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	Number of dimensions
	2	
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 10) from external forces. Can be left undefined.*	
	NONE	No source terms
	CONSTANT	Constant source terms (section 10.1)
	RADIAL	Radial constant source terms (section 10.1)
INTEGRATOR	*Which integrator to use to integrate the source terms (section 10) 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 10.2)
	RK4	Runge-Kutta 4 integrator (section 10.2)

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 ε = 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

5.1 General Properties

The core approach of SPH and meshless methods is the weighted interpolation approach, in particular for the fluid density:

$$\rho(\mathbf{r}) = \sum_{j}^{N_{neigh}} m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{22}$$

W is an as yet unspecified weight function. We want it to have following properties:

- Positive, decreases monotonically with distance, and is at least twice differentiable
- Spherical symmetry: $W(\mathbf{r} - \mathbf{r}_j, h) = W(|\mathbf{r} - \mathbf{r}_j|, h)$ such that no direction is preferred.
- Compact support: $W(r, H) = 0 \; \forall r \geq H$

  This allows us to treat the hydrodynamics locally, i.e. only close-by particles contribute to the hydrodynamic interactions, as opposed to all particles in the simulation all the time, and reduces computational cost.
- Normalisation: $\int_V W(\mathbf{r}' - \mathbf{r}_i, h) dV' = 1$

  This follows from the requirement to conserve total mass: If the total mass in a volume V is $M_{tot} = \sum_i^N m_i = \int_V \rho \mathrm{d}V$ for N total particles, then we can write

$$M_{tot} = \int_V \rho \mathrm{d}V = \int_V \sum_j^N m_j W(\mathbf{r} - \mathbf{r}_j, h) \mathrm{d}V = \sum_j^N m_j \int_V W(\mathbf{r} - \mathbf{r}_j, h) \mathrm{d}V = \sum_i^N m_i \tag{23}$$

  which is only satisfied if $\int_V W(\mathbf{r} - \mathbf{r}_j, h) \mathrm{d}V = 1$. Note that the reduction from the total number of particles N to the number of neighbours N_{neigh} in eq. 22 comes from our requirement for compact support of the kernel, that limits the number of interactions to only some neighbours. In principle however we always sum over all
- A flat central portion so the density estimate is not strongly affected by a small change in position of a near neighbour

Table 1: Functional forms, normalisation constants C and compact support length to smoothing length ratio $\Gamma = H/h$ for various popular kernel functions. ν is the dimension, and the kernel functions $w(q)$ are defined as $w(q = r/H) = 0$ for $q > 1$ i.e. $r > H$. $(\cdot)_+$ is shorthand for $\max(0, \cdot)$. Adapted from Dehnen and Aly [2012].

kernel name	kernel function $w(q)$	C			$\Gamma = H/h$		
		$\nu = 1$	$\nu = 2$	$\nu = 3$	$\nu = 1$	$\nu = 2$	$\nu = 3$
cubic spline	$(1-q)_+^3 - 4(\frac{1}{2}-q)_+^3$	$\frac{8}{3}$	$\frac{80}{7\pi}$	$\frac{16}{\pi}$	1.732051	1.778002	1.825742
quartic spline	$(1-q)_+^4 - 5(\frac{3}{5}-q)_+^4 + 10(\frac{1}{5}-q)_+^4$	$\frac{5^5}{768}$	$\frac{5^6 3}{2398\pi}$	$\frac{5^6}{512\pi}$	1.936492	1.977173	2.018932
quintic spline	$(1-q)_+^5 - 6(\frac{2}{3}-q)_+^5 + 15(\frac{1}{3}-q)_+^5$	$\frac{3^5}{40}$	$\frac{3^7 7}{478\pi}$	$\frac{3^7}{40\pi}$	2.121321	2.158131	2.195775
Wendland C^2, $\nu = 1$	$(1-q)_+^3(1+3q)$	$\frac{5}{4}$			1.620185		
Wendland C^4, $\nu = 1$	$(1-q)_+^5(1+5q+8q^2)$	$\frac{3}{2}$			1.936492		
Wendland C^6, $\nu = 1$	$(1-q)_+^7(1+7q+19q^2+21q^3)$	$\frac{55}{32}$			2.207940		
Wendland C^2, $\nu = 2,3$	$(1-q)_+^4(1+4q)$		$\frac{7}{\pi}$	$\frac{21}{2\pi}$		1.897367	1.936492
Wendland C^4, $\nu = 2,3$	$(1-q)_+^6(1+6q+\frac{35}{3}q^2)$		$\frac{9}{\pi}$	$\frac{495}{32\pi}$		2.171239	2.207940
Wendland C^6, $\nu = 2,3$	$(1-q)_+^8(1+8q+25q^2+32q^3)$		$\frac{78}{7\pi}$	$\frac{1365}{64\pi}$		2.415230	2.449490

There are many functions that satisfy these conditions. In general, we use the definition

$$W(\mathbf{r}, h) = \frac{C}{H^\nu} w(|\mathbf{r}|/H) = \frac{C}{H^\nu} w(q) \tag{24}$$

Some popular (and implemented) kernel functions are given in Table 1.

Following the work of Dehnen and Aly [2012], we distinguish between the smoothing length (smoothing *scale*) h and the compact support radius H for a kernel function. They showed that the smoothing length can be directly related to the numerical resolution of sound waves, and is as such a measure of resolution. Depending on the kernel used, the smoothing length and the kernel support radius have varying ratios. Those ratios are given in Table 1.

5.2 Kernel Derivatives

Computing kernel derivatives can easily be messed up, particularly so when using shorthand index notation, so we write them down explicitly here. Note that Cartesian coordinates are denoted as $\mathbf{x}$, and radial coordinates are denoted as r, θ, ϕ.

If a kernel is defined as (note the particle index notation)

$$W_j(\mathbf{x}_i) = W(\mathbf{x}_i - \mathbf{x}_j, h_i)) = \frac{C}{H_i^\nu} w\left(\frac{|\mathbf{x}_i - \mathbf{x}_j|}{H_i}\right) = \frac{C}{H_i^\nu} w(q_{ij}) \tag{25}$$

and we assume that the smoothing length h_i is treated as constant at this point w.r.t. spatial dimensions ($\mathbf{x}$ and r, θ, ϕ alike).

First, let us recall that we define

$$r_{ij} \equiv |\mathbf{x}_i - \mathbf{x}_j| \tag{26}$$

$$q_{ij} \equiv \frac{r_{ij}}{H_i} \tag{27}$$

We can pre-compute

$$\frac{\partial q_{ij}(r_{ij})}{\partial r_{ij}} = \frac{\partial}{\partial r_{ij}} \frac{r_{ij}}{H_i} = \frac{1}{H_i} \tag{28}$$

$$\frac{\partial q_{ij}(q_{ij})}{\partial h_i} = \frac{\partial}{\partial h_i} \frac{r_{ij}}{H_i} = -\frac{r_{ij}}{H_i^2} \frac{\partial}{\partial h_i} H_i(h_i) = -\Gamma \frac{r_{ij}}{H_i^2} \tag{29}$$

$$\frac{\partial r_{ij}}{\partial x} = \frac{\partial}{\partial x} \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2} = \frac{1}{2} \frac{1}{\sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2}} \cdot 2(\mathbf{x}_i - \mathbf{x}_j) = \frac{\mathbf{x}_i - \mathbf{x}_j}{r_{ij}} \tag{30}$$

In eq. 29 we use the fact that $H = \Gamma h$. Values for Γ are given in Table 1.

The radial derivative of the kernel along the line between particles i and j is

$$\begin{aligned} \frac{\partial}{\partial r_{ij}} W_j(\mathbf{x}_i) &= \frac{\partial}{\partial r_{ij}} \left(\frac{C}{H_i^\nu} w(q_{ij}) \right) \\ &= \frac{C}{H_i^\nu} \frac{\partial w(q_{ij})}{\partial q_{ij}} \frac{\partial q_{ij}(r_{ij})}{\partial r_{ij}} \\ &= \frac{C}{H_i^\nu} \frac{\partial w(q_{ij})}{\partial q_{ij}} \frac{1}{H_i} \\ &= \frac{C}{H_i^{\nu+1}} \frac{\partial w(q_{ij})}{\partial q_{ij}} \end{aligned} \tag{31}$$

or, neglecting particle indices:

$$\frac{\partial}{\partial r} W(\mathbf{x}) = \frac{C}{H_i^{\nu+1}} \frac{\partial w(q)}{\partial q} \tag{32}$$

The Cartesian gradient of the kernel is given by

$$\begin{aligned}
\frac{\partial}{\partial x} W_j(\mathbf{x}_i) &= \frac{\partial}{\partial x}\left(\frac{C}{H_i^\nu} w(q_{ij})\right) \\
&= \frac{C}{H_i^\nu}\frac{\partial w(q_{ij})}{\partial q_{ij}}\frac{\partial q_{ij}(r_{ij})}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x} \\
&= \frac{C}{H_i^{\nu+1}}\frac{\partial w(q_{ij})}{\partial q_{ij}}\frac{\mathbf{x}_i - \mathbf{x}_j}{r_{ij}}
\end{aligned} \tag{33}$$

The derivative with respect to the smoothing length is given by

$$\begin{aligned}
\frac{\partial}{\partial h_i} W_j(\mathbf{x}_i) &= \frac{\partial}{\partial h_i}\left(\frac{C}{H_i^\nu} w(q_{ij})\right) \\
&= -\frac{\nu C}{H_i^{\nu+1}}\Gamma w(q_{ij}) + \frac{C}{H_i^\nu}\frac{\partial w(q_{ij})}{\partial q_{ij}}\frac{\partial q_{ij}(r_{ij})}{\partial h_i} \\
&= -\frac{\nu C\Gamma}{H_i^{\nu+1}} w(q_{ij}) + \frac{C}{H_i^\nu}\frac{\partial w(q_{ij})}{\partial q_{ij}}\left(-\Gamma\frac{r_{ij}}{H_i^2}\right) \\
&= -\frac{\nu C\Gamma}{H_i^{\nu+1}} w(q_{ij}) - \frac{\Gamma C}{H_i^{\nu+2}}\frac{\partial w(q_{ij})}{\partial q_{ij}} r_{ij} \\
&= -\frac{\nu\Gamma}{H_i} W(\mathbf{x}_i) - \frac{\Gamma}{H_i}\frac{\partial W(\mathbf{x}_i)}{\partial r} r_{ij} \\
&= -\frac{\Gamma}{H_i}\left(\nu W(\mathbf{x}_i) + r_{ij}\frac{\partial W(\mathbf{x}_i)}{\partial r}\right) \\
&= -\frac{1}{h_i}\left(\nu W(\mathbf{x}_i) + r_{ij}\frac{\partial W(\mathbf{x}_i)}{\partial r}\right)
\end{aligned} \tag{34}$$

In summary:

$$\begin{aligned}
\frac{\partial}{\partial r} W(\mathbf{x}) &= \frac{C}{H_i^{\nu+1}}\frac{\partial w(q)}{\partial q} \\
\frac{\partial}{\partial x} W_j(\mathbf{x}_i) &= \frac{C}{H_i^{\nu+1}}\frac{\partial w(q_{ij})}{\partial q_{ij}}\frac{\mathbf{x}_i - \mathbf{x}_j}{r_{ij}} \\
\frac{\partial}{\partial h} W(\mathbf{x}) &= -\frac{1}{h}\left(\nu W(\mathbf{x}) + r_{ij}\frac{\partial W(\mathbf{x})}{\partial r}\right)
\end{aligned}$$

Table 2: Derivatives of the functional form for kernel functions $\frac{\partial w(p)}{\partial r}$.

kernel	derivatives
to	do

Values for Γ and C are given in table 1. Functional forms for the derivatives of the kernel functions, $\frac{\partial w(q)}{\partial r}$ are given in table 2.

5.3 On the definition of r_{ij}

The definition of r_{ij} requires a bit more discussion. Since kernels used in hydrodynamics are usually taken to be spherically symmetric, we might as well have defined

$$r'_{ij} = |\mathbf{x}_j - \mathbf{x}_i|$$

which would leave the evaluation of the kernels invariant $[r'_{ij} = r_{ij}]$, but the gradients would have the opposite direction:

$$\frac{\partial r'_{ij}}{\partial x} = \frac{\mathbf{x}_j - \mathbf{x}_i}{r'_{ij}} = -\frac{\mathbf{x}_i - \mathbf{x}_j}{r_{ij}} = -\frac{\partial r_{ij}}{\partial x}$$

So which definition should we take?

Consider a one-dimensional case where we choose two particles i and j such that $x_j > x_i$ and $q_{ij} = |x_j - x_i|/H_i < 1$. Because we're considering a one-dimensional case with $x_j > x_i$, we can now perform a simple translation such that particle i is at the origin, i.e. $x'_i = 0$ and $x'_j = x_j - x_i = |x_j - x_i| = r_{ij}$. In this scenario, the gradient in Cartesian coordinates and in spherical coordinates must be the same:

$$\frac{\partial}{\partial x'} W(|x'_i - x'|, h_i)\big|_{x'=x'_j} = \frac{\partial}{\partial r_{ij}} W(r_{ij}, h_i)$$

$$\Rightarrow \quad \frac{1}{h_i^\nu}\frac{\partial w(q'_{ij})}{\partial q'_{ij}}\frac{\partial q'_{ij}(r'_{ij})}{\partial r'_{ij}}\frac{\partial r'_i(x')}{\partial x'}\Big|_{x'=x'_j} = \frac{1}{h_i^\nu}\frac{\partial w(q_{ij})}{\partial q_{ij}}\frac{\partial q_{ij}(r_{ij})}{\partial r_{ij}} \tag{35}$$

We have the trivial case where

$$\begin{aligned} r'_{ij} &= |x'_i - x'_j| = |x_i - x_j| = r_{ij} \\ q'_{ij} &= r'_{ij}/h_i = q_{ij} \\ \Rightarrow \quad \frac{\partial w(q'_{ij})}{\partial q'_{ij}} &= \frac{\partial w(q_{ij})}{\partial q_{ij}}, \quad \frac{\partial q'_{ij}(r'_{ij})}{\partial r'_{ij}} = \frac{\partial q_{ij}(r_{ij})}{\partial r_{ij}} \end{aligned}$$

giving us the condition from 35:

$$\frac{\partial r'_i(x')}{\partial x'}\Big|_{x'=x'_j} = 1 = \frac{\partial r_i(x)}{\partial x} \tag{36}$$

this is satisfied for

$$\begin{aligned} & r_j(\mathbf{x}) = |\mathbf{x} - \mathbf{x}_j| \\ & \Rightarrow r_{ij} = |\mathbf{x}_i - \mathbf{x}_j|, \quad \text{not} \quad r_{ij} = |\mathbf{x}_j - \mathbf{x}_i| \end{aligned}$$

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.

6.2 Finding Neighbours to interact with

Once the smoothing length h_i (see Section 7 on how that's done) for a particle i is set, we don't want to keep all the particles from all surrounding cells in memory as neighbours, but only keep those that are going to interact with this particle, i.e. only those that fit within the compact support radius H_i of the kernel in use. To this end, we store for each particle i the neighbouring particle indices with distance smaller than H_i in the `int* neigh_iact`, where the index refers to the particle's index in the `particles` array, as well as the distance r_{ij} to each neighbouring particle j into the array `float* r`. These arrays will be sorted by ascending particle index to make interactions during particle loops simpler (see Section 8).

7 Smoothing Lengths

7.1 Computing the Smoothing Lengths

There is no unique way of defining how the smoothing length should be set. You could have for example a fixed smoothing length for all particles everywhere, or let it evolve over time based on the densest region in your simulation, or whatnot. However, it is desirable to resolve both clustered and sparse regions of your simulation evenly, i.e. with a roughly constant ratio of h to the mean local particle separation. Thus a natural choice for setting the smoothing length is to relate to the local number density of particles, i.e.

$$h(\mathbf{r}_i) \propto n(\mathbf{r}_i)^{-1/\nu} \tag{37}$$

for ν dimensions. Let η be a parameter specifying the smoothing length in units of the mean particle spacing $n^{-1/\nu} = (\rho/m)^{-1/\nu}$. Then

$$h(\mathbf{r}_i) = \eta n(\mathbf{r}_i)^{-1/\nu} = \eta \left(\frac{\rho_i}{m_i}\right)^{-1/\nu} = \eta \left(\frac{m_i}{\rho_i}\right)^{1/\nu} \tag{38}$$

and simultaneously, we have

$$\rho_i = \sum_n m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) \tag{39}$$

to solve, where n refers to all neighbouring particles of i, including itself.

This set of equations can be solved iteratively using the Newton-Raphson root finding algorithm: For a differentiable function $f(x)$, we find x_{root} that satisfies $f(x_{root}) = 0$ by iterating

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{\mathrm{d}f}{\mathrm{d}x}(x_n)} \tag{40}$$

until we reach some tolerance ε:

$$|f(x_n)| \leq \varepsilon \tag{41}$$

In our scenario, we have

$$\begin{aligned} h_i &= \eta \left(\frac{m_i}{\rho_i}\right)^{1/\nu} \\ \Rightarrow f = f(h_i) &= h_i - \eta \left(\frac{m_i}{\rho_i}\right)^{1/\nu} = 0 \end{aligned}$$

and we can use the root finding algorithm. All we need is $\frac{\mathrm{d}f}{\mathrm{d}h_i}$:

$$\begin{aligned}
\frac{\mathrm{d}f}{\mathrm{d}h_i} &= 1 - \eta m_i^{1/\nu} \frac{\mathrm{d}}{\mathrm{d}h_i}\rho_i^{-1/\nu} \\
&= 1 + \eta m_i^{1/\nu}\frac{1}{\nu}\,\rho_i^{-1/\nu-1}\frac{\mathrm{d}\rho_i}{\mathrm{d}h_i} \\
&= 1 + \eta m_i^{1/\nu}\frac{1}{\nu}\,\rho_i^{-1/\nu-1}\sum_n m_j \frac{\mathrm{d}}{\mathrm{d}h_i}W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) \\
&= 1 - \eta m_i^{1/\nu}\frac{1}{\nu}\,\rho_i^{-1/\nu-1}\sum_n m_j \frac{1}{h_i}\left(\nu W(\mathbf{x}_i) + r_{ij}\frac{\partial W(\mathbf{x}_i)}{\partial r}\right) \\
&= 1 - \frac{\eta}{h_i}\frac{m_i^{1/\nu}}{\rho_i^{1+1/\nu}}\sum_n m_j \left(W(\mathbf{x}_i) + \frac{r_{ij}}{\nu}\frac{\partial W(\mathbf{x}_i)}{\partial r}\right)
\end{aligned} \tag{42}$$

However, a perhaps better version can be obtained via

$$\begin{aligned}
h_i &= \eta n^{-\nu} \\
\text{Then} \quad n h_i^\nu &= \eta^\nu \\
\text{Giving us} \quad f &= f(h_i) = n h_i^\nu - \eta^\nu = h_i^\nu \sum_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) - \eta^\nu = 0
\end{aligned} \tag{43}$$

$$\begin{aligned}
\frac{\mathrm{d}f}{\mathrm{d}h_i} &= \frac{\mathrm{d}}{\mathrm{d}h_i}\left(h_i^\nu \sum_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i)\right) \\
&= \nu h_i^{\nu-1}\sum_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) + h_i^\nu \sum_j \frac{\mathrm{d}}{\mathrm{d}h_i}W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) \\
&= \nu h_i^{\nu-1}\sum_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i) - h_i^\nu \sum_j \frac{1}{h_i}\left(\nu W(\mathbf{x}_i) + r_{ij}\frac{\partial W(\mathbf{x}_i)}{\partial r}\right)
\end{aligned} \tag{44}$$

The main advantage is that instead of the mass density, only the number density $n = \sum_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_i)$ is used, and therefore the determination of the smoothing length (and with it the determination of the resolution) only depends on particle positions and no other fluid properties. This doesn't change much for simulations where all SPH particles have equal masses, but in finite volume meshless methods, where masses are exchanged via fluxes, the smoothing lengths will be weighted by the variable particle masses. This is also the implemented version.

7.2 η and N_{neigh}

Add discussion that is in Price [2012].

8 Particle Interaction Loops

8.1 Interaction Loops in General

8.2 SPH

8.3 Meshless Methods

9 Boundary Conditions

10 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

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\mathbf{v}\\ E\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\mathbf{v}\\ \rho\mathbf{v}\otimes\mathbf{v}+p\\ (E+p)\mathbf{v}\end{pmatrix}=\begin{pmatrix}0\\ \rho\mathbf{a}\\ \rho\mathbf{a}\mathbf{v}\end{pmatrix} \tag{45}$$

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

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u\\ E\end{pmatrix}+\frac{\partial}{\partial x}\begin{pmatrix}\rho u\\ \rho u^2+p\\ (E+p)u\end{pmatrix}=\begin{pmatrix}0\\ \rho a\\ \rho a u\end{pmatrix} \tag{46}$$

$$\frac{\partial}{\partial t}\mathbf{U}+\frac{\partial}{\partial x}\mathbf{F}(\mathbf{U})=\mathbf{S}(\mathbf{U}) \tag{47}$$

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

$$\frac{\partial\mathbf{U}}{\partial t}+F[\mathbf{U}]=S[\mathbf{U}] \tag{48}$$

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 :

$$\begin{cases} \text{PDE:} & \frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{U}) = 0 \\ \text{IC:} & \mathbf{U}^n \end{cases} \tag{49}$$

$$\mathbf{U}^{n+1/2} = F(\Delta t)[\mathbf{U}^n] \tag{50}$$

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

$$\begin{cases} \text{PDE:} & \frac{\partial}{\partial t}\mathbf{U} = \mathbf{S} \\ \text{IC:} & \mathbf{U}^{n+1/2} \end{cases} \tag{51}$$

$$\mathbf{U}^{n+1} = S(\Delta t)[\mathbf{U}^{n+1/2}] \tag{52}$$

   Possible integrators to solve this equation is discussed in section 10.2.

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:

$$\mathbf{U}^{n+1} = S(\Delta t/2)F(\Delta t)S(\Delta t/2)[\mathbf{U}^n] \tag{53}$$

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

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{pmatrix} = \begin{pmatrix} 0 \\ \rho a_x \\ \rho a_y \\ \rho \mathbf{a}\cdot\mathbf{v} \end{pmatrix} \tag{54}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial y} = \mathbf{S}(\mathbf{U}) \tag{55}$$

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

$$\frac{\partial \mathbf{U}}{\partial t} + F[\mathbf{U}] + G[\mathbf{U}] = S[\mathbf{U}] \tag{56}$$

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

$$\mathbf{U}^{n+1} = H(\Delta t)S(\Delta t)[\mathbf{U}^n] \tag{57}$$

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

$$H(\Delta t)[\mathbf{U}] = F(\Delta t)G(\Delta t)[\mathbf{U}] \tag{58}$$

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

$$\mathbf{U}^{n+1} = F(\Delta t)G(\Delta t)S(\Delta t)[\mathbf{U}] \tag{59}$$

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

$$H(\Delta t)[\mathbf{U}] = F(\Delta t/2)G(\Delta t)F(\Delta t/2)[\mathbf{U}] \tag{60}$$

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

$$\mathbf{U}^{n+1} = S(\Delta t/2)H(\Delta t)S(\Delta t/2)[\mathbf{U}] \tag{61}$$
$$= S(\Delta t/2)F(\Delta t/2)G(\Delta t)F(\Delta t/2)S(\Delta t/2)[\mathbf{U}] \tag{62}$$

10.1 Implemented Sources

Constant Acceleration

Probably the simplest external force terms are constant accelerations, i.e. the acceleration **a** doesn't change during the entire simulation. The value of the acceleration term **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.

10.2 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

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

11 Generating Initial Conditions

11.1 The Algorithm

Generating initial conditions for SPH-type of simulations is a bit more tricky than for grids. Grid cells have well defined volumes, and we can just distribute masses in the volumes to obtain densities, or simply follow the density profile to assign the cell center value. For SPH on the other hand, the densities are determined both through the particle positions and particle masses, both of which we need to set simultaneously to reproduce the density that we want to have. Even when keeping particle masses fixed, just shifting particle positions influences the density at the particle position, since the kernel weights naturally change too.

An elegant way of generating initial conditions for SPH simulations is described by **?**. The essential idea is to start with some initial particle configuration, then compute some sort of "force" on the particles that will displace them towards the model density, and re-iterate until the configuration has relaxed sufficiently.

To start off, assuming the particle number that we want is set, we need to find some initial particle placements. Typical choices are uniform particle distributions, random distributions, or configurations obtained by rejection sampling the density that you want to reproduce.

Secondly, the particle masses need to be determined. It is in our interest that the SPH particles have equal masses, hence we require the total mass in the simulation box as determined by the given density. To this end, the density field is integrated numerically, and the total mass divided equally amongst the particles.

Every iteration step follows these steps:

1. Compute "model smoothing lengths"[1] h_i^m:

$$h_i^m = \left(\frac{N_{ngb}}{V_\nu} \frac{L_x L_y L_z}{\rho_m(\mathbf{x}_i) \sum_j^N 1/\rho_m(\mathbf{x}_i)} \right)^{1/\nu} \tag{63}$$

   V_ν is the volume of the unit "sphere" in ν dimensions, namely

$$V_1 = 2, \qquad V_2 = \pi, \qquad V_3 = \frac{4}{3}\pi$$

[1] The given formula in eq. 63 is not the one as given in **?**, but it is the one used in their code, and it appears that it works well.

L_x, L_y, and L_z are the box sizes in x, y, and z direction, respectively, which I take to be unity. Note that the sum in eq. 63 is the sum over all particles, not over a single particle's neighbours. They are based only on the simulation parameters and the particle positions, and don't need to be computed like proper smoothing lengths in step section 7.

2. Find the neighbours of each particle, and compute the "displacement force" on each particle. The "displacement force" experienced by particle i due to particle j is given by

$$\Delta r = C_{\Delta r} h_{ij} W(|\mathbf{x}_i - \mathbf{x}_j|, h_{ij}) \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|} \tag{64}$$

Where $C_{\Delta r}$ is a negative constant discussed later, and

$$h_{ij} = \frac{h_i + h_j}{2} \tag{65}$$

and only contributions from all neighbours j of particle i are taken into account.

3. Move particles with the computed Δr

4. Optionally, displace some overdense particles close to underdense particles. Typically, this is not done every step, but after some set number of iteration steps have passed. A fraction (which will decrease as the iteration count increases) of overdense particles is selected to be moved, if the following condition is satisfied:

$$r_o \in [0, 1] < \text{erf}\left(\frac{\rho_i - \rho_m(\mathbf{x}_i)}{\rho_m(\mathbf{x}_i)}\right) \tag{66}$$

where r_o is a random number, ρ_i is the density of some overdense particle i and $\rho_m(\mathbf{x}_i)$ is the model density that we want our initial conditions to have at particle position $\mathbf{x}_i$. Similarly, an underdense particle j is chosen as a target if

$$r_u \in [0, 1] < \frac{\rho_m(\mathbf{x}_i) - \rho_i}{\rho_m(\mathbf{x}_i)} \tag{67}$$

This ensures that particles with large density differences to the model density are more likely to be chosen to be moved and to be the target of the movement. Once a target for some overdense particle is decided, the overdense particle is placed randomly around the target's coordinates with distance < 0.3 the kernel support radius.

A few thing remain to be discussed. Firstly, when can the iteration stop? When the initial conditions start to converge, the forces Δr decrease. So the first condition for convergence is that an upper threshold for any displacement is never reached. If that

is satisfied, we may consider the initial conditions to be converged when a large fraction (e.g. 99% or 99.9% or...) of the particles has a displacement lower than some "convergence displacement threshold", which typically should be lower than the upper threshold for any displacement. Finally, the iteration may stop if some maximal number of iterations has been completed.

Secondly, what constant $C_{\Delta r}$ should we use in eq. 64? It needs to be negative, but other than that, we're relatively free. In the code, it is defined in units of the mean interparticle distance

$$\bar{l} = 1/N^{1/\nu} \tag{68}$$

How large the Δr without the constant $C_{\Delta r}$ will be depends on multiple factors, like what density function you're trying to reproduce, how many neighbours you include in your kernel summation, etc. You should set it in a way such that the displacements at the start of the iteration are in the order of unity in units of $\bar{l}$.

11.2 User's Guide

The main function you will be calling is in `particle_hydro_IC.py`:

```
generate_IC_for_given_density(rho_anal, nx, ndim, eta, x=None,
                m=None, kernel='cubic spline', periodic=True)
```

Parameters:

```
    rho_anal:   function rho_anal(x). Should return a numpy
                array of the analytical function rho(x) for
                given coordinates x, where x is also a numpy
                array.
    nx:         How many particles you want along each
                dimension. Total number of particles will be
                nx^ndim
    ndim:       How many dimensions we're working with
    eta:        "resolution", that defines number of
                neighbours
    x:          Initial guess for coordinates of particles.
                If none, an initial guess will be generated
                by rejection sampling rho_anal.
                Should be numpy array of shape
                (nx^ndim, ndim) or None.
    m:          Numpy array of particle masses. If None, an
```

```
                array will be created such that the total mass
                in the simulation box given the analytical
                density is reproduced, and all particles will
                have equal masses.
        kernel:     which kernel to use
        periodic:   Whether we're having periodic boundary
                conditions or not.


returns:
        x:      numpy array of particle positions
        m:      numpy array of particle masses.
        rho:    numpy array of particle densities
        h:      numpy array of particle smoothing lengths
```

If you have a good guess for particle masses and initial positions, pass it on to the function, otherwise it'll create good guesses for you. There are also already implemented functions to give you either uniformly distributed particle configurations:

```
IC_uniform_coordinates(nx, ndim = 2, periodic = True)
```

or uniform coordinates that have been randomly displaced:

```
IC_perturbed_coordinates(nx, ndim = 2, periodic = True)
```

The parameters have the same meaning as above.

The iteration parameters are stored as global variables, and can be changed by calling

```
IC_generation_set_params(
        iter_max                        = IC_ITER_MAX,
        convergence_threshold           = IC_CONVERGENCE_THRESHOLD,
        tolerance_part                  = IC_TOLERANCE_PART,
        displacement_threshold          = IC_DISPLACEMENT_THRESHOLD,
        redistribute_at_iteration       = IC_REDISTRIBUTE_AT_ITERATION,
        delta_init                      = IC_DELTA_INIT,
        delta_reduction_factor          = IC_DELTA_REDUCTION_FACTOR,
        delta_min                       = IC_DELTA_MIN,
        redistribute_fraction           = IC_REDISTRIBUTE_FRACTION,
        no_redistribution_after         = IC_NO_REDISTRIBUTION_AFTER,
        plot_at_redistribution          = IC_PLOT_AT_REDISTRIBUTION
        )
```

The parameters in all caps and with the `IC_` prefix are the default values (and differ from the function parameters names only by being in all caps and having the prefix `IC_`).

Their meaning is:

type	name	default
int	IC_ITER_MAX	2000
	max numbers of iterations for generating IC conditions	
float	IC_CONVERGENCE_THRESHOLD	1e-3
	if enough particles are displaced by distance below IC_CONVERGENCE_THRESHOLD $*$ mean interparticle distance, stop iterating. Enough particles is defined as less than IC_TOLERANCE_PART particles being **not** converged.	
float	IC_TOLERANCE_PART	0.01
	tolerance for not converged particle fraction: this fraction of particles can be displaced with distances $>$ IC_CONVERGENCE_THRESHOLD	
float	IC_DISPLACEMENT_THRESHOLD	1e-2
	iteration halt criterion: While any particle is displaced by a distance $>$ IC_DISPLACEMENT_THRESHOLD $*$ mean interparticle distance, keep iterating.	
float	IC_DELTA_INIT	0.1
	Initial normalisation constant for particle displacement in units of mean interparticle distance.	
float	IC_DELTA_MIN	1e-4
	Minimal (=lower treshold) normalisation constant for particle displacement in units of mean interparticle distance.	
float	IC_DELTA_REDUCTION_FACTOR	0.97
	Reduce normalisation constant for particle displacement by this factor every iteration to force convergence after a while.	
int	IC_REDISTRIBUTE_AT_ITERATION	10
	Redistribute a handful of particles every IC_REDISTRIBUTE_AT_ITERATION-th iteration.	
float	IC_REDISTRIBUTE_FRACTION	0.01
	fraction of particles to redistribute when doing so.	
int	IC_NO_REDISTRIBUTION_AFTER	200
	Don't redistribute particles after this iteration number.	
boolean	IC_PLOT_AT_REDISTRIBUTION	True
	Create and store a plot of the current particle configuration and density before redistributing particles?	

References

Walter Dehnen and Hossam Aly. Improving convergence in smoothed particle hydrodynamics simulations without pairing instability. April 2012. doi: 10.1111/j.1365-2966.2012.21439.x.

Daniel J. Price. Smoothed Particle Hydrodynamics and Magnetohydrodynamics. *Journal of Computational Physics*, 231(3):759–794, February 2012. ISSN 00219991. doi: 10.1016/j.jcp.2010.12.011.