gubser flow

To do the Gubser flow test, we will describe what setting should be changed based on the Bjorken flow test above. If not specified, the setting can be left unchanged.

```
numLatticePointsX=201
numLatticePointsY=201
numLatticePointsRapidity=1
numProperTimePoints=500
latticeSpacingX=0.05
latticeSpacingY=0.05
latticeSpacingRapidity=0.025
latticeSpacingProperTime=0.005
```

```
initialConditionType=1
```

```
initialProperTimePoint=1.0
```

The initial profile will be read in from the file tests/Guber-test/Gubser_InitialProfile_IS_Baryon.dat.

```
#define THETA 1.8
```

```
const PRECISION tau_pipi = 0;
```

Under rhic/src/, for HydroPlugin.cpp,

```
#define FREQ 100
void outputDynamicalQuantities(double t, const char *outputDir,
void * latticeParams)
{
    output(e, t, outputDir, "e", latticeParams);
    output(u->ux, t, outputDir, "ux", latticeParams);
    output(u->ut, t, outputDir, "ut", latticeParams);
    output(q->pixx, t, outputDir, "pixx", latticeParams);
    output(q->pixy, t, outputDir, "pixy", latticeParams);
    output(q->piyy, t, outputDir, "piyy", latticeParams);
    output(q->pinn, t, outputDir, "pinn", latticeParams);
    output(rhob, t, outputDir, "rhob", latticeParams);
    output(q->nbn, t, outputDir, "nbn", latticeParams);
}
```

for FullyDiscreteKurganovTadmorScheme.cpp, turn off the regulation

```
//#ifndef IDEAL
//        regulateDissipativeCurrents(t, Q, e, p, rhob, u, ncx,
//  ncy, ncz);
//#endif
```

Under rhic/include/ please modify the following .h files: for EquationOfState.h, use conformal EoS

```
#define CONFORMAL_EOS
```

for FluxLimiter.h

```
#define THETA 1.8
```

for SourceTerms.cpp

```
PRECISION kappaB = 0.0;
```

for DynamicalVariables.h

```
#define PIMUNU
//#define PI
#define NBMU
#define VMU
#define RootSolver_with_Baryon
```

Compile the code and run the notebook tests/Plotting/Test_Gubser.nb, one can get plots comparing the numerical results with sem-analytical solutions.

5.4. Compare to MUSIC

For this test, the initial profile is read in from tests/MUSIC-test/musictest.dat. To set up the test

```
numLatticePointsX=2
numLatticePointsY=2
numLatticePointsRapidity=695
numProperTimePoints=4000
latticeSpacingX=0.05
latticeSpacingY=0.05
latticeSpacingRapidity=0.02
latticeSpacingProperTime=0.005
```

```
initialConditionType=14
```

```
initialProperTimePoint=1.0
```

```
//#define CONFORMAL_EOS
```

```
//#define PIMUNU
//#define PI
#define NBMU
#define VMU
#define RootSolver_with_Baryon
#define EOS_with_baryon
```

```
// baryon transport coefficients
const PRECISION Cb = 0.2;
const PRECISION delta_nn = 0;
const PRECISION lambda_nn = 0;
const PRECISION tau_nw = 0;
```

```
#define FREQ 200
void outputDynamicalQuantities(double t, const char *outputDir,
void * latticeParams)
{
    output(e, t, outputDir, "e", latticeParams);
    output(rhob, t, outputDir, "rhob", latticeParams);
    output(q->nbn, t, outputDir, "nbn", latticeParams);
}
```

```
PRECISION kappaB = baryonDiffusionCoefficientTest(T, rhob, alphaB);
```

Compile the code and run the notebook tests/Plotting/Test_MUSIC.nb, one can get plots comparing these two codes.

6. Dynamical initialization

In DynamicalVariables.h turn on

```
#define DYNAMICAL_SOURCE
```

then in DynamicalVariables.cpp the code allocates memory for arrays storing the dynamical sources in allocateHostMemory()

```
    // dynamical source terms
#ifdef DYNAMICAL_SOURCE
    Source = (DYNAMICAL_SOURCES *)calloc(1, sizeof(DYNAMICAL_SOURCES));
    Source->sourcet = (PRECISION *)calloc(len, bytes);
    Source->sourcex = (PRECISION *)calloc(len, bytes);
    Source->sourcey = (PRECISION *)calloc(len, bytes);
    Source->sourcen = (PRECISION *)calloc(len, bytes);
    Source->sourceb = (PRECISION *)calloc(len, bytes);
#endif
```

Under rhic-conf/ic.properties, modify the parameter shown below

```
    initialConditionType = 13
    sourceType=1
    numberOfSourceFiles=0
```

and then in InitialConditions.cpp, the code calls the function setDynamicalSourceInitialCondition() as initial condition.

In HydroPlugin.cpp, call functions in DynamicalSources.cpp to read in dynamical sources for each time step under input/dynamical-source/SourceX.dat, in which we have $(x, y, \eta_s, S_\tau, S_x, S_y, S_\eta, S_b)$.

```
    if(initialConditionType==13){
          if(n <= numberOfSourceFiles)
                readInSource(n, latticeParams, initCondParams,
                             hydroParams, rootDirectory);
          else if(n == numberOfSourceFiles + 1)
                zeroSource(latticeParams, initCondParams);
    }
```

Here after all dynamical source files for n time steps are read in, zeroSource() will set arrays of sources as 0 at $n + 1$ step.

The dynamical sources are loaded in SourceTerms.cpp by loadSourceTerms2() in the following way

```
    // added dynamical source terms
#ifdef DYNAMICAL_SOURCE
    S[0] = Source->sourcet[s] - (ttt / t + t * tnn)
         + dkvk*(pitt-p-Pi) - vx*dxp - vy*dyp - vn*dnp;
    S[1] = Source->sourcex[s] - ttx/t -dxp + dkvk*pitx;
    S[2] = Source->sourcey[s] - tty/t -dyp + dkvk*pity;
    S[3] = Source->sourcen[s] - 3*ttn/t -dnp/pow(t,2)
```

```
          + dkvk*pitn;
#else
    S[0] = - (ttt / t + t * tnn) + dkvk*(pitt-p-Pi)
          - vx*dxp - vy*dyp - vn*dnp;
    S[1] = - ttx/t -dxp + dkvk*pitx;
    S[2] = - tty/t -dyp + dkvk*pity;
    S[3] = - 3*ttn/t -dnp/pow(t,2) + dkvk*pitn;
#endif
#ifdef DYNAMICAL_SOURCE
    S[NUMBER_CONSERVED_VARIABLES] = Source->sourceb[s] - Nbt/t
          + dkvk*nbt;
#else
    S[NUMBER_CONSERVED_VARIABLES] = - Nbt/t + dkvk*nbt;
#endif
```

7. Hydro+

The Hydro+ framework [arXiv:1712.10305] is embedded in BESHYDRO: by extending the q vector to include a set of slow modes ϕ_Q, their equations of motion can be solved along with the other conserved quantities using the same RK-KT algorithm; by extending the root finder to include corrections from the slow modes to pressure Δp, the back-reaction can be taken into account. In this section, we discuss about relevant files that involve Hydro+ framework, and then show how to do a test of the code within ideal Gubser flow.

7.1. Code structure

The extra files designed for Hydro+ include: `input/profiles/xivsmuT.dat` (providing a parametrization of the correlation length $\xi(T, \mu)$), `rhic/include/HydroPlus.h` and `rhic/src/HydroPlus.cpp`, and `tests/HydroPlus-test` (providing a test of the code for Hydro+).

The simulation of equations of motion for the slow modes shares a lot of information from Sec. 4. For example, ϕ_Q and $\bar{\phi}_Q$ can be printed out by `outputDynamicalQuantities()` defined in `rhic/src/HydroPlugin.cpp`, where one can choose which mode to print out. Also, `outputHydroPlus()` defined in `rhic/src/HydroAnalysis.cpp` is to print out quantites, e.g. Γ_Q and Δp, for testing purpose at this point.

The code structure is also very similar. In `HydroPlugin.cpp`, function `run()` sets up the simulation for the slow modes:

- `allocateHostMemory()` (in `DynamicalVariables.cpp`): allocate memory for the evolving slow modes; also provide allocation for the correlation length if a parametrization needs to be read in;

- `getCorrelationLengthTable()` (in `HydroPlus.cpp`): read in a table of (T, μ, ξ) when needed;

- `setInitialConditionSlowModes()` (in `HydroPlus.cpp`): initialize the Q vector, ϕ_Q and $\bar{\phi}_Q$; it also calls `equilibriumPhiQ()` for caculating $\bar{\phi}_Q$;

- `rungeKutta2()` is called to evolve the slow modes, in the same way as dissipative quantities;

Here `rungeKutta2()` calls two important functions for the slow modes

- `eulerStepKernelSource()`, `eulerStepKernelX()` etc.;
- `setInferredVariablesKernel()` (in `PrimaryVariables.cpp`): includes the extended root finder with corrections Δp etc. from the slow mdoes;

The function `eulerStepKernelSource()` includes the source terms from the relaxation equations for ϕ_Q, where `loadSourceTerms2()` calls `setDissipativeSourceTerms()`, which contains

```
#ifdef HydroPlus
    PRECISION corrL2 = corrL * corrL;
    PRECISION gammaPhi = relaxationCoefficientPhi(rhob, seq, T, corrL2);
#endif
```

```
#ifdef HydroPlus
    for(unsigned int n = 0; n < NUMBER_SLOW_MODES; ++n)
    {
        PRECISION gammaQ = relaxationCoefficientPhiQ(gammaPhi,
                           corrL2, Qvec[n]);
        phiQRHS[n] = - 1/ut * gammaQ * (PhiQ[n] - equiPhiQ[n])
                     + PhiQ[n] * dkvk;
    }
#endif
```

where function `relaxationCoefficientPhiQ()` is used to calculate Γ_Q.

The function `setInferredVariablesKernel()` in `PrimaryVariables.cpp` calls `InferredVariablesVelocityIterationHydroPlus()` or `InferredVariables UtauIteration HydroPlus()` as the root finder. Here `getPressurePlusFromSlowModes()` in `HydroPlus.cpp` is used to calculate Δp, $\Delta\alpha$ and $\Delta\beta$ with (e, n) as input.

7.2. Code validation

In this section, we show how to do a test for the numerical methods involving Hydro+. The setup will be kept in the `HydroPlus` branch so that the test can be repeated easily. First we set

```
    initialConditionType=3
```

to run the hydro background with ideal Gubser initial condition, which is read in from `tests/Guber-test/Gubser_InitialProfile_ideal.dat`. The lattice spacings and numbers should be the same as in Sec. 5.3. Then we can output two slow modes for testing, and also energy and baryon density and flow profile can also be printed out for testing.

```
#ifdef HydroPlus
  output(q->phiQ[0], t, outputDir, "phiQ0", latticeParams);
  output(q->phiQ[1], t, outputDir, "phiQ1", latticeParams);
  //output(q->phiQ[2], t, outputDir, "phiQ2", latticeParams);
  output(eqPhiQ->phiQ[0], t, outputDir, "eqPhiQ0", latticeParams);
  output(eqPhiQ->phiQ[1], t, outputDir, "eqPhiQ1", latticeParams);
  //output(eqPhiQ->phiQ[2], t, outputDir, "eqPhiQ2", latticeParams);
#endif
```

Here ϕ_Q with the first two elements of the Q vector would be printed out. The conformal Equation of State should be turned on. Most importantly, we should turn on Hydro+ and allocate memory for it. In DynamicalVariables.h

```
/*********************************************************/
//Main switch//

//#define PIMUNU
//#define PI

#define NBMU
//#define VMU

#define RootSolver_with_Baryon
#define EOS_with_baryon

//#define DYNAMICAL_SOURCE

#define HydroPlus
#define CRITICAL
```

Here we can see the ideal fluid dynamics at non-zero baryon density is used as the background, and we use conformal EoS at non-zero baryon density, i.e., EOS3 in Ref. [2]. To do the test, $\alpha = \mu/T = 0.145$ should be used. Also in the same header file

```
/*********************************************************/
//HydroPlus extra modes//

#ifndef HydroPlus
#define NUMBER_SLOW_MODES 0
#else
#define NUMBER_SLOW_MODES 2
#endif
```

where NUMBER_SLOW_MODES is used to set the number of slow modes we evolve in the code (Here Q vector has two elements: Q[0] and Q[1]). This vector is initialized when

`setInitialConditionSlowModes()` is called, which is defined in `HydroPlus.cpp`. In that file, one can find

```
#define dQvec 0.7
#define Q0 0.5
```

which gives the first element `Q[0]` (`Q0`) and the spacing between `Q[i]` and `Q[i+1]` (`dQvec`). The `Q` vector is initialized in `setInitialConditionSlowModes()` by

```
// initialization of the Q vector
    for(unsigned int n = 0; n < NUMBER_SLOW_MODES; ++n){
        Qvec[n] = Q0 + n * dQvec;
    }
```

where one can edit it to have more flexible elements for `Q` vector.

Compile the code and run the notebook `tests/Plotting/HydroPlus_Gubser_test.nb`, one can get plots comparing the numerical results with sem-analytical solutions, which are provided under `tests/HydroPlus-test`. In the folder, the two files, `Gubser_PhiQ_0.5.dat` and `Gubser_PhiQ_1.2.dat`, give the profiles of $\phi_Q(\tau, r)$ for $Q = 0.5/\text{fm}$ and $1.2/\text{fm}$, respectively, where the four columns correspond to r and ϕ_Q at four times, $\tau = 1.0, 1.5, 2.0, 2.5$ fm.