Smooth Particle Hydrodynamics

Marc de Miguel
Andrés Tamargo
Vincent Boulard
Radovan Dabetić

June 5, 2024

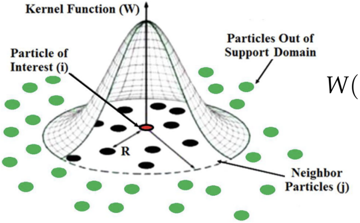
Content

Content

1. Introduction
2. Our C++ implementation
    - Nearest Neighbors
    - Numerical integration
    - Gridding problem
3. Results
    - Kelvin-Helmholtz Instability
    - 1D Sod-shock tube
4. Conclusion
5. References

Smoothed Particle Hydrodynamics

- Fluid discretized into particles.
- Physical properties obtained by smoothing over nearby particles using a **kernel**.
- EOM derived from Euler equations



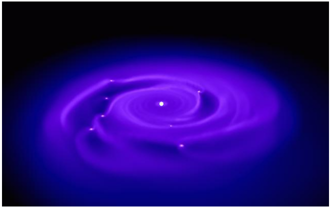
$$W(\mathbf{r}, h) = \sigma(h^d) \begin{cases} \frac{1}{4}(2-q)^3, & 1 \le q \le 2 \\ 1 - \frac{3}{2}q^2(1 - \frac{q}{2}), & 0 \le q \le 1 \\ 0, & q > 1 \end{cases}$$

$$q = \frac{|\mathbf{r}|}{h}$$

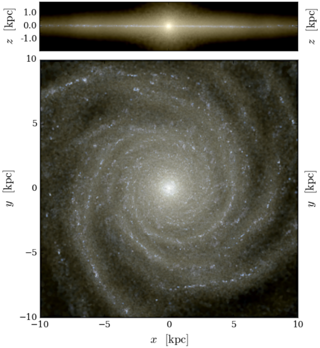
Smoothed Particle Hydrodynamics

- Some examples of SPH:

Gas giant formation simulation (GASOLINE).

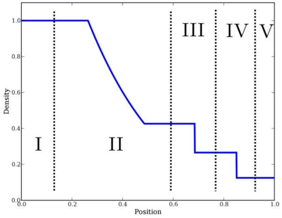


Galaxy modelling (GASOLINE).



Artificial viscosity and entropy

- Addition of a viscosity term to avoid discontinuities from shock waves.
- Entropy is allowed to increase



Example of discontinuity in space in a Shock Tube density profile.

Equations of motion

- Final equations for the particles (with viscosity term):

$$\boxed{\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\right) \nabla_i W_{ij}(h)}$$

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h), \quad P_i = A_i \rho_i^\gamma$$

$$\frac{dA_i}{dt} = \frac{1}{2} \frac{\gamma - 1}{\rho_i^{\gamma-1}} \sum_{j=1}^{N} m_j \Pi_{ij} \mathbf{v}_{ij} \nabla_i W_{ij}$$

Content

Approximate Nearest Neighbors

- Put a mesh over the domain
- Put the particles in cells of the mesh
- Use particles from nearby mesh cells for the smoothening

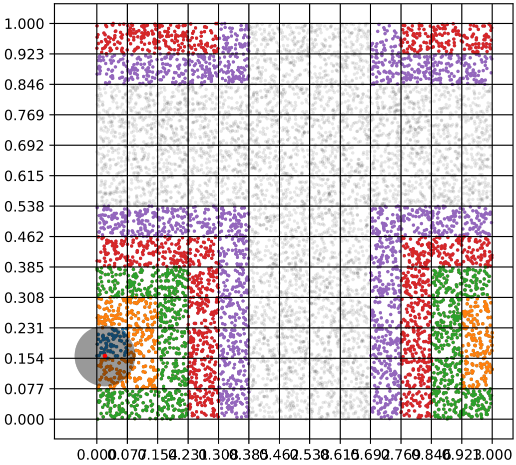


Figure 1: Example nearest-neighbor problem with periodic boundary conditions. The grid corresponds to the mesh used.

The “Hash”

$x_{i,0}$: origin coordinate i

$$\texttt{cell\_idx}(\mathbf{x})_i := \left\lfloor \frac{x_i - x_{i,0}}{\Delta x_i} \right\rfloor \qquad \mathbf{x} \in \mathbb{R}^d.$$

The “Hash”

$x_{i,0}$: origin coordinate i

$$\texttt{cell\_idx}(\mathbf{x})_i := \left\lfloor \frac{x_i - x_{i,0}}{\Delta x_i} \right\rfloor \qquad \mathbf{x} \in \mathbb{R}^d.$$

- How do we store the particles?
- How to retrieve them?

Considering Memory Locality

- IPPL: struct of vectors
- Potential solution: linked lists of indices in each cell

Considering Memory Locality

- IPPL: struct of vectors
- Potential solution: linked lists of indices in each cell
- Iterate over neighbors $\implies$ store neighbors close in memory
  $\implies$ sort

Considering Memory Locality

- IPPL: struct of vectors
- Potential solution: linked lists of indices in each cell
- Iterate over neighbors $\implies$ store neighbors close in memory
  $\implies$ sort
- Partition vector by number of particles in each cell
- Copy back in the right order via a temporary

```cpp
template <typename VEC_TYPE, typename D_TYPE>
void sort(const Kokkos::View<VEC_TYPE*>& pos,
          Kokkos::View<D_TYPE*>& target){
    // Current index in each cell
    Kokkos::View<IDX_TYPE*> current_idx("Current Index", ncells);
    Kokkos::deep_copy(current_idx, start_idx);
    // Create temporary
    Kokkos::View<D_TYPE*> temp_target("Temporary", target.size());
    // Copy in the right order
    Kokkos::parallel_for("Reorder-Loop", target.size(),
      KOKKOS_LAMBDA (const IDX_TYPE i){
        const IDX_TYPE key = idx_to_key(cell_idx(pos(i)));
        const IDX_TYPE new_idx =
            Kokkos::atomic_fetch_add(&current_idx(key), 1);
        temp_target(new_idx) = target(i);
      }
    );
    // Copy it back
    Kokkos::deep_copy(target, temp_target);
}
```

Iteration

Iterate over them with a triple loop

```
! Loop over particles
do p_idx = 0, N_particles
    ! Do something with the particle at p_idx
    dummy(p_idx)
    cell_neighbor_idx = get_neighbor_idx(position(p_idx))
    ! Loop over the neighbor cells
    do cell_idx = 0, cell_neighbor_idx.size
        ! Loop over particles in the cell
        do neighbor_idx_offset = 0, cell_size(cell_idx)
            neighbor_idx = start_idx(neighbor_idx(cell_idx))
                         + neighbor_idx_offset
            ! Perform some operation
            do_smth(p_idx, neighbor_idx)
        end do
    end do
end do
```

Iteration

Or even simpler with the help of templates

```
CM.it_over_neighbors(cell_idx, radius,
    [&](const std::size_t i){
        do_smth(i); // i == index of a neighbor
    }
);
```

Scheme choice

- Without viscosity, we choose a leapfrog scheme because it's symplectic.
- With viscosity, there is no Hamiltonian structure, so we choose RK2.

```
Kokkos::parallel_for(N_particles,
  KOKKOS_LAMBDA (const int p_idx){
    x_n(p_idx) = particles.position(p_idx);
    v_n(p_idx) = particles.velocity(p_idx);
    s_n(p_idx) = particles.entropy(p_idx);

    particles.position(p_idx) = x_n(p_idx) + (dt/2)*v_n(p_idx);
    particles.velocity(p_idx) = v_n(p_idx) + (dt/2)*particles.
        accel(p_idx);
    particles.entropy(p_idx) = s_n(p_idx) + (dt/2)*particles.
        d_entropy(p_idx);
    //here expect boundary conditions
}); //need to do a second step !
```

Smoothing kernel size

- For the simplicity we choose to have a constant h. But ideally what we should aim is something like this :

$$h_i^{dim}\rho_i = cste \quad \text{or} \quad h_i^{dim}\rho_i \propto m_i$$

Gridding behavior

- We think that having a constant h leads to this behavior.

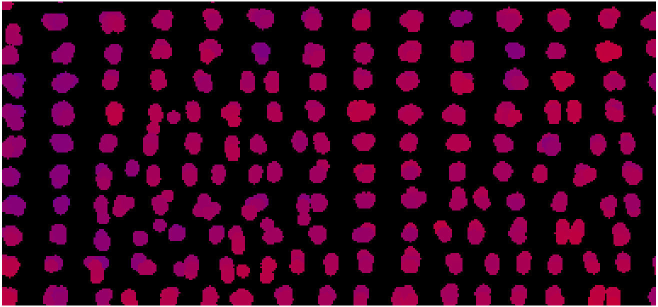


Figure 2: Illustration of the gridding behavior.

Content

Main codes

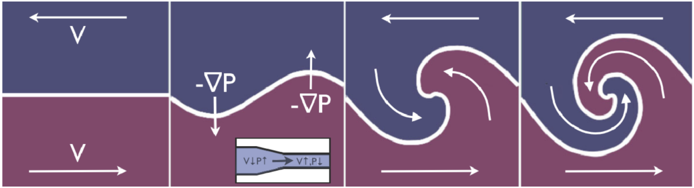
- Declare a Manager object
- Initial particles positions, velocities and entropies
- Integration for certain time domain

```
Manager<1> manager(myparticlelayout, origin, extent, dt, h, 1.4)
    ;
for(unsigned i = 0; i < N_particles; ++i){
    double v = maxwellBoltzmann(T, mass);
    R_part_0.push_back(Vector<double, 1>(origin[0] + extent[0]*i
        /((double)N_particles)));
    v_part_0.push_back(Vector<double, 1>(v));
    m_part_0.push_back(mass);
    entropy_part_0.push_back(1.0);}

//adding initial conditions to the simulation
manager.pre_run(R_part_0, v_part_0, m_part_0, entropy_part_0);
```

Kelvin-Helmholtz Instability

- Fluid flows exerting shear forces on one another
- Interesting behaviour with viscosity
- Instability behaviour at the border between the flows

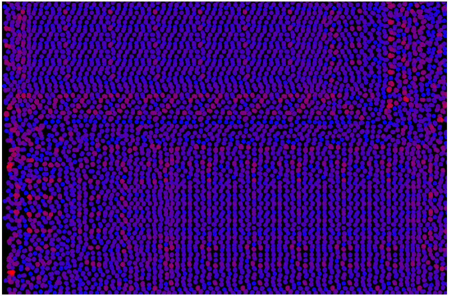


[Gilbert(2017)]

Kelvin-Helmholtz Instability

- Fluid flows exerting shear forces on one another
- Interesting behaviour with viscosity
- Simulation: (preliminary)

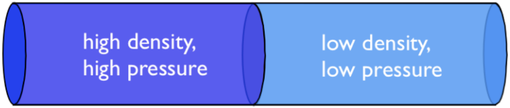
$\Longrightarrow$



$\Longleftarrow$

1D Sod-shock tube

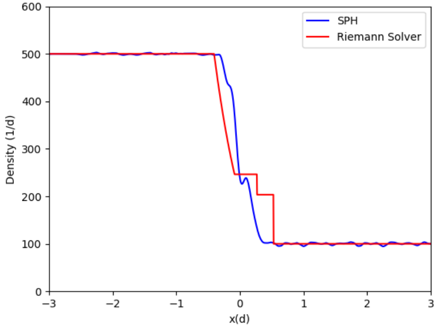
- 1D fluid $\rightarrow$ density discontinuity
- Null fluid initial velocity
- Viscous flow $\rightarrow$ shock waves

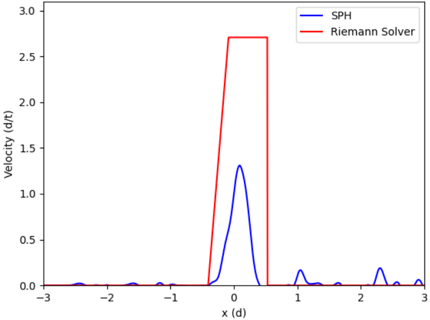


[Rosswog(2009)]

1D Sod-shock tube

- 1D fluid $\rightarrow$ density discontinuity
- Null fluid initial velocity
- Viscous flow $\rightarrow$ shock waves





Riemann solver Github

Content

Conclusions

- Implementation of SPH in C++ using IPPL
- Nearest Neighbor method
- Test cases: KH Instability and shock tube
- "Gridding" Problem
- More work! $\rightarrow$ Adaptive kernel size, add gravity, etc.

Content

[Gilbert(2017)] C. Gilbert (2017).

[Rosswog(2009)] S. Rosswog, New Astronomy Reviews **53**, 78–104 (2009).

[Springel(2010)] V. Springel, Annual Review of Astronomy and Astrophysics **48**, 391–430 (2010).

[Hockney and Eastwood(2021)] R. Hockney and J. Eastwood, *Computer Simulation Using Particles* (CRC Press, 2021).

[Cruz Pérez and Cervera(2012)] J. P. Cruz Pérez and J. A. G. Cervera, "Efficient neighborhood search in sph," in *Environmental Science and Engineering* (Springer Berlin Heidelberg, 2012) pp. 185–199.

[Muralikrishnan(2024)] S. Muralikrishnan, in *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)* (SIAM, 2024) p. 26–38.

[Monaghan and Gingold(1983)] J. J. Monaghan and R. A. Gingold, Journal of computational physics **52**, 374 (1983).

[Horwood *et al.*(2019)Horwood *et al.*] J. A. Horwood *et al.*, ResearchGate (2019).