Particle Based Numerical Hydrodynamics of Ideal Gases

The Used and Implemented Equations

Mladen Ivkovic (mladen.ivkovic@hotmail.com)

Contents

1 General Implementation Details

TODO

needs updating

This section gives an overview on how this code works. To read about how to use the code, see the `/README.md` file.

1.1 How the program works

- All the source files are in the `/program/src/` directory.
- The main program files are `/program/src/main.c` if you want to use the code as a hydro/hyperbolic conservation law solver, or `/program/src/main-riemann.c` if you want to use the code as a Riemann solver.
- The program starts off by reading in the initial conditions (IC) file and the parameter file, which are both required command line arguments when executing the program. All functions related to reading and writing files are in `/program/src/io.c`
- Then a few checks are done to make sure no contradictory or missing parameters are given, and the grid on which the code will work on is initialized, as well as some global variables like the step number and current in-code time.
- There are two global variables used throughout the code:
  - `struct params param`: A parameter struct (defined in `/program/src/params.h`) that stores global parameters that are useful to have everywhere throughout the code (even if it isn't optimal coding practice...) All parameter related functions are in `/program/src/params.c`.
  - `struct cell* grid` or `struct cell** grid`, depending on whether you run the code in 1D or 2D, respectively. It's an array of `struct cell`, defined in `/program/src/cell.h`, which is meant to store all cell related quantities: primitive states, `prim`, as `struct pstate`, and conserved states, `cons`, as `struct cstate`. Furthermore they have `struct pstate pflux` and `struct cstate cflux` to store fluxes of primitive and conserved variables. All cell/grid related functions are written in `/program/src/cell.c`.

- If the code is used as a hydro/conservation law solver:

  - The main loop starts now:

  - Advance the solver for a time step.

  - Write output data if you need to (when to dump output is specified in the parameter file). All functions related to reading and writing files are in `/program/src/io.c`

  - Write a message to the screen after the time step is done. This includes the current step number, in-code time, the ratio of initial mass to the current mass on the entire grid, and the wall-clock time that the step needed to finish.

- If the code is used as a Riemann solver:

  - Only do a loop over all cells and solve the Riemann problem given by the IC at the cell's position at given time t. Some functions for the Riemann problem that are used by all implemented Riemann solvers are given in `/program/src/riemann.h` and `/program/src/riemann.c`. When a specific Riemann solver is set, `/program/src/riemann.h` includes that file from the directory `/program/src/riemann/`. The file names in that directory should be obvious.

  - Write output data. All functions related to reading and writing files are in `/program/src/io.c`

1.2 Contents of specific files and directories in `/program/src/`

- `/program/src/cell.c`, `/program/src/cell.h`:

  All cell/grid related functions. The grid is used as `struct cell* grid` or `struct cell** grid`, depending on whether you run the code in 1D or 2D, respectively.

  It's an array of `struct cell`, defined in `/program/src/cell.h`, which is meant to store all cell related quantities: primitive states, `prim`, as `struct pstate`, and conserved states, `cons`, as `struct cstate`. Furthermore they have `struct pstate pflux` and `struct cstate cflux` to store fluxes of primitive and conserved variables. All cell/grid related functions are written in `/program/src/cell.c`.

- `/program/src/defines.h`:

Contains all macro definitions, like iteration tolerance, the box size, the adiabatic coefficient γ etc, as well as some physical constants (mostly related to γ).

- `/program/src/io.h`, `/program/src/io.c`:

  All input/output related functions, i.e. anything related to reading and writing files.

- `/program/src/limiter.h`, `/program/src/limiter.c`:

  Slope and flux limiter related functions (section **??**) that are used regardless of the choice of the limiter. For a specific choice of slope limiter, `/program/src/limiter.h` includes a specific file from `/program/src/limiter/`. The file name in `/program/src/limiter/` should be obvious.

- `/program/src/limiter/`:

  Slope limiter functions (section **??**) for specific limiters. They will be included by `/program/src/limiter.h` during compile time by setting the corresponding variable name in the `/program/bin/Makefile`.

  Essentially, these files only contain the actual computation of $\phi(r)$ and $\xi(r)$.

- `/program/src/main.c`:

  The main function of the program when the program is utilized as a hydro/hyperbolic conservation law solver.

- `/program/src/main-riemann.c`:

  The main function of the program when the program is utilized as a Riemann solver.

- `/program/src/riemann.h`, `/program/src/riemann.c`:

  Riemann solver related functions (section **??**) that are used regardless of the choice of the Riemann solver. For a specific choice of Riemann solver, `/program/src/riemann.h` includes a specific file from `/program/src/riemann/`. The file name in `program/src/riemann/` should be obvious.

- `/program/src/riemann/`:

  Riemann solver functions (section **??**) for specific Riemann solvers. They will be included by `/program/src/riemann.h` during compile time by setting the corresponding variable name in the `/program/bin/Makefile`.

Essentially, these files contain only the specific function to get the star state pressure and contact wave velocity. The only exception is the HLLC solver, which works a bit differently than the other implemented solvers. There, essentially everything needs to be done in a special way, so the solver contains its own routines, with a "`HLLC`" added to the function names.

- `/program/src/solver.h`, `/program/src/solver.c`:

  Hydro and advection solver related functions (section **??**, 4) that are used regardless of the choice of the hydro solver. For a specific choice of solver, `/program/src/solver.h` includes a specific file from `/program/src/solver/`. The file name in `/program/src/solver/` should be obvious. For implementation details of each solver, look up the implementation details in their respective section **??**, 4.

- `/program/src/solver/`:

  Hydro and advection solver functions (section **??**, 4) for specific solvers. They will be included by `/program/src/solver.h` during compile time by setting the corresponding variable name in the `/program/bin/Makefile`. For implementation details of each solver, look up the implementation details in their respective section **??**, 4.

- `/program/src/utils.h`, `/program/src/utils.c`:

  Miscellaneous small utilities that are irrelevant for the actual hydro or hyperbolic conservation law solving, like printing a banner every time the code starts, standardized functions to print outputs to screen or throw errors, etc.

1.3 Compilation Flags

1.3.1 Intended to be set by the user

variable	options	description
`EXEC`	`some_name`	resulting executable name. Can be set as you wish.

`NDIM`	`1`
	`2`
		Number of dimensions

`SOLVER`	*Which conservation law solver to use. Two classes are implemented: To solve advection (section ??) and Euler equations (section 4).*	
	`ADVECTION_PWCONST`	Piecewise constant advection (section ??)
	`ADVECTION_PWLIN`	Piecewise linear advection (section ??)
	`ADVECTION_WAF`	Weighted Average Flux advection (section ??)
	`GODUNOV`	Godunov Upwind first order hydrodynamics (section ??)
	`WAF`	Weighted Average Flux hydrodynamics (section ??)
	`MUSCL`	MUSCL-Hancock hydrodynamics (section ??)
`RIEMANN`	*Riemann solvers (section ??) are used in combination with hydro solvers for the Euler equations. Advection solvers don't need them.*	
	`EXACT`	Exact Riemann solver (section ??)
	`HLLC`	HLLC (section ??)
	`TRRS`	Two Rarefaction Riemann solver (section ??)
	`TSRS`	Two Shock Riemann solver (section ??)
`LIMITER`	*Flux/slope limiters (section ??) are needed for all methods higher than first order to avoid unphysical oscillations that arise from the numerical schemes.*	
	`MC`	Monotoniced Center Difference limiter (section ??)
	`MINMOD`	Minmod limiter (section ??)
	`SUPERBEE`	Superbee limiter (section ??)
	`VANLEER`	Van Leer limiter (section ??)
`SOURCES`	*Source Terms (section 9) from external forces. Can be left undefined.*	
	`NONE`	No source terms
	`CONSTANT`	Constant source terms (section 9.1)
	`RADIAL`	Radial constant source terms (section 9.1)
`INTEGRATOR`	*Which integrator to use to integrate the source terms (section 9) from external forces in time. Can (and should) be left undefined if sources are `NONE` or undefined.*	
	`NONE`	No source terms
	`RK2`	Runge-Kutta 2 integrator (section 9.2)
	`RK4`	Runge-Kutta 4 integrator (section 9.2)

1.3.2 Behind the Scenes

It's simpler to pass on an integer as a definition `-Dsomedefinition` than any kind of string, in particular if you want to do comparisons like `#if SOLVER == GODUNOV` inside the code. Hence all choices are being translated to integers behind the scenes. Here is the list of translations:

variable	option	integer
`SOLVER`	`ADVECTION_PWCONST`	11
	`ADVECTION_PWLIN`	12
	`ADVECTION_WAF`	13
	`GODUNOV`	21
	`WAF`	22
	`MUSCL`	23
`RIEMANN`	`NONE`	0
	`EXACT`	1
	`HLLC`	2
	`TRRS`	3
	`TSRS`	4
`LIMITER`	`NONE`	0
	`MINMOD`	1
	`SUPERBEE`	2
	`VANLEER`	3
	`MC`	4
`SOURCES`	`NONE`	0
	`CONSTANT`	1
	`RADIAL`	2
`INTEGRATOR`	`NONE`	-
	`RK2`	-
	`RK4`	-

The `INTEGRATOR` definition requires no checks, it only determines which files will be included and compiled, so no integers are assigned.

Some other definitions for checks are required. These definitions are automatically identified and passed to the compiler, so no additional action from the user's side is required.

- If any advection solver is used, the code needs an additional `-DADVECTION` flag
- If any source terms are used, the code needs an additional `-DWITH_SOURCES` flag

All these things are handled in the `/program/bin/processing.mk` file, as well as checks whether all required variables are defined, and if not, default values are set.

The default values are:

variable	**default value**
`SOLVER`	`GODUNOV`
`RIEMANN`	`EXACT`
`LIMITER`	`NONE`
`SOURCES`	`NONE`

1.4 Tests

Check whether the code works as intended

A primitive test suite is set up in `/program/test` and can be run by executing `/program/test/run.sh`. It will compile pretty much every possible combination of solvers, Riemann solvers, limiters, dimensions etc. and run some pre-defined tests. When finished correctly, a LaTeX pdf `/proram/test/test_results.pdf` will be generated, with the generated results being shown right below expected results.

It's not the most exact way of testing the code, but if you break something major, you will notice.

Code Coverage

A coverage test is set up in `/program/coverage`. It uses `gcovr` and compiles and runs pretty much all possible versions of the code. Finally, it combines the coverage data from all runs and generates a html file `/program/coverage/coverage.html` that shows the code coverage.

2 Ideal Gases

2.1 Governing Equations

We are mostly going to concern ourselves with ideal gases, which are described by the Euler equations:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho \\ \rho\mathbf{v} \\ E\end{pmatrix} + \nabla \cdot \begin{pmatrix}\rho\mathbf{v} \\ \rho\mathbf{v}\otimes\mathbf{v} + p \\ (E+p)\mathbf{v}\end{pmatrix} = \begin{pmatrix}0 \\ \rho\mathbf{a} \\ \rho\mathbf{a}\mathbf{v}\end{pmatrix} \tag{1}$$

Where

- ρ: fluid density
- $\mathbf{v}$: fluid (mean/bulk) velocity at a given point. I use the notation $\mathbf{v} = (u, v, w)$, or when indices are useful, $\mathbf{v} = (v_1, v_2, v_3)$
- p: pressure
- E: specific energy. $E = \frac{1}{2}\rho\mathbf{v}^2 + \rho\varepsilon$, with $\varepsilon =$ specific internal thermal energy
- $\mathbf{a}$: acceleration due to some external force.

The outer product $\cdot \otimes \cdot$ gives the following tensor:

$$(\mathbf{v}\otimes\mathbf{v})_{ij} = \mathbf{v}_i\mathbf{v}_j \tag{2}$$

Furthermore, we have the following relations for ideal gasses:

$$p = nkT \tag{3}$$

$$p = C\rho^\gamma \qquad \text{entropy relation for smooth flow, i.e. no shocks} \tag{4}$$

$$s = c_V \ln\left(\frac{p}{\rho^\gamma}\right) + s_0 \qquad \text{entropy} \tag{5}$$

$$a = \sqrt{\left.\frac{\partial p}{\partial \rho}\right|_s} = \sqrt{\frac{\gamma p}{\rho}} \qquad \text{sound speed} \tag{6}$$

with

- n : number density
- k : Boltzmann constant
- T : temperature
- s : entropy
- γ: adiabatic index
- c_V: specific heat

and the Equation of State

$$\varepsilon = \frac{1}{\gamma - 1}\frac{p}{\rho} \tag{7}$$

The Euler equations can be written as a conservation law as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{U}) = 0 \tag{8}$$

where we neglect any outer forces, i.e. $\mathbf{a} = 0$, and

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ E \end{pmatrix}, \qquad \mathbf{F}(\mathbf{U}) = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\otimes\mathbf{v} + p \\ (E+p)\mathbf{v} \end{pmatrix} \tag{9}$$

2.1.1 Euler equations in 1D

In 1D, we can write the Euler equations without source terms ($\mathbf{a} = 0$) as

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} = 0 \tag{10}$$

or explicitly (remember $\mathbf{v} = (u, v, w)$)

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} \rho u \\ \rho u^2 + p \\ (E+p)u \end{pmatrix} = 0 \tag{11}$$

2.1.2 Euler equations in 2D

In 2D, we have without source terms ($\mathbf{a} = 0$)

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial y} = 0 \tag{12}$$

or explicitly (remember $\mathbf{v} = (u, v, w)$)

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{pmatrix} = 0 \tag{13}$$

2.2 Conserved and primitive variables

For now, we have described the Euler equation as a hyperbolic conservation law using the (conserved) state vector $\mathbf{U}$:

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho \mathbf{v} \\ E \end{pmatrix} \tag{14}$$

for this reason, the variables ρ, $\rho\mathbf{v}$, and E are referred to as "conserved variables", as they obey conservation laws.

However, this is not the only set of variables that allows us to describe the fluid dynamics. In particular, the solution of the Riemann problem (section **??**) will give us a set of with so called "primitive variables" (or "physical variables") with the "primitive" state vector $\mathbf{W}$

$$\mathbf{W} = \begin{pmatrix} \rho \\ \mathbf{v} \\ p \end{pmatrix} \tag{15}$$

Using the ideal gas equations, these are the equations to translate between primitive and conservative variables:

Primitive to conservative:

$$(\rho) = (\rho) \tag{16}$$

$$(\rho\mathbf{v}) = (\rho) \cdot (\mathbf{v}) \tag{17}$$

$$(E) = \frac{1}{2}(\rho)(\mathbf{v})^2 + \frac{(p)}{\gamma - 1} \tag{18}$$

Conservative to primitive:

$$(\rho) = (\rho) \tag{19}$$

$$(\mathbf{v}) = \frac{(\rho\mathbf{v})}{(\rho)} \tag{20}$$

$$(p) = (\gamma - 1)\left((E) - \frac{1}{2}\frac{(\rho\mathbf{v})^2}{(\rho)}\right) \tag{21}$$

2.3 Implementation Details

TODO

Needs checking

All the functions for computing gas related quantities are written in `gas.h` and `gas.c`. Every cell is represented by a struct `struct cell` written in `cell.h`. It stores both the primitive variables/states and the conservative states in the `pstate` and `cstate` structs, respectively.

The adiabatic index γ is hardcoded as a macro in `defines.h`. If you change it, all the derived quantities stored in macros (e.g. $\gamma - 1$, $\frac{1}{\gamma - 1}$) should be computed automatically.

3 Notation

TODO

Needs updating. Indices are particles now, not cells.

We are working on numerical methods. Both space and time will be discretized.

Space will be discretized in cells which will have integer indices to describe their position. Time will be discretized in fixed time steps, which may have variable lengths. Nevertheless the time progresses step by step.

The lower left corner has indices $(0, 0)$ in 2D. In 1D, index 0 also represents the leftmost cell.

We have:

- integer subscript: Value of a quantity at the cell, i.e. the center of the cell. Example: $\mathbf{U}_i$, $\mathbf{U}_{i-2}$ or $\mathbf{U}_{i,j+1}$ for 2D.
- non-integer subscript: Value at the cell faces, e.g. $\mathbf{F}_{i-1/2}$ is the flux at the interface between cell i and $i - 1$, i.e. the left cell as seen from cell i.
- integer superscript: Indication of the time step. E.g. $\mathbf{U}^n$: State at timestep n
- non-integer superscript: (Estimated) value of a quantity in between timesteps. E.g. $\mathbf{F}^{n+1/2}$: The flux at the middle of the time between steps n and $n + 1$.

4 Hydrodynamics Methods

4.1 Smoothed Particle Hydrodynamics

4.2 Meshless Methods

5 Kernels

6 Searching For Neighbouring Particles

Hydrodynamics is local, and the particle interactions will only depend on a few neighbouring particles. How many neighbours we use is a parameter that we can set as `nngb` in the parameter file. So for every particle, we only want them to interact with a given number of neighbours, which we need to identify first. A naive way of doing that is comparing for each particle the distances of all other particles. However, that results in an $\mathcal{O}(N^2)$ algorithm, which can get quite expensive when the number of particles increases, especially since at least in principle the neighbour search needs to be done every time step. Instead, a more efficient way divide up the simulation box in cells, and distribute particles based on their position in those cells.

6.1 The Cell Grid

There are many good ways of distributing particles in cells, for example we could build adaptive trees and ensure a minimal or maximal number of particles in each cell. But that is not the point of this program, where we want to focus on the hydrodynamics. So instead, I built a simple uniform grid, where all cells have equal size.

The determination of the smoothing length (see Section 7) needs to be done iteratively. Hence we need enough particles so that the iteration can be performed. The criterion for "enough particles" is set to be that the number of particles in any given cell and all its neighbours must be at least `CELL_MIN_PARTS_IN_NEIGHBOURHOOD_FACT` $\times$ `nngb` particles. `CELL_MIN_PARTS_IN_NEIGHBOURHOOD_FACT` is set in `defines.h`. If this is not the case, we reduce the number of cells, making each cell bigger, and try again.

7 Smoothing Lengths

8 Boundary Conditions

9 Source Terms: Dealing with External Forces

TODO

Needs revision

So far, the hydro solvers have been dealing with the homogeneous, i.e. source free, Euler equations. But what if external forces like gravity or combustion are present? Then the full governing equations are given by

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\mathbf{v}\\ E\end{pmatrix} + \nabla\cdot\begin{pmatrix}\rho\mathbf{v}\\ \rho\mathbf{v}\otimes\mathbf{v}+p\\ (E+p)\mathbf{v}\end{pmatrix} = \begin{pmatrix}0\\ \rho\mathbf{a}\\ \rho\mathbf{a}\mathbf{v}\end{pmatrix} \tag{22}$$

where **a** is some external (velocity-independent) acceleration resulting from an external force. In 1D, the equations can be written as

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u\\ E\end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix}\rho u\\ \rho u^2+p\\ (E+p)u\end{pmatrix} = \begin{pmatrix}0\\ \rho a\\ \rho a u\end{pmatrix} \tag{23}$$

$$\frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{U}) = \mathbf{S}(\mathbf{U}) \tag{24}$$

Just like for the dimensional splitting approach, one can derive first and second order accurate schemes to include source terms by using the operator splitting technique: We split the PDE and solve for the two “operators” $\frac{\partial}{\partial x}\mathbf{F}(\mathbf{U})$ and $\mathbf{S}(\mathbf{U})$ individually, where we use the updated solution of the application of the first operator as the initial conditions for the application of the second operator.

Let $F(\Delta t)$ describe the operator $\frac{\partial}{\partial x}\mathbf{F}(\mathbf{U})$ applied over a time step Δt, and let $S(\Delta t)$ be the operator $\mathbf{S}(\mathbf{U})$ over a time step Δt, such that

$$\frac{\partial\mathbf{U}}{\partial t} + F[\mathbf{U}] = S[\mathbf{U}] \tag{25}$$

analytically. To obtain a first order accurate scheme, we need to complete the following two steps:

1. We obtain an intermediate result $\mathbf{U}^{n+1/2}$ by solving the homogeneous Euler equation over the full time interval Δt :

$$\begin{cases} \text{PDE:} & \frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{U}) = 0 \\ \text{IC:} & \mathbf{U}^n \end{cases} \tag{26}$$

$$\mathbf{U}^{n+1/2} = F(\Delta t)[\mathbf{U}^n] \tag{27}$$

2. then we evolve the solution to the final $\mathbf{U}^{n+1}$ by solving source ODE over the full time interval Δt :

$$\begin{cases} \text{PDE:} & \frac{\partial}{\partial t}\mathbf{U} = \mathbf{S} \\ \text{IC:} & \mathbf{U}^{n+1/2} \end{cases} \tag{28}$$

$$\mathbf{U}^{n+1} = S(\Delta t)[\mathbf{U}^{n+1/2}] \tag{29}$$

Possible integrators to solve this equation is discussed in section 9.2.

The order of which operator is solved first doesn't matter. The method is first order accurate as long as the operator F is also at least first order accurate. It can be shown that the operator splitting technique is second order accurate if we instead evolve the source term over half a time step twice:

$$\mathbf{U}^{n+1} = S(\Delta t/2)F(\Delta t)S(\Delta t/2)[\mathbf{U}^n] \tag{30}$$

To achieve second order accuracy, the operator F needs also to be at least second order accurate.

Finally, we already use the operator splitting technique to get a multidimensional scheme. How will a second order accurate scheme in multiple dimensions look like?

Let us express the two dimensional Euler equations as

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{pmatrix} = \begin{pmatrix} 0 \\ \rho a_x \\ \rho a_y \\ \rho \mathbf{a}\cdot\mathbf{v} \end{pmatrix} \tag{31}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial y} = \mathbf{S}(\mathbf{U}) \tag{32}$$

Let again $F(\Delta t)$ describe the operator $\frac{\partial}{\partial x}\mathbf{F}(\mathbf{U})$ applied over a time step Δt, $G(\Delta t)$ describe the operator $\frac{\partial}{\partial y}\mathbf{G}(\mathbf{U})$ applied over a time step Δt, and let $S(\Delta t)$ be the operator $\mathbf{S}(\mathbf{U})$ over a time step Δt, such that

$$\frac{\partial \mathbf{U}}{\partial t} + F[\mathbf{U}] + G[\mathbf{U}] = S[\mathbf{U}] \tag{33}$$

analytically. Let $H[\mathbf{U}]$ be the operator describing the homogeneous part of the Euler equations, i.e. $H[\mathbf{U}] = F[\mathbf{U}] + G[\mathbf{U}]$. The condition for first order accuracy is that in the scheme

$$\mathbf{U}^{n+1} = H(\Delta t)S(\Delta t)[\mathbf{U}^n] \tag{34}$$

$H[\mathbf{U}]$ must be at least first order accurate. This is given when we split the F and G operators just like for dimensional splitting:

$$H(\Delta t)[\mathbf{U}] = F(\Delta t)G(\Delta t)[\mathbf{U}] \tag{35}$$

So finally, a first order accurate two dimensional method is given by

$$\mathbf{U}^{n+1} = F(\Delta t)G(\Delta t)S(\Delta t)[\mathbf{U}] \tag{36}$$

The same logic can be applied for second order accuracy. A second order accurate operator $H(\Delta t)$ is given by

$$H(\Delta t)[\mathbf{U}] = F(\Delta t/2)G(\Delta t)F(\Delta t/2)[\mathbf{U}] \tag{37}$$

So finally, a second order accurate two dimensional method is given by

$$\begin{aligned}\mathbf{U}^{n+1} &= S(\Delta t/2)H(\Delta t)S(\Delta t/2)[\mathbf{U}] && (38)\\ &= S(\Delta t/2)F(\Delta t/2)G(\Delta t)F(\Delta t/2)S(\Delta t/2)[\mathbf{U}] && (39)\end{aligned}$$

9.1 Implemented Sources

Constant Acceleration

Probably the simplest external force terms are constant accelerations, i.e. the acceleration $\mathbf{a}$ doesn't change during the entire simulation. The value of the acceleration term $\mathbf{a}$ in each (Cartesian) direction can be set in the parameter file.

Constant Radial Acceleration

This one was mainly implemented to see whether acceleration works at all, and whether it works correctly. The acceleration is assumed to be constant and radial with origin at the middle of the simulation box, $(0.5, 0.5)$ in 2D or (0.5) in 1D. A positive acceleration value points away from the origin at the center of the box, a negative towards it.

9.2 Implemented Integrators

Throughout, we assume that the source terms $\mathbf{S}$ may depend both on the time t and on the conserved state $\mathbf{U}$: $\mathbf{S} = \mathbf{S}(t, \mathbf{U})$

Runge Kutta 2

The Runge Kutta 2 second-order explicit integrator is given by

$$\begin{aligned}\mathbf{K}_1 &= \Delta t\ \mathbf{S}(t^n, \mathbf{U}^n)\\ \mathbf{K}_2 &= \Delta t\ \mathbf{S}(t^n + \Delta t, \mathbf{U}^n + \mathbf{K}_1)\\ \mathbf{U}^{n+1} &= \mathbf{U}^n + \frac{1}{2}(\mathbf{K}_1 + \mathbf{K}_2)\end{aligned}$$

Runge Kutta 4

The Runge Kutta 4 fourth-order explicit integrator is given by

$$\begin{aligned}
\mathbf{K}_1 &= \Delta t\ \mathbf{S}(t^n, \mathbf{U}^n) \\
\mathbf{K}_2 &= \Delta t\ \mathbf{S}(t^n + \frac{1}{2}\Delta t, \mathbf{U}^n + \frac{1}{2}\mathbf{K}_1) \\
\mathbf{K}_3 &= \Delta t\ \mathbf{S}(t^n + \frac{1}{2}\Delta t, \mathbf{U}^n + \frac{1}{2}\mathbf{K}_2) \\
\mathbf{K}_4 &= \Delta t\ \mathbf{S}(t^n + \Delta t, \mathbf{U}^n + \mathbf{K}_3) \\
\mathbf{U}^{n+1} &= \mathbf{U}^n + \frac{1}{6}(\mathbf{K}_1 + 2\mathbf{K}_2 + 2\mathbf{K}_3 + \mathbf{K}_4)
\end{aligned}$$

References