`closures-2d`
Design Documentation

C. Kristopher Garrett, Tim Shaffer

July 28, 2015

Contents

1 Purpose

This software implements various numerical methods for approximating the angular variable in kinetic transport in order to study performance, accuracy, and robustness on common test problems. Whereas fluid equations describe a vector of quantities dependent on space, kinetic transport equations describe a scalar density of particles depending on both space and velocity. Such numerical methods occur frequently in production codes modeling certain physical systems, such as neutron transport inside a reactor, neutrino transport in supernova simulations, photon transport, and rarefied gas dynamics. Rather than simulating a specific physical system, this software solves simpler problems that capture issues commonly encountered in full simulations. Comparing the performance of these numerical methods on our simplified problems can give researchers insight into the behavior of similar methods in more complicated implementations.

In addition to evaluating the methods themselves, this software is written to allow study of performance with modern high-performance computing resources. This software optionally uses OpenMP and Message Passing Interface (MPI) to carry out parallel and concurrent computations. Studying the behavior of numerical methods with such acceleration techniques provides reference for implementations on large-scale problems.

This software also supports profiling measurements such as timings, flop rates, memory rates, and cache misses. Using statistics about runtime performance, we examined code optimizations, bottlenecks, etc. that occur in our implementations.

2 Problem

This software simulates particles with unit speed that scatter isotropically according to a simple kinetic equation. The equation governing particle behavior takes the form

$$\partial_t f + \Omega \cdot \nabla_x f + \sigma_t f = \frac{\sigma_s}{4\pi}\langle f \rangle \tag{1}$$

where $f(x, \Omega, t)$ is the density of particles, $x \in \mathcal{D} \subset \mathbb{R}^2$ is position, $\Omega \in \mathbb{S}^2 \subset \mathbb{R}^3$ (the unit sphere) is the direction of travel, $\sigma_t(x) \geq \sigma_s(x)$ are the total and scattering cross sections, and $\langle \cdot \rangle$ is shorthand for integration over $\mathbb{S}^2$. The boundary conditions implemented are either zero inflow boundary conditions or periodic boundary conditions depending on the initial condition discussed in Section 2.1.

2.1 Initial Conditions

The following initial conditions are implemented in this software.

2.1.1 Gaussian Initial Condition

In the Gaussian initial condition, a two dimensional Gaussian function is placed with its peak at the center of the grid. Each point on the initial grid is set to

$$f(x, y, \Omega, 0) = \max\left(\frac{1}{2\pi\sigma_g^2} e^{-(x^2+y^2)/(2\sigma_g^2)}, \texttt{floor}\right), \tag{2}$$

where σ_g is set by the user. The cross sections σ_s and σ_t are constants defined by the user. The zero inflow boundary condition is used. Figure 1 shows the initial condition of $\langle f \rangle$ for $\sigma_g =$???. Follow this pattern for other initial conditions.

2.1.2 Delta Initial Condition

The limiting case, as $\sigma \to 0$, is the Delta initial condition. For this condition, the centermost cell is set to $\frac{1}{\Delta x \Delta y}$ where Δx and Δy are the cell dimensions. σ_S and σ_T are uniformly set to the configured value for sigma.

2.1.3 Lattice Initial Condition

The Lattice initial condition corresponds to a checker board pattern of highly scattering and highly absorbing regions. This configuration is reminiscent of a small section of a nuclear reactor core. The Lattice initial condition leaves the grid initially empty, but has the most complicated scattering pattern of the initial conditions. σ_S and σ_T are set to 1 at all positions, except for several blocks arranged throughout the grid at which $\sigma_S = 0$ and $\sigma_T = 10$. State the source too.

2.1.4 Smooth Initial Condition

The Smooth initial condition is primarily intended for testing convergence. This configuration initializes the grid points with a periodic boundary. Each point in

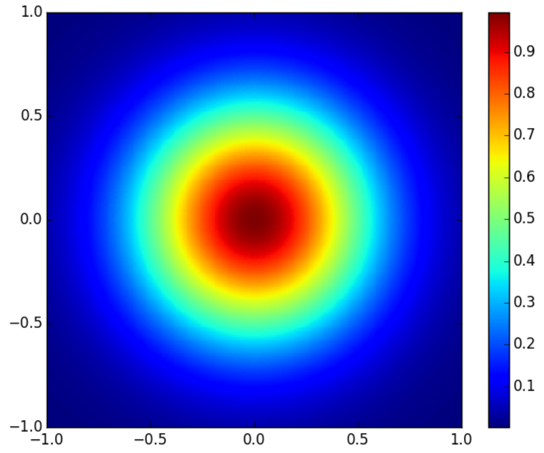

Figure 1: Gaussian Initial Condition

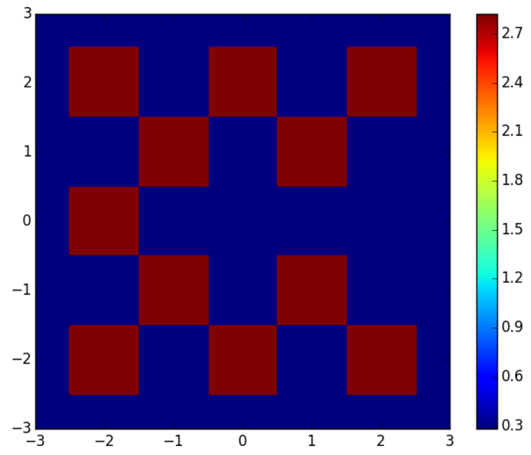

Figure 2: σ_T for Lattice Initial Condition

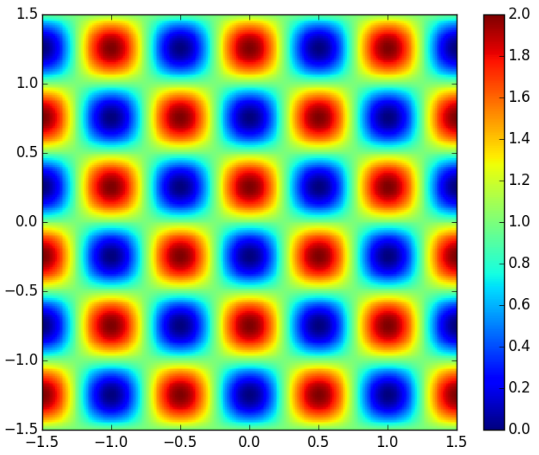

Figure 3: Smooth Initial Condition

the initial grid is set to

$$1 + \sin(2\pi x)\cos(2\pi y). \tag{3}$$

σ_S and σ_T are uniformly set to the configured value for sigma.

3 Angular Approximations

3.1 Discrete Ordinates

The `kinetic` solver is an implementation of the discrete ordinates method, also known as S_N. Our implementation uses a Chebyshev-Legendre quadrature on the unit sphere. A Lebedev quadrature is also implemented for this solver, but is considered experimental. Details of the quadratures are given in Section 4.1. Let $\{\Omega_1, \dots, \Omega_Q\} \in \mathbb{S}^2$ be be a set of nodes with corresponding weights $\{w_1, \dots, w_Q\}$. Then from Equation 1

$$\partial_t f_q + \Omega_q \cdot \nabla_x f_q + \sigma_t f_q = \frac{\sigma_s}{4\pi} \sum_{q'=1}^{Q} w_{q'} f_{q'}, \tag{4}$$

where $f_q(x,t) \approx f(x, \Omega_q, t)$ for $q = 1, \dots, Q$.

This implementation uses Heun's method to achieve second order convergence. Edge values are computed via upwinding. For approximate slopes, the double minmod limiter is used.

3.2 Moment Solvers

Show basic setup. Then define $\mathcal{E}(\mathbf{u})$ for each type. The `moment` solver uses standard spectral methods with Equation 1. The real spherical harmonics serve as an orthonormal basis of L^2 with respect to $\mathbb{S}^2$ for the expansion of the moments. Let $\mathbf{m}(\Omega) = (Y_{0,0}, Y_{1,-1}, Y_{1,0}, Y_{1,1}, \ldots, Y_{N,-N}, \ldots, Y_{N,N})^T$ be a vector of spherical harmonics up to and including degree N. The moments with respect to $\mathbf{m}$ are given by

$$\mathbf{u}^{\text{exact}}(\mathbf{x}, t) = \langle \mathbf{m}(\Omega) f(\mathbf{x}, \Omega, t) \rangle \tag{5}$$

since $u_{\ell,m} = f_{\ell,m}$ and the collision operator is diagonalized. Separate exact u from approximate u. The exact moment system is given by

$$\partial_t \mathbf{u} + \nabla_x \cdot \langle \Omega \mathbf{m} f \rangle = \mathrm{D}\mathbf{u}, \tag{6}$$

but this formulation is not closed. Thus f is replaced with a P_N moment closure $\mathcal{E}(\mathbf{u})$ such that $\langle \mathbf{m}\mathcal{E}(\mathbf{u}) \rangle = \mathbf{u}$. In this case,

$$\mathcal{E}(\mathbf{u}) = \sum_k u_k m_k = \mathbf{u}^T \mathbf{m} \tag{7}$$

yielding the closed moment system

$$\partial_t \mathbf{u} + \nabla_x \cdot \langle \Omega \mathbf{m} \mathcal{E}(\mathbf{u}) \rangle = \mathrm{D}\mathbf{u}. \tag{8}$$

It is only necessary to use the spherical harmonics $Y_{\ell,m}$ such that $\ell + m$ is even. State why...

3.2.1 Entropy Minimization

Suppose $\mathcal{E}(\mathbf{u})$ is obtained as an entropy minimization

$$\mathcal{E}(\mathbf{u}) = \arg\min_{g \in L^1} \langle \eta(g) \rangle \quad \text{subject to} \quad \langle \mathbf{m} g \rangle = \mathbf{u} \tag{9}$$

Fill in the rest here.

3.2.2 $\mathbf{P}_N$

$\mathcal{E}(\mathbf{u}) = \mathbf{m}^T \mathbf{u}$. Can get this by (1) truncating, (2) least squares, (3) minimizing entropy. Fill in a little bit.

Various filters can also be applied to suppress oscillations in spherical harmonics.

- The Hauck filter is described in [2]. Let N be the moment order and ω be the filter tune, and define

  $$\alpha = \frac{\omega}{N^2 (\sigma_T L + N)^2}. \tag{10}$$

  Scale each moment by

  $$\frac{1}{1 + \alpha n^2 (n+1)^2} \tag{11}$$

  for the n^{th} moment.

- The Spline filter is given in [3]. With moment order N and filter tune σ_e, let
$$s = \frac{-\sigma_e \Delta t}{\log F(N/(N+1))} \tag{12}$$
where $F(x) = 1/(1+x^4)$. The scale factor is given by
$$F(n/(N+1))^s. \tag{13}$$
- The Lanczos filter is taken from (insert citation). Again with moment order N and filter tune σ_e, let
$$s = \frac{-\sigma_e \Delta t}{\log L(N/(N+1))} \tag{14}$$
where
$$L(x) = \begin{cases} 1 & \text{when } x = 0 \\ \dfrac{\sin x}{x} & \text{otherwise.} \end{cases} \tag{15}$$
Now the moment scale factor is
$$L(n/(N+1))^s. \tag{16}$$

3.3 $\mathbf{M}_N$ and $\mathbf{PP}_N$

One serious drawback of the P_N method is the possibility of non-realizable solutions with negative densities, oscillatory approximations of nonsmooth solutions. Can just put the definitions for η and other stuff here for M_N and PP_N. Also, the link boxes are useful but ugly. Get rid of them.

Nonlinear approaches can ensure positivity, but come with increased complexity and computational cost. In general, entropy-based moment closures, like the `momopt` solver, can be cast in the framework of the following minimization problem
$$\mathcal{E}(\mathbf{u}) = \underset{g \in L^1}{\arg\min} \langle \eta(g) \rangle \quad \text{subject to} \quad \langle \mathbf{m} g \rangle = \mathbf{u} \tag{17}$$
where η is a smooth, strictly convex, coercive[1] function. For the `momopt` solver, $\eta(\mathbf{r}) = \mathbf{r} \log \mathbf{r} - \mathbf{r}$ and its Legendre dual[2] $\eta * (\mathbf{s}) = \mathbf{e}^{\mathbf{s}}$.

The solution to Equation 17, if it exists, is given by
$$\mathcal{E}(\mathbf{u}) = \eta_*'(\hat{\alpha}(\mathbf{u})^T \mathbf{m}) \tag{18}$$
where η_* is the Legendre dual of η and $\hat{\alpha}(\mathbf{u})$ solves the dual problem
$$\hat{\alpha}(\mathbf{u}) = \underset{\alpha \in \mathbb{R}^n}{\arg\min} \left\{ \langle \eta_*(\alpha^T \mathbf{m}) \rangle - \alpha^T \mathbf{u} \right\}. \tag{19}$$

[1] We define a function η to be coercive if $\lim_{r \to \infty} \frac{\eta(r)}{|r|} = \infty$

[2] The Legendre dual of η is given by $\eta_*(s) = rs - \eta(r)$ where $s = \eta'(r)$. By differentiating this relation, one can show that $r = \eta_*'(s)$. Thus η' and η_*' are inverses of each other.

4 Implementation

4.1 Quadratures

Let's go over this. Some of the wording is not quite right. The Chebyshev-Legendre quadrature [1] is used for numerical integration over $\mathbb{S}^2$ in all of the solvers. This quadrature is constructed from an n point Gauss-Legendre rule, which exactly integrates smooth functions of order less than $2n - 1$. Since this is a 2D code, it is only necessary to integrate over the upper half of $\mathbb{S}^2$, which is divided into n layers along the z axis. Abscissae and weights from GSL are arranged in a circle around $\mathbb{S}^2$ at each layer, giving n^2 points on the upper half of $\mathbb{S}^2$. The Chebyshev-Legendre quadrature is not optimal with respect to number of points, but is simple to implement. The Lebedev quadrature, for example, is optimal in number of points, and an experimental implementation is available for the `kinetic` solver. This quadrature, however, uses a more complicated arrangement of points, making exploiting symmetries in the integral more difficult. In addition, the points of the Chebyshev-Legendre quadrature are simply the Cartesian product of n angles spaced around a circle and n values of z. This structure could allow the evaluation of integrals via the Chebyshev-Legendre quadrature to be optimized more easily than the Lebedev quadrature, which is arranged nontrivially and becomes denser with increasing order. Write in implementation here. Not in appendix.

4.2 Time

Time steps are carried out via Heun's method which is an explicit two-stage Runge-Kutta method that is second order accurate. Write equation. Notice Heun's method is the average of two Euler steps which puts it into the category of a strong stability preserving (SSP) method. SSP methods preserve properties satisfied by the Euler method. In particular, the software uses this method to ensure positivity for some methods.

4.3 Space

All the implementation stuff should be here. Not in the appendix. Add proof of positivity for S_N and M_N and PP_N. Space is discretized using the finite volume method. The problem domain is decomposed into a regular grid of cells with an additional halo of ghost cells to enforce the boundary condition and allow synchronization with other nodes. At each update, the program first computes the flux at each cell, then updates the cell values according to the kinetic transport equation. Several of the solvers guarantee positivity on the grid.

For the `kinetic` solver (implementing S_N), the timestep Δt is chosen with respect to the cell size so as not to violate the CFL condition. To eliminate the possibility of negative density due to scattering, an additional term is included in the CFL check.

The `momopt` solver also ensures positivity as part of the flux calculations. The flux is based on an ansatz grid with strictly positive values. For M_N, the ansatz grid value a at each cell and direction is given by

$$a = \exp(\alpha^T m). \tag{20}$$

For PP_N,

$$a = \begin{cases} \frac{1}{2}k + \frac{1}{2}\sqrt{k^2 + 4\delta} & \text{if } k > 0 \\ \frac{-\delta}{\frac{1}{2}k - \frac{1}{2}\sqrt{k^2 + 4\delta}} & \text{if } k \leq 0 \end{cases} \tag{21}$$

where $k = \alpha^T m$.

4.4 Optimization Procedure

...

5 Program Layout

Make this nicer. Introduce each solver here (i.e. kinetic_solver) as associated with for instance S$_N$. Describe binary file formats. All options controlling the runtime operation of the program reside in `input.deck`. Comments prefixed with `#` are allowed, and options not used by the selected solver are ignored. Available options are listed in Section **??**, and example options for the various solvers are included in the `examples/` directory. The outputs are binary files with extensions `.sn`, `.pn`, and `.opt`, depending on the solver. Python code for reading and working with these files is provided in `util/formats.py`.

The common functionality is split up among the files in `src/`.

- `main.cpp` – Entry point of program
- `comm.cpp` – Controls MPI communication
- `utils.cpp` – Helper functions used throughout the code

The solver code has an approximately common layout, e.g. in `src/moment/`

- `moment_init.cpp` – Read config, set up quadrature and filters, etc.
- `moment_boundaries.cpp` – Communicate boundary data with other nodes
- `moment_update.cpp` – Solve flux, update the grid with time, etc.
- `moment_output.cpp` – Write out results

6 Results

...

A Gritty Details

Most of this should not be here, but rather in the math section.

A.1 `main.cpp`

The main entry point for the program, `main.cpp`, is responsible for reading the input files dictating runtime behavior, setting up MPI, and running the chosen solver. In addition, the grid initialization used by all solvers is defined here, so `main.cpp` controls the grid geometry and initial configuration. Program flow is as follows

1. Determine the current node and total number of nodes. If MPI support is not compiled in, fall back to default values corresponding to the primary and only node.
2. Read in the configuration stored in `input.deck`. This format is described in Section C.1.
3. Configure the solver. Solvers are described in Section B.
4. Set the domain of the current node.
5. Initialize the grid.
6. Run the solver over the specified time interval.
7. Output data.

The default grid initialization, `Solver::initializeGrid`, also resides in `main.cpp`. At present, all solvers use this method. The grid is represented as a two dimensional array of floating point values. The grids used in the program are larger than configured in `input.deck`; a border of of a specified width of ghost cells is added outside the grid described in the configuration. The center points of the cells are used for calculation of initial conditions. Figure 4 illustrates this layout. There are two other arrays of identical size storing σ_S and σ_T values. To initialize the grid, first set the the border cells to the specified floor value, `c_floor`, and then calculate the initial values for the main cells.

A.1.1 Delta Initial Condition

Probably need this text somewhere. When `INITCOND_LINESOURCE` is specified with Gaussian $\sigma = 0$, the Delta Initial Condition applies. In this case, σ_S and σ_T for each cell are set to σ. The cells of the initial grid are set to `c_floor`, with the exception of the centermost cell, which is set to $1/\Delta x \Delta y$ where Δx and Δy denote the dimensions of each cell.

A.1.2 Gaussian Initial Condition

This condition is used when `INITCOND_LINESOURCE` is selected as the the initial condition and Gaussian $\sigma \neq 0$. Each cell's initial value is set to $\max\{$`gaussianFactor`, `c_floor`$\}$ where

$$\texttt{gaussianFactor} := \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x_i^2 + y_j^2}{2\sigma^2}\right). \tag{22}$$

σ_S and σ_T for each cell is set to σ.

A.1.3 Lattice Initial Condition

When `INITCOND_LATTICE` is specified, each cell's initial value is set to `c_floor`, and $\sigma_S, \sigma_T := 1$. For any cell (x_i, y_j) with

$$\|(x_i, y_j) - (s_x, s_y)\|_\infty < 0.5 \tag{23}$$

for some $(s_x, s_y) \in \{$ (2.0,2.0), (2.0,0.0), (2.0,-2.0), (1.0,1.0), (1.0,-1.0), (0.0,-2.0), (-1.0,1.0), (-1.0,-1.0), (-2.0,2.0), (-2.0,0.0), (-2.0,-2.0) $\}$, σ_T is set to 10 and σ_S is set to 0.

A.1.4 Smooth Periodic Condition

When `INITCOND_PERIODIC` is selected, σ_S, σ_T are initialized to σ, and the initial value at each point (x_I, y_j) is set to

$$\sin(2\pi x_i)\cos(2\pi y_j) + 1. \tag{24}$$

A.2 `comm.cpp`

Maybe put how MPI works in Program Layout. Also put OpenMP stuff in there too so the user knows what is sped up and how. When using MPI, `comm.cpp` communicates boundary data between nodes. `Solver::getInnerBoundaries` is first called to obtain grid data on the boundaries of the current node's region of the domain. Next, each node trades boundary data with its neighbors to the north, south, east, then west via `MPI_ISend` and `MPI_IRecv`. Finally, the node calls `Solver::setOuterBoundaries` to update its grid with the data from neighboring nodes. Note scaling has not been tested for MPI.

A.3 `utils.cpp`

`utils.cpp` is mostly comprised of IO-related utility functions. In addition, there are functions to compute the 1-norm of an arbitrary vector and to compute Gaussian weights and nodes using GSL.

A.4 `timer.cpp`

`timer.cpp` contains a timer class used in `main.cpp` to record the time taken by computations.

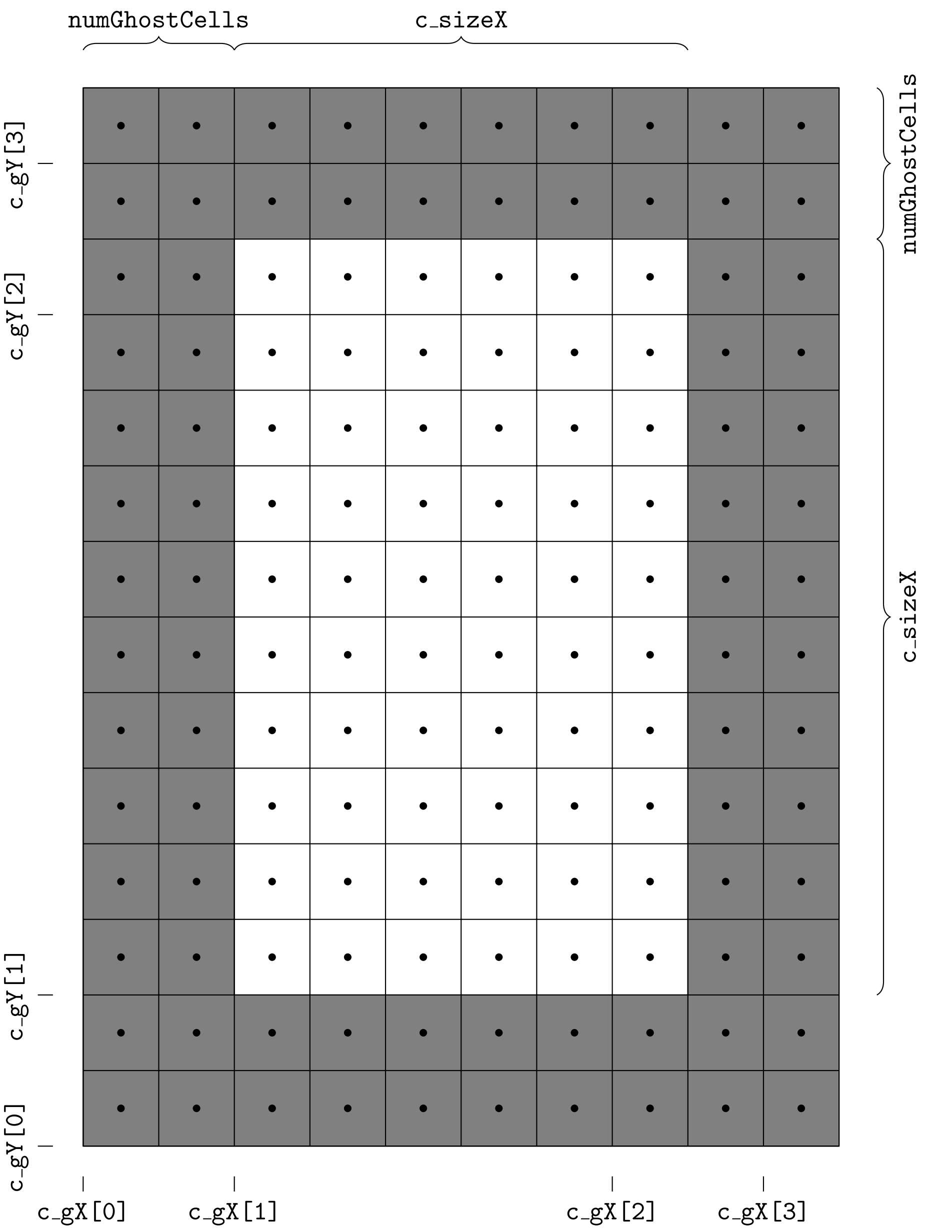

Figure 4: Grid layout

B Solvers

All solvers must store internal parameters and grid data, and implement methods to initialize, query grid parameters, update, and output data. See `solver.h` for more detail. Each solver is organized into a directory containing the necessary functionality split across several files. The solvers currently implemented are `kinetic`, `moment`, `momopt`, and `dn`.

B.1 `kinetic`

B.1.1 `kinetic_init.cpp`

`kinetic` reads its runtime configuration from the file `kinetic.deck` (see Section C.2). The maximum Δt value `maxDt` is calculated as

$$\texttt{maxDt} := \frac{1}{2}(\texttt{cflFactor})\frac{\Delta x \Delta y}{\Delta x + \Delta y}. \tag{25}$$

The grid is initialized as described in Section A.1. In addition to the grid layers common to all solvers, `kinetic` uses two others, `c_kinetic` and `c_flux`. `utils.cpp` is used to get the quadrature as described in Section A.3. These fixed-order Gauss-Legendre integration points and weights returned will be referred to as μ_i, and w_i, respectively. Next, the azimuthal angles of quadrature, ϕ_k, are calculated as

$$\phi_k := \frac{(k+0.5)\pi}{\texttt{quadOrder}} \tag{26}$$

for $k \in \mathbb{Z}$ such that $0 \le k < 2(\texttt{quadOrder})$, placing $2(\texttt{quadOrder})$ points evenly around the unit circle. The quadrature weights are ultimately stored in `c_quadWeights`, and the points are mapped into cylindrical coordinates and stored in `c_xi` and `c_eta`.

For q_1, q_2 such that $0 \le q_1 < (\texttt{quadOrder})/2$ and $0 \le q_2 < 2(\texttt{quadOrder})$, define a quadrature counter $q := 2(\texttt{quadOrder})q_1 + q_2$. Now

$$\texttt{c\_quadWeights}[\texttt{q}] := \frac{2\pi w_{q_1}}{\texttt{quadOrder}} \tag{27}$$

$$\texttt{c\_xi}[\texttt{q}] := \sqrt{1-\mu_{q_1}^2}\cos\phi_{q_2} \tag{28}$$

$$\texttt{c\_eta}[\texttt{q}] := \sqrt{1-\mu_{q_1}^2}\sin\phi_{q_2} \tag{29}$$

B.1.2 `kinetic_update.cpp`

The core of the `kinetic` solver is the `update` method defined here. Each call to update one time step operates as follows:

1. Make a copy of the `c_kinetic` grid.

2. Carry out the first Euler step.

3. Carry out the second Euler step.

4. Average the initial grid with the results of the second iteration.

Some preliminary functions will be used in the discussion of the above steps.

The minmod preserves positivity. Need to show this. The `minmod(double x, double y)` function is designed to return

- 0 if $xy < 1$
- $\min\{x, y\}$ if $x, y > 0$
- $-\min\{|x|, |y|\}$ if $x, y < 0$

and implemented as follows.

$$\operatorname{minmod}(x, y) = \operatorname{sgn}'(x) \max\Big(0, \min\big(|x|, y \operatorname{sgn}'(x)\big)\Big) \tag{30}$$

where

$$\operatorname{sgn}'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{31}$$

Additionally,

$$\operatorname{slopefit}(\ell, c, r, \theta) = \operatorname{minmod}\left((r-c)\theta, \operatorname{minmod}\left(\frac{1}{2}(r-\ell), (c-\ell)\theta\right)\right). \tag{32}$$

To carry out each Euler step, the `kinetic` solver first communicates the current cell's boundaries with neighboring MPI cells (described in Section B.1.3) and then solves flux for the cell (described below). For each cell (i, j) in the main grid (excluding the ghost cells), the integral of F is calculated as

$$\texttt{integral} := \frac{\sigma_{S,(i,j)}}{4\pi} \sum_{q=0}^{n} w_q f_q(i, j) \tag{33}$$

where w_q is the calculated quadrature weight for the q^{th} point of an n point Gauss-Legendre quadrature and $f_q(i, j)$ is the value of the `c_kinetic` grid at (i, j, q).

For each $0 \leq q < n$, subtract from the value of `c_kinetic` at (i, j, q)

$$\Delta t \Big(\sigma_{T,(i,j)} (\texttt{c\_kinetic})_{(i,j,q)} - \texttt{integral}\Big). \tag{34}$$

Next, if using `INITCOND_LATTICE`, with the cell's bottom left point (x_i, y_j) add an additional Δt to `c_kinetic` at (i, j, q) if $\|(x_i, y_j)\|_\infty < 0.5$. Finally, subtract $\Delta t (\texttt{c\_flux})_{i,j,q}$ from `c_kinetic` at (i, j, q).

Communicating boundaries, solving flux, and evaluating an Euler step are carried out again as above. Now for each (i, j) in the main grid, $0 \leq q < n$, average `c_kinetic` at (i, j, q) with the corresponding cell from the copy made at the beginning of the procedure.

Solving for flux involves approximating the flux in each of four directions (north, south, east, west) using `slopefit` applied to the current cell and its neighbors. The overall flux for each cell is then calculated as

$$\frac{\xi_q}{\Delta x}(\texttt{eastFlux} - \texttt{westFlux}) + \frac{\eta_q}{\Delta y}(\texttt{northFlux} - \texttt{southFlux}). \tag{35}$$

B.1.3 `kinetic_boundaries.cpp`

The `kinetic` solver uses the code in `kinetic_boundaries.cpp` in conjunction with `Solver::communicateBoundaries()` (see Section A.2) to coordinate the boundaries of different nodes when using MPI. The methods defined here take pointers to buffers from which and into which to copy grid data. These methods copy the north, south, east, and west parts of the gray area as shown in Figure 4.

B.1.4 `kinetic_output.cpp`

The `kinetic` solver exports data to files named `out_%.3f_%d.sn` formatted with the time under simulation and the node index. This file consists of a short header recording the dimensions of the grid, the domain bounds, and the number of quadrature points and weights. The rest of the file contains the floating point values of `c_kinetic` over the main grid (*not* including the ghost cells).

C Special Files UPDATE ME

C.1 `input.deck`

`input.deck` is the primary input file which controls the basic operation of the program. `input.deck` is a line-based text file storing a set of runtime configuration options as an ordered listing of key-value pairs. An example file is included with the source code. The required options are discussed in Table 1.

C.2 `kinetic.deck`

`kinetic.deck` is a brief runtime configuration file controlling the behavior of the `kinetic` solver. This file uses the same format as `input.deck` (Section C.1). The required options are given in Table 2.

Table 1: Parameters for `input.deck`

Option	Type	Description
`SOLVER`	`char[]`	Solver to be used. Allowed values are `kinetic`, `moment`, `momopt`, and `dn`. The operation of each solver is discussed in Section B.
`NUM_CELLS_X`	`int`	Number of grid cells in the x direction
`NUM_CELLS_Y`	`int`	Number of grid cells in the y direction
`NUM_MPI_PARTITIONS_X`	`int`	Number of MPI partitions in the x direction
`NUM_MPI_PARTITIONS_Y`	`int`	Number of MPI partitions in the y direction
`A_X`	`double`	x coordinate of the bottom left corner of the grid
`B_X`	`double`	x coordinate of the top right corner of the grid
`A_Y`	`double`	y coordinate of the bottom left corner of the grid
`B_Y`	`double`	y coordinate of the top right corner of the grid
`T_FINAL`	`double`	Duration of the simulation
`OUT_DELTA_T`	`double`	Temporal resolution of the output files
`GAUSSIAN_SIGMA`	`double`	σ used in the initial grid configurations
`FLOOR`	`double`	Minimum value that occurs in the grid
`INIT_COND`	`int`	How the initial grid values are calculated. Allowed values are 0, 1, and 2, corresponding to `INITCOND_LINESOURCE`, `INITCOND_LATTICE`, and `INITCOND_PERIODIC`, respectively. Grid initialization is described in Section A.1.
`SIGMA`	`double`	Default scattering used in the σ_S and σ_T grids. The scattering values for each grid position are determined based on the particular initial condition selected.

Table 2: Parameters for `kinetic.deck`

Option	Type	Description
`QUAD_ORDER`	`int`	Order of the Gaussian quadrature used for integration
`CFL_FACTOR`	`double`	Parameter used to choose the maximum Δt without violating the CFL Condition.

References

[1] Kendall Atkinson. Numerical integration on the sphere. *J. Austral. Math. Soc. Ser. B*, 23:332–347, 1982.

[2] Ryan G. McClarren and Cory D. Hauck. Robust and accurate filtered spherical harmonics expansions for radiative transfer. *Journal of Computational Physics*, 229(16):5597 – 5614, 2010.

[3] David Radice, Ernazar Abdikamalov, Luciano Rezzolla, and Christian D. Ott. A new spherical harmonics scheme for multi-dimensional radiation transport I: static matter configurations. *Journal of Computational Physics*, 242(0):648 – 669, 2013.