VAH Manual

M. McNelis[a,*]

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

Contents

*Email address: mcnelis.9@osu.edu

1. Setup

The code's default `Makefile` uses the `gcc` compiler. Alternatively, you could use the other `Makefile` in `makefiles`, which uses the `icpc` compiler (also assumes you have OpenMP support). You can switch out the `Makefile` by doing[1]

```
sh makefiles.sh icpc        # or gcc
```

You may the edit the `Makefile` for your computer, but the GSL libraries need to be installed.

To set up the code on the Ohio Supercomputer Center, for example, login and do

```
git clone https://github.com/mjmcnelis/cpu_vah.git
cd cpu_vah && sh makefile.sh icpc
module load intel/19.0.3
module load python/3.6-conda5.2
```

The python scripts in `python` are used to generate model parameter samples and train (or launch) the auto–grid, but the code can run without them.

[1]Unless stated otherwise, the shell scripts here assume you are in `$HOME/cpu_vah`, where `$HOME` is your home directory.

2. Running the code

To compile and run the hydrodynamic simulation once, simply type

```
sh hydro.sh 1
```

The code will run with the default runtime and macro parameters.[2,3] The results are stored in `output` (or memory). For multiple runs, do

```
sh hydro.sh n                # n = number of events
```

The script is useful for the simpler test runs (e.g. Gubser flow) on your computer, but keep in mind that it clears `output` prior to compiling.

Alternatively, you can clear the results once and run your events (or jobs) by doing

```
sh clear_results.sh
make clean
make
for((i = 0; i < n; i++))     # n = number of events (or jobs)
do
    ./cpu_vah                # or submit job
done
```

This routine is useful for the tests that require multiple job submissions.[4]

[2] The default mode runs a 2+1d central Pb+Pb collision with nonconformal anisotropic hydrodynamics and fluctuating T_RENTo initial conditions (customized version). The freezeout surface is written to file.

[3] If you edited the parameters, you can restore the default values by running the script `default_parameters.sh`.

[4] To run the job scripts in `jobs`, you will need to change the project number and email address.

3. Parameters

Here we summarize the parameters used in the code. The runtime parameters are located in parameters. The macro parameters are located in rhic/include/Macros.h.

3.1. Runtime switches

These are the runtime switches that are typically edited (default values):

```
run_hydro = 2                          # output of simulation
// 1: hydrodynamic evolution
// 2: particlization hypersurface

kinetic_theory_model = 1               # viscous hydrodynamic model
// 0: standard m/T << 1 (VH2)
// 1: quasiparticle m(T) (VH)

temperature_etas = 1                   # model for shear viscosity
// 0: eta/s is constant
// 1: (eta/s)(T) parameterization

initial_condition_type = 4             # initial condition model
// 1: Bjorken
// 2: Gubser (ideal, viscous)
// 3: Gubser (aniso)
// 4: Trento (custom version)
// 5: read energy density from file (block format)
// 6: read energy density from memory (wrap Trento + VAH)

trento_average_over_events = 0         # event-average custom Trento
// 0: fluctuating energy density profile
// 1: smooth energy density profile

resolve_nucleons = 1                   # for Trento energy deposition
// 0: use custom lattice spacings dx, dy (see Sec. 3.6)
// 1: adjust dx, dy (and Nx, Ny) to resolve nucleon width w

auto_grid = 0                          # optimize transverse grid
// 0: use custom transverse grid (see Sec. 3.6)
// 1: adjust Nx, Ny to set transverse grid length Lx = Ly = L_auto
```

The `kinetic_theory_model` parameter sets the hydrodynamic model to VH or VH2 if you commented the `ANISO_HYDRO` macro (see next subsection).

3.2. Marco parameters

These are the macro parameters that are typically edited (default):

```
#define ANISO_HYDRO          // run anisotropic hydro
                             // comment to run viscous hydro

#define BOOST_INVARIANT      // run 2+1d hydro (eta = 0)
                             // comment to run 3+1d hydro

//#define CONFORMAL_EOS      // use QCD equation of state
                             // define to use conformal eos

//#define JETSCAPE           // write freezeout surface to file
                             // define to store surface in memory

//#define FREEZEOUT_SIZE     // fireball radius, freezeout slices,
//#define FREEZEOUT_SLICE    // runtime benchmarks and adaptive
//#define BENCHMARKS         // time step not written to file
//#define ADAPTIVE_FILE      // define to output them to file
```

3.3. Initial condition parameters

The impact parameter, T$_\mathrm{R}$ENTo parameters, rapidity plateau parameters are for the custom T$_\mathrm{R}$ENTo initial condition (default values):

```
impact_parameter = 0.0                  # b [fm]

trento_normalization_GeV = 14.2         # N [GeV] (Pb+Pb 2.76 TeV)
trento_nucleon_width = 1.12             # w [fm]
trento_min_nucleon_distance = 1.44      # d_min [fm]
trento_geometric_parameter = 0.063      # p
trento_gamma_standard_deviation = 1.05  # sigma_k

rapidity_variance = 3.24                # (sigma_eta)^2
rapidity_flat = 4.0                     # eta_flat
```

3.4. Viscosity parameters

These parameters are for the shear and bulk viscosity parameterization (default values):

```
etas_aL = -0.776                    # a_low [GeV^-1]
etas_aH = 0.37                      # a_high [GeV^-1]
etas_Tk_GeV = 0.223                 # T_eta [GeV]
etas_etask = 0.096                  # (eta/s)_kink
zetas_normalization_factor = 0.133  # (zeta/s)_max
zetas_peak_temperature_GeV = 0.12   # T_zeta [GeV]
zetas_width_GeV = 0.072             # w_zeta [GeV]
zetas_skew = -0.122                 # lambda_zeta

etas_min = 0.01                     # lower cutoff for (eta/s)(T)
constant_etas = 0.2                 # value for eta/s = constant
```

If the `temperature_etas` switch is turned off, the shear viscosity is set to `constant_etas`. Defining `CONFORMAL_EOS` sets the bulk viscosity to $\zeta/\mathcal{S} = 0$.

3.5. Hydrodynamic parameters

These parameters are for the hydrodynamic evolution (default values):

```
tau_initial = 0.05                  # starting time [fm/c]
plpt_ratio_initial = 0.3            # initial pl/pt across grid (R)

freezeout_temperature_GeV = 0.136   # switching temperature [GeV]
tau_coarse_factor = 2               # freezeout finder call period

energy_min = 1.e-1                  # energy density cutoff [fm^-4]
pressure_min = 1.e-3                # pl, pt, peq cutoff [fm^-4]

flux_limiter = 1.8                  # flux limiter in KT algorithm

regulation_scheme = 1               # switch for regulation scheme
// 0: VISHNU   1: VAH   2: None
```

Here the `tau_coarse_factor` coarse-grains the temporal resolution of the freezeout surface. For conformal systems, the lower cutoffs for the energy

density and pressures can be set to much smaller values. We recommend using the VAH regulation scheme, which is what we used in all our test runs.

3.6. Spatial grid and time step

These parameters configure the spatial grid and time step (default values):

```
lattice_points_x = 281      # custom grid points Nx
lattice_points_y = 281      # custom grid points Ny
lattice_points_eta = 1      # custom grid points Neta

lattice_spacing_x = 0.1     # custom lattice spacing dx [fm]
lattice_spacing_y = 0.1     # custom lattice spacing dy [fm]
lattice_spacing_eta = 0.1   # custom lattice spacing deta [1]

fit_rapidity_plateau = 0    # switch for eta plateau extension
// 0: use custom Neta, deta
// 1: adjust Neta, deta to fit and resolve plateau

training_grid = 0           # switch for training auto-grid
// 0: use custom transverse grid
// 1: adjust Nx, Ny to set transverse grid length Lx = Ly = 30 fm

train_coarse_factor = 2.0   # coarse-grain training grid only

sigma_factor = 0.0          # margin controls for auto-grid
buffer = 2.5

adaptive_time_step = 1      # switch for adaptive time step
// 0: dt is constant
// 1: dt is adaptive

delta_0 = 0.004             # controls for adaptive time step
alpha = 0.5

fixed_time_step = 0.0125    # value for dt = constant [fm/c]

max_time_steps = 2000       # code fails if time steps exceed this
```

With the first six parameters, you can customize the grid size and spatial resolution.[5] Here the custom transverse grid has dimensions $L_x = L_y = 28$ fm and spatial resolution $\Delta x = \Delta y = 0.1$ fm (make sure to use an odd number of lattice points to center the grid at the origin).

There are several switches that overwrite the custom transverse lattice points and/or spacings:

1. Turning on the `resolve_nucleons` switch sets `lattice_spacing_x` and `lattice_spacing_y` to $0.2 \times$ `trento_nucleon_width`. In additional, the lattice points `lattice_points_x` and `lattice_points_y` are adjusted to more or less keep the custom (training or auto) grid size.
2. If the `training_grid` switch is turned on, the transverse lattice spacings are further rescaled by `train_coarse_factor`. Then the transverse lattice points are adjusted so that the grid size is about $L_x = L_y = 30$ fm.[6]
3. If the `auto_grid` switch is turned on (and `training_grid` turned off), the transverse lattice points are adjusted so that the grid size is about $L_x = L_y = L_{\text{auto}}$ (see Eq. (1)). You can increase the margins of the auto–grid with the parameters `sigma_factor` and `buffer` (see Eq. (1)).

If `BOOST_INVARIANT` is commented, the `fit_rapidity_plateau` switch can be turned on to overwrite `lattice_points_eta` and `lattice_spacing_eta` to fit and resolve the custom plateau extension used in the paper.

Turn on `adaptive_time_step` to implement the adaptive time step algorithm. You can adjust the error tolerance `delta_0` and safety `alpha` to control the growth rate of $\Delta\tau_n$ at early times (the default values work fine).

Alternatively, you could use a fixed time step, but this is not recommended. If you turn off `adaptive_time_step`, the time step $\Delta\tau$ is set to `fixed_time_step`.[7]

3.7. Random model parameters

You can replace the impact parameter b (if `initial_condition_type` is set to 4) and Bayesian model parameters P_B with randomized values (within

[5] If `BOOST_INVARIANT` is defined, then `lattice_points_eta` is set to 1.

[6] Note that even if the `training_grid` (or `auto_grid`) switch is on, it is only activated if the random model parameters are read in (see Sec. 3.7).

[7] This must satisfy the CFL condition `fixed_time_step` $\leq \frac{1}{8}\min(\Delta x, \Delta y, \Delta\eta_s)$.

finite intervals) during runtime. To generate random parameter samples, do

```
cd scripts/auto_grid
sh sample_model_parameters.sh s
```

where `s` is the number of samples. The parameter samples are stored in `python/model_parameters`.

Then to run the simulation with one of these samples, do

```
sh hydro.sh 1 p              # or ./cpu_vah p
```

where `p` $\in$ `[1,s]` is the parameter sample. The other runtime parameters remain unchanged.

4. Source code

The source and header files can be found in `rhic`. We briefly summarize the purpose of each file and list the main classes, structs and functions.[8] Hopefully this will be helpful for those who are looking to get familiar with the inner workings of the code and update it in the future.

4.1. Simulation

`Main.cpp`
`Hydrowrapper.cpp`

Creates an instance of the `HYDRO` class, which runs the hydrodynamic simulation and stores the particlization hypersurface (or outputs results to file). The wrapper can run as a stand-alone program or integrated into a larger program (e.g. JETSCAPE).

`HYDRO`

`start_hydro_no_arguments()`
`store_freezeout_surface()`
`free_freezeout_surface()`

`Parameters.cpp`

Reads the parameter files in `parameters` and sets the runtime parameters in the structs below. The impact parameter and Bayesian model parameters can be overwritten by the random samples in `python/model_parameters`.

`hydro_parameters`
`initial_condition_parameters`
`lattice_parameters`

`load_hydro_parameters()`
`load_initial_condition_parameters()`
`load_lattice_parameters()`

[8] Any function arguments are not listed here.

`DynamicalVariables.cpp`

Allocates memory for the dynamical and inferred variables on the spatial grid at a given time. The code makes use of `extern` variables and structs to store and access the hydrodynamic quantities.

`hydro_variables`
`fluid_velocity`

`allocate_memory()`
`free_memory()`

`InitialConditions.cpp`

Sets the initial conditions for the dynamical and inferred variables. The energy density is the most sensitive to the type of initial-state model.

`set_initial_conditions()`

`Hydrodynamics.cpp`

Configures the spatial grid and evolves the hydrodynamic equations until all fluid cells are below the switching temperature.

`run_hydro()`
`all_cells_below_freezeout_temperature()`

`KurganovTadmor.cpp`

Computes the intermediate Euler steps in the Runge–Kutta scheme using the Kurganov–Tadmor algorithm. The hydrodynamic variables are then updated with the averaged iteration.

`evolve_hydro_one_time_step()`
`euler_step()`

`FluxTerms.cpp`

Computes the flux terms in the Kurganov–Tadmor algorithm.

`flux_terms()`

`SourceTerms.cpp`

Computes the source terms in the hydrodynamic equations.

`source_terms_aniso_hydro()`
`source_terms_viscous_hydro()`

`NeighborCells.cpp`

Collects the neighbor cells of the fluid cell being evaluated. These are used for the numerical spatial derivatives in the flux and source terms.

`get_hydro_variables_neighbor_cells()`
`get_fluid_velocity_neighbor_cells()`

`Projections.cpp`

The `projection` classes compute and store the spatial (or transverse) projection tensors. These are used to spatially (or transversely) project vectors or rank-2 tensors that appear in the source terms.

`InferredVariables.cpp`

Reconstructs the energy density and fluid velocity after each iteration.

`set_inferred_variables_aniso_hydro()`
`set_inferred_variables_viscous_hydro()`

`AnisoVariables.cpp`

Reconstructs the anisotropic variables after each iteration. The anisotropic variables of a single fluid cell are stored in the `aniso_variables` struct.

aniso_variables

find_anisotropic_variables()
set_anisotropic_variables()

Regulation.cpp

Regulates the residual shear stress and mean-field (or standard shear stress and bulk viscous pressure) after each iteration.

regulate_residual_currents()
regulate_viscous_currents()

GhostCells.cpp

Sets the boundary conditions for the ghost cell layers interfaced with the physical spatial grid.

set_ghost_cells()

AdaptiveTimeStep.cpp

Computes the adaptive time step for the next Runge–Kutta iteration.

compute_dt_source()
compute_dt_CFL()

FreezeoutFinder.cpp

The freezeout_finder class holds the hydrodynamic variables from the current and previous spatial grids. The energy density hypercubes are then constructed and passed to the Cornelius class from cornelius-c++-1.3 to search for freezeout cells. The freezeout cells' centroid position and surface element vector, along with the interpolated hydrodynamic variables, are appended to the freezeout_surface struct.

freezeout_finder
Cornelius

freezeout_surface

load_initial_grid()
load_current_grid()
find_3d_freezeout_cells()

4.2. Semi-analytic

AnisoBjorken.cpp
ViscousBjorken.cpp

- Evolve the (non)conformal Bjorken semi-analytic solutions for anisotropic and viscous hydrodynamics

run_semi_analytic_aniso_bjorken()
run_semi_analytic_viscous_bjorken()

AnisoGubser.cpp
ViscousGubser.cpp
IdealGubser.cpp

- Evolve the conformal Gubser semi-analytic solutions for anisotropic, viscous and ideal hydrodynamics.

run_semi_analytic_aniso_gubser()
run_semi_analytic_viscous_gubser()
run_analytic_ideal_gubser()

4.3. Equation of state and transport coefficients

EquationOfState.cpp

The constructor of the equation_of_state class reads in the energy density and computes the equilibrium temperature. Other thermodynamic variables such as the equilibrium pressure and beta transport coefficients can then be evaluated.

equation_of_state

equilibrium_pressure()
beta_shear()
beta_bulk()

Viscosities.cpp

Computes the shear and bulk viscosities $(\eta/\mathcal{S})(T)$ and $(\zeta/\mathcal{S})(T)$.

eta_over_s()
zeta_over_s()

TransportViscous.cpp

Compute the second-order transport coefficients in viscous hydrodynamics.

viscous_transport_coefficients

compute_shear_transport_coefficients()
compute_bulk_transport_coefficients()

TransportAniso.cpp
TransportAnisoNonconformal.cpp

Compute the conformal and nonconformal transport coefficients in anisotropic hydrodynamics.

aniso_transport_coefficients
aniso_transport_coefficients_nonconformal

compute_transport_coefficients()

4.4. Initial conditions

Trento.cpp

A simpler, customized version of the T$_{\mathrm{R}}$ENTo code to make fluctuating (or event-averaged) initial energy density profiles.

`set_trento_energy_density_and_flow_profile()`

4.5. Other

`Output.cpp`

Outputs the hydrodynamic evolution of the simulation (or semi-analytic solution) to file. This is used for the code test runs and comparison studies.

`output_hydro_simulation()`

`Print.cpp`

Prints the hydrodynamic simulation model, parameters, runtime status (at the center of the grid) and benchmarks.

`Memory.cpp`

Allocates memory for multi-dimensional arrays. These are mainly used by the routines in `freezeout_finder`.

`Precision.h`

The macro variable type `precision` determines the numerical precision of float-type variables in the program.[9] Almost all the float-type variables in the code are declared as `precision`.

[9] Only the variable type `double` works at the moment.

5. Workflow

The program creates an instance of the HYDRO class to run the hydrodynamic simulation.[10,11] If the simulation is successful, the particlization hypersurface is either stored in the HYDRO wrapper (so that it can be passed to another program via memory) or written to the file output/surface.dat.[12] The JETSCAPE macro determines the method of output. In either case, we deallocate the freezeout surface at the end of the program.

```
int main(..)
{
    HYDRO vah;                              // hydro wrapper
    vah.start_hydro_no_arguments();         // start hydro simulation
    vah.free_freezeout_surface();           // deallocate hypersurface
    return 0;
}
```

By default, VAH runs as a stand-alone program, but the HYDRO wrapper allows it to be integrated into a larger program such as the JETSCAPE framework.

At the start of the simulation, we read in the runtime parameters from the files in parameters. They are organized into three categories: `hydro`, `lattice` and `initial`. After configuring the grid, we pass the parameters to run_hydro(..), which is the heart of the program. The function returns a freezeout_surface struct containing the freezeout cells' information. If JETSCAPE is defined, the store_freezeout_surface(..) function loads the freezeout surface members to individual vectors in the HYDRO class. If JETSCAPE is commented, we write the freezeout surface to file (not shown) with the following format:

$$\tau \;\; x \;\; y \;\; \eta_s \;\; d^3\sigma_\tau \;\; d^3\sigma_x \;\; d^3\sigma_y \;\; d^3\sigma_\eta \;\; u^x \;\; u^y \;\; u^\eta \;\; \mathcal{E} \;\; T \;\; \mathcal{P}_{\rm eq} \;\; \pi^{xx} \;\; \pi^{xy} \;\; \pi^{x\eta} \;\; \pi^{yy} \;\; \pi^{y\eta} \;\; \Pi$$

[10] In this section we oversimplify the code blocks to highlight the main features of the program.

[11] Unless stated otherwise, the arguments of each function are represented by (..).

[12] If the code crashes or does not finish within the allotted time, the freezeout surface file will be empty.

```
void HYDRO::store_freezeout_surface(freezeout_surface surface)
{
    for(long i = 0; i < total_cells; i++)
    {
        tau.push_back(surface.tau[i]);              // x^\mu
        x.push_back(surface.x[i]);
        y.push_back(surface.y[i]);
        eta.push_back(surface.eta[i]);
        dsigma_tau.push_back(surface.dsigma_tau[i]);// dsigma_\mu
        dsigma_x.push_back(surface.dsigma_x[i]);
        dsigma_y.push_back(surface.dsigma_y[i]);
        dsigma_eta.push_back(surface.dsigma_eta[i]);
        ux.push_back(surface.ux[i]);                // u^\mu
        uy.push_back(surface.uy[i]);
        un.push_back(surface.un[i]);
        E.push_back(surface.E[i]);                  // e
        T.push_back(surface.T[i]);                  // T
        P.push_back(surface.P[i]);                  // Peq
        pixx.push_back(surface.pixx[i]);            // pi^\munu
        pixy.push_back(surface.pixy[i]);
        pixn.push_back(surface.pixn[i]);
        piyy.push_back(surface.piyy[i]);
        piyn.push_back(surface.piyn[i]);
        Pi.push_back(surface.Pi[i]);                // Pi
    }
}
void HYDRO::start_hydro_no_arguments()
{
    // read runtime parameters
    hydro_parameters hydro = load_hydro_parameters(..);
    lattice_parameters lattice = load_lattice_parameters(..);
    initial_condition_parameters initial;
    initial = load_initial_condition_parameters(..);

    // run hydro simulation and store hypersurface
    store_freezeout_surface(run_hydro(..));
}
```

The next code block contains the main features of the `run_hydro(..)` function. The workflow is explained in Sec. 3.7 of the code documentation paper but we summarize the steps again here:

1. We allocate memory for the dynamical and inferred variables on the spatial grid.
2. We set the initial time τ_0 to `tau_initial` and the initial time step $\Delta\tau_0 = 0.05\,\tau_0$ (or `fixed_time_step`).
3. We set the initial conditions and ghost cell boundary conditions.
4. We configure the `freezeout_finder` class and load the initial grid.
5. We start the hydrodynamic evolution loop:
   (a) We compute the adaptive time step (or use the fixed time step).
   (b) We call the freezeout finder every `tau_coarse_factor` time steps and search for freezeout cells. If the entire grid is below the switching temperature given by `freezeout_temperature_GeV`, we stop the evolution.
   (c) We evolve the system one time step with the Runge–Kutta scheme.
6. After the simulation finishes, we deallocate the spatial grid and return the freezeout surface.

The function `evolve_hydro_one_time_step(..)` computes the second-order Runge–Kutta iteration of each time step:

1. We compute the first intermediate Euler step with the KT algorithm and swap the fluid velocity variables `u` and `up`.
2. We solve for the intermediate inferred variables and perform the regulation. We also set the ghost cells.
3. Similarly, we compute the second intermediate Euler step and store the RK2 update in `Q`.
4. Afterwards, we solve for the updated inferred variables and do the regulation.
5. Finally, we swap the dynamical variables `q` and `Q` and set the ghost cells for the next time step.

We have not yet integrated VAH into the JETSCAPE framework. We plan to set `initial_condition_type` = 6 to combine it with the T$_{\mathrm{R}}$ENTo module, but the exact implementation is still in the works. The functionality of the `JETSCAPE` macro is also subject to change (stay tuned).

```
freezeout_surface run_hydro(..)
{
    allocate_memory(..);            // grid allocation

    double t = tau_initial;         // starting time
    double dt = 0.05 * t;           // initial time step

    if(!adaptive_time_step)
    {
        dt = fixed_time_step;
    }
    set_initial_conditions(..);     // initialize (q,e,u,up)
    set_ghost_cells(..);            // for (q,e,u)

    freezeout_finder finder(..);    // initialize freezeout finder
    finder.load_initial_grid(..);

    for(int n = 0; n < max_time_steps; n++)
    {
        dt = set_time_step(..);         // adaptive time step

        // call finder to search for freezeout cells
        if(n > 0 && (n % tau_coarse_factor) == 0)
        {
            finder.load_current_grid(..);
            finder.find_2d_freezeout_cells(..);

            if(all_cells_below_freezeout_temperature(..))
            {
                break;                  // stop hydro evolution
            }
        }
        evolve_hydro_one_time_step(..); // RK2 iteration
        t += dt;
    }
    free_memory();                  // deallocate grid
    return finder.surface;
}
```

```
void evolve_hydro_one_time_step(..)
{
    // first intermediate Euler step
    euler_step(..);                          // qI <= q + E.dt
    swap_fluid_velocity(..);                 // swap u <=> up
    set_inferred_variables_aniso_hydro(..); // compute (e,u)
    set_anisotropic_variables(..);           // compute X
    regulate_residual_currents(..);          // regulate qI
    set_ghost_cells(..);                     // for (qI,e,u)

    // second intermediate Euler step
    euler_step(..);                          // Q <= RK2 update
    set_inferred_variables_aniso_hydro(..); // compute (e,u)
    set_anisotropic_variables(..);           // compute X
    regulate_residual_currents(..);          // regulate Q
    swap_hydro_variables(..);                // swap q <=> Q
    set_ghost_cells(..);                     // for (q,e,u)
}
```

6. Tests and model comparison studies

Several initial condition models are built into the code. They are primarily used for testing the simulation and comparing hydrodynamic models.

6.1. Conformal Bjorken flow test

To run anisotropic hydrodynamics with conformal Bjorken flow, adjust the following parameters and MACROS:

```
run_hydro = 1
tau_initial = 0.01
pl_pt_ratio_initial = 0.001
temperature_etas = 0
freezeout_temperature_GeV = 0.136
energy_min = 1.e-4
pressure_min = 1.e-6

initial_condition_type = 1
initial_central_temperature_GeV = 1.05

lattice_points_x = 1
lattice_points_y = 1
```

```
#define BOOST_INVARIANT
#define CONFORMAL
#define ADAPTIVE_FILE
```

Here initial_central_temperature_GeV sets the initial temperature at the center of the grid (for Bjorken and Gubser initial conditions only). The run_hydro = 1 mode outputs both the simulation results and semi-analytic solution.

Alternatively, you can run the conformal Bjorken test by doing

```
cd scripts/conformal_bjorken
sh run_conformal_bjorken_test.sh
```

which copies the files from tests/conformal_bjorken/parameters to the appropriate directories and runs the code. The test results are stored in tests/conformal_bjorken. You can use the Mathematica notebook to plot them.

6.2. Conformal Gubser flow test

To run anisotropic hydrodynamics with conformal Gubser flow, adjust the parameters

```
run_hydro = 1
tau_initial = 0.01
pl_pt_ratio_initial = 0.001
temperature_etas = 0
freezeout_temperature_GeV = 0.065
energy_min = 1.e-4
pressure_min = 1.e-6

initial_condition_type = 3
initial_central_temperature_GeV = 1.05

lattice_points_x = 281
lattice_points_y = 281
lattice_spacing_x = 0.05
lattice_spacing_y = 0.05
```

```
#define BOOST_INVARIANT
#define CONFORMAL
#define ADAPTIVE_FILE
```

Note that (for Gubser flow only) `pl_pt_ratio_initial` corresponds to the initial pressure ratio $\mathcal{P}_L/\mathcal{P}_\perp$ at the corners of the transverse grid ($\mathcal{P}_L/\mathcal{P}_\perp$ is about $10\times$ larger in the central region).

You can also run the script

```
cd scripts/conformal_gubser
sh run_conformal_gubser_test.sh
```

and plot the simulation results, as well as the semi-analytic solution, with the Mathematica notebook in `tests/conformal_gubser`.

6.3. Nonconformal Bjorken flow test

To run nonconformal anisotropic hydrodynamics with Bjorken flow, adjust the parameters

```
run_hydro = 1
tau_initial = 0.05
pl_pt_ratio_initial = 0.3
kinetic_theory_model = 0 or 1
temperature_etas = 1
freezeout_temperature_GeV = 0.136
energy_min = 1.e-1
pressure_min = 1.e-3

initial_condition_type = 1
initial_central_temperature_GeV = 0.718

lattice_points_x = 1
lattice_points_y = 1
```

```
#define ANISO_HYDRO
#define BOOST_INVARIANT
//#define CONFORMAL
#define ADAPTIVE_FILE
```

Make sure you comment `CONFORMAL` to use the QCD equation of state. To run the nonconformal second-order viscous hydrodynamic models, comment `ANISO_HYDRO` and set `kinetic_theory_model` to either 0 (VH2) or 1 (VH).

You can also do

```
cd scripts/lattice_bjorken
sh run_lattice_bjorken_test.sh vah      # or vh, vh2
```

to run the test with one of the three hydrodynamic models. Once you run all three tests, plot them using the notebook in `tests/lattice_bjorken`.

6.4. Conformal hydrodynamic models with smooth T$_R$ENTo *profile*

To run (3+1)–dimensional conformal anisotropic hydrodynamics with smooth T$_R$ENTo initial conditions, adjust the parameters

```
run_hydro = 1
tau_initial = 0.01
pl_pt_ratio_initial = 0.001
temperature_etas = 0
freezeout_temperature_GeV = 0.165
energy_min = 1.e-4
pressure_min = 1.e-7

initial_condition_type = 4
trento_average_over_events = 1

lattice_points_x = 281
lattice_points_y = 281
lattice_spacing_x = 0.1
lattice_spacing_y = 0.1
resolve_nucleons = 1
fit_rapidity_plateau = 1
```

```
#define ANISO_HYDRO
//#define BOOST_INVARIANT
#define CONFORMAL
#define ADAPTIVE_FILE
```

Turning on the `trento_average_over_events` switch event-averages multiple fluctuating T$_R$ENTo events to make a smooth initial energy density profile.

Make sure you comment `BOOST_INVARIANT` to run the code in (3+1)–dimensions. If you want to run conformal standard viscous hydrodynamics, comment `ANISO_HYDRO`.

You can also run the script

```
cd scripts/conformal_trento
sh run_conformal_trento_job.sh vah      # or vh
```

which submits a job to run the code with OpenMP acceleration. If you don't have access to a computing node, you can run

```
cd scripts/conformal_trento
sh run_conformal_trento_local.sh vah    # or vh
```

on your computer, although it would take much longer. Once you run the anisotropic and viscous hydrodynamic models, you can compare them in the notebook in `tests/conformal_trento`.

6.5. Nonconformal hydrodynamic models with smooth T$_R$ENTo *profile*

To run (3+1)–dimensional nonconformal anisotropic hydrodynamics with smooth T$_R$ENTo initial conditions, adjust the parameters

```
run_hydro = 1
tau_initial = 0.05
pl_pt_ratio_initial = 0.3
kinetic_theory_model = 0 or 1
temperature_etas = 1
freezeout_temperature_GeV = 0.136
energy_min = 1.e-1
pressure_min = 1.e-3

initial_condition_type = 4
trento_average_over_events = 1

lattice_points_x = 281
lattice_points_y = 281
lattice_spacing_x = 0.1
lattice_spacing_y = 0.1
resolve_nucleons = 1
fit_rapidity_plateau = 1
```

```
#define ANISO_HYDRO
//#define BOOST_INVARIANT
//#define CONFORMAL
#define ADAPTIVE_FILE
```

Similar to the previous run in Sec. 6.4, you can do

```
cd scripts/lattice_trento_smooth
sh run_lattice_trento_smooth_job.sh vah     # or vh, vh2
```

to submit a job to a computing node or

```
cd scripts/lattice_trento_smooth
sh run_lattice_trento_smooth_local.sh vah   # or vh, vh2
```

to run the code on your computer. After running all three hydrodynamic models, you can plot the results in `tests/lattice_trento_smooth`.

6.6. Nonconformal hydrodynamic models with fluctuating T$_R$ENTo *profile*

To run (3+1)–dimensional nonconformal anisotropic hydrodynamics with fluctuating T$_R$ENTo initial conditions, adjust the parameters

```
run_hydro = 1
tau_initial = 0.05
pl_pt_ratio_initial = 0.3
kinetic_theory_model = 0 or 1
temperature_etas = 1
freezeout_temperature_GeV = 0.136
energy_min = 1.e-1
pressure_min = 1.e-3

initial_condition_type = 4
trento_average_over_events = 0
trento_fixed_seed = 1

lattice_points_x = 281
lattice_points_y = 281
lattice_spacing_x = 0.1
lattice_spacing_y = 0.1
resolve_nucleons = 1
fit_rapidity_plateau = 1
```

```
#define ANISO_HYDRO
//#define BOOST_INVARIANT
//#define CONFORMAL
#define FREEZEOUT_SLICE
```

Make sure you turn off `trento_average_over_events` to use a fluctuating energy density profile. You also need to turn on `trento_fixed_seed` to fix the seed for reproducible results. Defining `FREEZEOUT_SLICE` outputs the freezeout surface slices for the inverse Reynolds number plots.

You can also run

```
cd scripts/lattice_trento_fluctuating
sh run_lattice_trento_fluctuating_job.sh vah     # or vh, vh2
```

or

```
cd scripts/lattice_trento_fluctuating
sh run_lattice_trento_fluctuating_local.sh vah   # or vh, vh2
```

Once the simulations are finished, you can compare them with the two Mathematica notebooks in `tests/lattice_trento_fluctuating`.

7. Auto–grid

7.1. Launching the transverse auto–grid

The regression models used to configure the transverse auto–grid are located in `tests/auto_grid/regression_model`. To launch them, do

```
cd scripts/auto_grid
sh predict_fireball_radius.sh h s n
```

where `h` ∈ `[vah, vh, vh2]` is the hydrodynamic model, `s` is the number of new model parameter samples (see Sec. 3.7) and `n` is the number of hydrodynamic events per job used to generate the training data (we used `n` = 1). The newly generated parameter samples and fireball radius predictions are stored in `python/model_parameters` and `python/fireball_size_predictions`, respectively.

Finally, turn on the `auto_grid` switch and run

```
sh hydro.sh 1 p
```

where `p` ∈ `[1,s]` is the model parameter sample. For the pre-trained auto–grid, leave `sigma_factor` as 0 and `buffer` as 2.5.

7.2. Training the auto–grid

The transverse auto–grid for the three hydrodynamic models have already been trained, but you can retrain them with more training data (or to include statistical fluctuations in the fireball radius) for increased accuracy. Later down the road, you may want to update the auto–grid algorithm for 3+1d hydrodynamics.

The training routine will require access to multiple computing nodes to generate the training data quickly.[13]

7.2.1. Training data

To generate the training data, do

```
cd scripts/auto_grid/generate_training_data
sh generate_training_data_smooth.sh h t 1
```

[13]For even faster training, you can coarse-grain your training grid by the factor `training_coarse_factor` (we set it to 2), although the fireball radius data will be less precise (especially if you are using fluctuating initial conditions).

where `h` ∈ `[vah, vh, vh2]` is the hydrodynamic model (e.g. `vah`) and `t` is the number of training parameter samples (we used 1000). The script submits one job per training sample and evolves a smooth T_RENTo profile on the 30 fm × 30 fm training grid (the `training_grid` switch is turned on) with the selected hydrodynamic model and training parameters.[14]

After the jobs have finished, the fireball radius data from the training runs will be stored in `tests/auto_grid/train_data/$h` (`FREEZEOUT_SIZE` is defined).

Alternatively, you can run this script to include statistical fluctuations in the fireball radius for a given parameter sample:

```
cd scripts/auto_grid/generate_training_data
sh generate_training_data_fluctuating.sh h t n
```

where `n` is the number of fluctuating hydrodynamic events per training sample. However, it takes much more computing resources. As a first step, it would be much easier to use smooth initial conditions to generate the training data (which is what we did).

7.2.2. Regression model

After generating the training data, you can fit the regression model by doing

```
cd scripts/auto_grid
sh fit_regression_model.sh h t n
```

where the command arguments are the same as before (i.e. `vah 1000 1`). The script processes the training data[15] and optimizes the regression model. The model for the mean fireball radius $\bar{r}$ and its cross-validated error $\delta\bar{r}_{\mathrm{RMSE}}$ are saved to `tests/auto_grid/regression_model/$h`.[16]

[14] The default training routine uses the custom T_RENTo model (Pb+Pb 2.76 TeV) to generate the initial conditions. If you want to train the auto-grid for a different collision system, you will need to read in the energy density profile from an initial-state module.

[15] After processing the training data, you can visualize it with the Jupyter notebook `visualize_training_data.ipynb` in `python`.

[16] If you generated the training data with fluctuating initial conditions, a second regression model predicting the standard deviation σ_r will be saved along with its cross-validated error $\delta\sigma_{r,\mathrm{RMSE}}$.

Finally, you can go back to Sec. 7.1 and launch the newly trained model (the second argument `s` can vary but use the same values for the remaining arguments (e.g. `vah 40 1`).

7.2.3. Performance

The remaining scripts evaluate the auto–grid's performance but they take up considerable computing resources. If you find the model errors to be acceptable, you can skip this section. However, you will probably want to verify that the auto–grid is fitting the fireball correctly.

First we need to generate the test data. We already have 200 test parameter samples in `tests/auto_grid/benchmark_test/model_parameters`. To run fluctuating hydrodynamic events on the fixed grid with these test samples, do

```
cd scripts/auto_grid
sh benchmark_fixed_grid.sh h m
```

where `m` is the number of fluctuating events per test parameter sample (we used 50 but you can choose a smaller number like 5). Once the jobs are done, the fireball radius test data (as well as runtime benchmarks) will be stored in `tests/auto_grid/benchmark_test/fixed_grid/$h` (`FREEZEOUT_SIZE` and `BENCHMARKS` are defined).

Then to test the performance of the auto–grid, do

```
cd scripts/auto_grid
sh benchmark_auto_grid.sh h m n b f
```

where the first three arguments are the same as before (`h m n = vah 50 1`). In addition, the argument `b` corresponds to the `buffer` parameter and `f` the `sigma_factor` parameter (we used `b` = 2.5 and `f` = 0).[17]

The script launches the regression model for the test parameter samples and prints the overall fireball fit success rate (e.g. 99%) if you were to repeat

[17] The formula for the auto transverse grid length is

$$L_{\rm auto} = 2\,[\bar{r}_{\rm pred} + \delta\bar{r}_{\rm RMSE} \;+\; \texttt{sigma\_factor}(\sigma_{r,\rm pred} + \delta\sigma_{r,\rm RMSE}) \;+\; \texttt{buffer}]\;, \qquad (1)$$

where $\bar{r}_{\rm pred}$ and $\sigma_{r,\rm pred}$ are the predicted mean fireball radius and standard deviation. (for our case $\sigma_{r,\rm pred} = \delta\sigma_{r,\rm RMSE} = 0$).

the previous test run with the auto–grid. It also prints the average grid area reduction relative to the fixed grid (to give you an estimate for the speedup).

You will then receive a prompt that asks you whether you want to submit the jobs for the auto–grid run. If you are not satisfied with the fit success rate or grid area reduction, say no and readjust the arguments `b` (`buffer`) and `f` (`sigma_factor`) in the previous script until the auto–grid margins are optimized.

Otherwise, the job submission is completely optional since it only outputs the auto–grid runtimes. However, if you want to produce the runtime data for the benchmark comparison, make sure you edit the `sigma_factor` and `buffer` parameters in `parameters/lattice.properties` before saying yes to submit the batch.

After the jobs are done, you can compare the fixed–grid and auto–grid runtimes with the Juptyer notebook `benchmark_runtimes.ipynb` in `python`.

8. Acknowledgements

Special thanks to Lipei Du for providing the BESHYDRO manual script as a base for this manual.