Design Document

C. Kristopher Garrett, Tim Shaffer

July 21, 2015

1 Problems Being Solved

This software simulates particles with unit speed that scatter isotropically from an infinite background medium according to a simple kinetic equation. External sources are not included, but could be added in the future. 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 $x \in \mathbb{R}^3$ is position, $\Omega \in \mathbb{S}^2$ (the unit sphere) is the direction of travel, $\sigma = \sigma(x)$ is the scattering cross section, and $\langle \cdot \rangle$ is shorthand for integration over $\mathbb{S}^2$.
The `kinetic` solver directly computes solutions to Equation 1, while the `moment` solver calculates solutions based on the spherical harmonic basis functions. The `momopt` solver includes additional optimizations and experimental features. This program explicitly computes approximate solutions to Equation 1 as the system evolves from one of several initial conditions, defined in Section 2.

2 Initial Conditions

At the beginning of simulation, the grid is initialized according to one of several configurations. In the Gaussian initial condition, a two dimensional Gaussian function is placed with its peak at the center of the grid. The σ value is provided as a configuration option to the program. The limiting case, when $\sigma = 0$, 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. 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. The Smooth initial condition is primarily intended for testing convergence. This configuration initializes the grid points with a periodic

boundary given by

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

3 `kinetic` Solver

The `kinetic` solver is an implementation of the discrete ordinates method, also known as S_N. This method uses Gauss-Legendre quadrature sets on the unit sphere. A Lebedev quadrature is also implemented, but is considered experimental. Let $\{\Omega_1, \ldots, \Omega_Q\} \in \mathbb{S}^2$ be be a set of nodes with corresponding weights $\{w_1, \ldots, w_Q\}$. Then from Equation 1

$$\partial_t f_q + \Omega_q \cdot \nabla_x f_q + \sigma_t f_q = \frac{\sigma_s}{4\pi} \sum_{q'=1}^{Q} w_{q'} f_{q'}, \tag{3}$$

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

4 `moment` Solver

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 the moments with respect to $\mathbf{m}$ are given by

$$\mathbf{u}(\mathbf{x}, t) = \langle \mathbf{m}(\Omega) f(\mathbf{x}, \Omega, t) \rangle \tag{4}$$

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

$$\partial_t \mathbf{u} + \nabla_x \cdot \langle \Omega \mathbf{m} f \rangle = \mathrm{D}\mathbf{u}, \tag{5}$$

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,

$$\mathcal{E}(\mathbf{u}) = \sum_k u_k m_k = \mathbf{u}^T \mathbf{m} \tag{6}$$

yielding the closed moment system

$$\partial_t \mathbf{u} + \nabla_x \cdot \langle \Omega \mathbf{m} \mathcal{E}(\mathbf{u}) \rangle = \mathrm{D}\mathbf{u}. \tag{7}$$

It is only necessary to use the spherical harmonics $Y_{\ell,m}$ such that $\ell + m$ is even. Various filters can also be applied to suppress oscillations in spherical harmonics. These filters use a scale factor on each cell and timestep to compute a perturbed differential equation with minimal added computation.

4.1 Hauck Filter

The Hauck filter is described in (insert citation). Let N be the moment order and ω be the filter tune, and define

$$\alpha = \frac{\omega}{N^2(\sigma_T L + N)^2}. \tag{8}$$

Scale each moment by

$$\frac{1}{1 + \alpha n^2 (n+1)^2} \tag{9}$$

for the n^{th} moment.

4.2 Sspline Filter

The Spline filter is given in (insert citation). With moment order N and filter tune σ_e, let

$$s = \frac{-\sigma_e \Delta t}{\log F(N/(N+1))} \tag{10}$$

where $F(x) = 1/(1+x^4)$. The scale factor is given by

$$F(n/(N+1))^s. \tag{11}$$

4.3 Lanczos Filter

The Lanczos filter is taken from (insert citation). Again with moment order N and filter tune σ_e, let

$$s = \frac{-\sigma_e \Delta t}{\log L(N/(N+1))} \tag{12}$$

where

$$L(x) = \begin{cases} 1 & \text{when } x = 0 \\ \dfrac{\sin x}{x} & \text{otherwise.} \end{cases} \tag{13}$$

Now the moment scale factor is

$$L(n/(N+1))^s. \tag{14}$$

5 `momopt` Solver

One serious drawback of the `moment` solver is the possibility of non-realizable solutions with negative densities, oscillatory approximations of nonsmooth solutions. 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

$$\mathcal{E}(\mathbf{u}) = \underset{g \in L^1}{\arg\min} \langle \eta(g) \rangle \quad \text{subject to} \quad \langle \mathbf{m} g \rangle = \mathbf{u} \tag{15}$$

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 15, if it exists, is given by

$$\mathcal{E}(\mathbf{u}) = \eta'_*(\hat{\alpha}(\mathbf{u})^T \mathbf{m}) \tag{16}$$

where η_* is the Legendre dual of η and $\hat{\alpha}(\mathbf{u})$ solves the dual problem

$$\hat{\alpha}(\mathbf{u}) = \underset{\alpha \in \mathbb{R}^n}{\arg\min} \left\{ \langle \eta_*(\alpha^T \mathbf{m}) \rangle - \alpha^T \mathbf{u} \right\} \tag{17}$$

6 Installation

This software is built using SCons. If SCons is not installed, a copy has been included in the `build/` directory. If Python 2.x is installed but not SCons, replace `scons` with `python build/scons.py`.
To build this software using default options, simply run

```
scons
```

in the project directory. SCons will use default compilation options to build `solver_serial`. If acceleration is desired, it should be provided as arguments to SCons, e.g.

```
scons --omp
```

builds `solver_omp`. Supported options are `--omp`, `--mpi`, and `--cuda`. If multiple acceleration options are provided, an optimized executable will be build for each. In addition, it is possible to build executables with multiple types of acceleration. Simply give the names of the desired optimizations in the order MPI, OMP, CUDA, e.g.

```
scons --omp --mpi solver_mpiomp
```

To successfully build and run, this software requires

- GSL
- BLAS
- LAPACK

[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.

- OpenMP (optional, provided by many compilers)
- Open MPI (optional)
- CUDA development files (optional)

If these libraries and header files are not in the system's default locations, additional search paths can be provided in `config.py`. This file also includes options for controlling the compiler that SCons uses.
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

7 Implementation Details

7.1 `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 9.1.

3. Configure the solver. Solvers are described in Section 8.
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 1 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.

7.1.1 Delta Initial Condition

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.

7.1.2 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 $\max\{\texttt{gaussianFactor}, \texttt{c\_floor}\}$ where

$$\texttt{gaussianFactor} := \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x_i^2 + y_j^2}{2\sigma^2}\right). \tag{18}$$

σ_S and σ_T for each cell is set to σ.

7.1.3 Lattice Initial Condition

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

$$\|(x_i, y_j) - (s_x, s_y)\|_\infty < 0.5 \tag{19}$$

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.

7.1.4 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

$$\sin(2\pi x_i)\cos(2\pi y_j) + 1. \tag{20}$$

7.2 `comm.cpp`

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_Sendrecv`. Finally, the node calls `Solver::setOuterBoundaries` to update its grid with the data from neighboring nodes.

7.3 `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.

7.4 `timer.cpp`

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

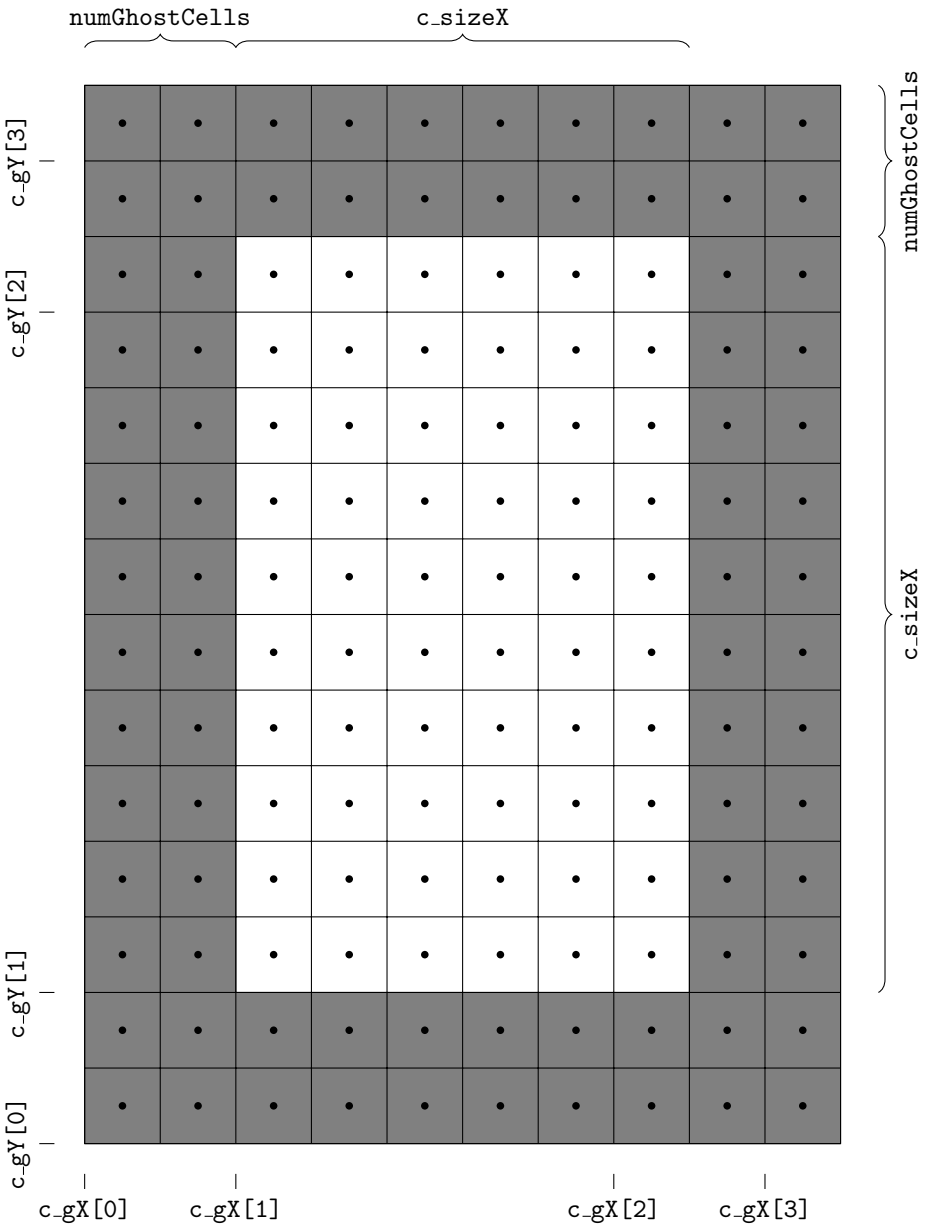

Figure 1: Grid layout

8 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`.

8.1 `kinetic`

8.1.1 `kinetic_init.cpp`

`kinetic` reads its runtime configuration from the file `kinetic.deck` (see Section 9.2). The maximum Δt value `maxDt` is calculated as

$$\texttt{maxDt} := \frac{1}{2}(\texttt{cflFactor})\frac{\Delta x \Delta y}{\Delta x + \Delta y}. \tag{21}$$

The grid is initialized as described in Section 7.1. 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 7.3. 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

$$\phi_k := \frac{(k+0.5)\pi}{\texttt{quadOrder}} \tag{22}$$

for $k \in \mathbb{Z}$ such that $0 \leq k < 2(\texttt{quadOrder})$, 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 $0 \leq q_1 < (\texttt{quadOrder})/2$ and $0 \leq q_2 < 2(\texttt{quadOrder})$, define a quadrature counter $q := 2(\texttt{quadOrder})q_1 + q_2$. Now

$$\texttt{c\_quadWeights[q]} := \frac{2\pi w_{q_1}}{\texttt{quadOrder}} \tag{23}$$

$$\texttt{c\_xi[q]} := \sqrt{1-\mu_{q_1}^2}\cos\phi_{q_2} \tag{24}$$

$$\texttt{c\_eta[q]} := \sqrt{1-\mu_{q_1}^2}\sin\phi_{q_2} \tag{25}$$

8.1.2 `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(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.

$$\operatorname{minmod}(x, y) = \operatorname{sgn}'(x) \max\left(0, \min\left(|x|, y \operatorname{sgn}'(x)\right)\right) \tag{26}$$

where

$$\operatorname{sgn}'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{27}$$

Additionally,

$$\operatorname{slopefit}(\ell, c, r, \theta) = \operatorname{minmod}\left((r-c)\theta, \operatorname{minmod}\left(\frac{1}{2}(r-\ell), (c-\ell)\theta\right)\right). \tag{28}$$

To carry out each Euler step, the `kinetic` solver first communicates the current cell's boundaries with neighboring MPI cells (described in Section 8.1.3) 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

$$\texttt{integral} := \frac{\sigma_{S,(i,j)}}{4\pi} \sum_{q=0}^{n} w_q f_q(i, j) \tag{29}$$

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)

$$\Delta t \left(\sigma_{T,(i,j)}(\texttt{c\_kinetic})_{(i,j,q)} - \texttt{integral}\right). \tag{30}$$

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

$$\frac{\xi_q}{\Delta x}(\texttt{eastFlux} - \texttt{westFlux}) + \frac{\eta_q}{\Delta y}(\texttt{northFlux} - \texttt{southFlux}). \tag{31}$$

8.1.3 `kinetic_boundaries.cpp`

The `kinetic` solver uses the code in `kinetic_boundaries.cpp` in conjunction with `Solver::communicateBoundaries()` (see Section 7.2) 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 1.

8.1.4 `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).

9 Special Files

9.1 `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.

9.2 `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 9.1). 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 8.
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 7.1.
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.