Diving into SPH

Mustafa Bhotvawala

Navier-Stokes Equations
(Still N-S!)

The key idea is unchanged. You want to approximate the N-S equations as before, but not discretize your solution on a mesh. This is like any Lagrangian method. To recap the equations for continuum flow:

- Continuity Equation:

$$\frac{D\rho}{Dt} = -\rho \nabla \cdot \mathbf{u}$$

- Momentum Equation:

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla P + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

Smoothed Particle Hydrodynamics (SPH)

What makes it different

- SPH is mesh-free! Treatment of complex and/or moving geometries is easier
- Free surface flows that are challenging in Eulerian methods are more or less natural in SPH
- Exact (and simultaneous) conversion of mass and momentum
- Inherent parallelism

Kernels

Mathematical Form

In 3D and 4th order (also what I've used)

$$W(q) = \frac{21}{64\pi h^3} \cdot \begin{cases} (2-q)^4(1+2q) & \text{if } 0 \leq q \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where:

- Kernels just convolve over the function, and aim to be an approximation to the Dirac Delta function
- It has compact support, which means the tail is not infitine, it takes a certain range of particles around it
- A kernel function must be consistent, i.e.:

$$\int_{-\infty}^{\infty} W(q)\, dq = 1 \tag{2}$$

SPH Equations

Continuity

The approach in SPH is sto sum up contributions from neighbours.

$$\rho_a = \sum_b m_b \frac{W(\mathbf{r}_{ab}, h)}{\rho_b} \tag{3}$$

where:

- W is the smoothing kernel function.
- $\mathbf{r}_{ab}$ is the vector from particle a to b.
- h is the smoothing length.

SPH Equations

Momentum

The kernel functions can be differentiated - and this is used in the gradient for pressure in the momentum equation. Pressure is calculated from the equation of state.

$$\frac{D\mathbf{v}_a}{Dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla_a W_{ab}(h) + \nu \nabla^2 \mathbf{v}_a + \mathbf{f}_a \tag{4}$$

where:

- P_a and P_b, ρ_a and ρ_b are the pressures and densities of the current particle and its neighbours
- $\nabla_a W_{ab}(h)$ is the gradient of the smoothing kernel between particles a and b with smoothing length h.

Numerical quantities

Kernel Weighted Interpolation

- Postprocessing of SPH would involve interpolation of the same nature to obtain a numerical quantity from particle quantities
- We renormalize the interpolation using a Shepherd summation in the denominator
- For instance, the numerical pressure P_a at particle a can be computed as a weighted sum of neighbour particles

$$P_a = \frac{\sum_b P_b \frac{m_b}{\rho_b} W_{ab}}{\sum_b \frac{m_b}{\rho_b} W_{ab}}$$

A Simple Solver

Pressure Force

```
for(const auto& other : particles)
{
  if(&p != &other)
  {
    double r = distance(p, other);
    if(r > 0)
    {
      double pj = calculatePressureTait(other);
      double rhoj = other.density;

      double w_press = wendlandGradient2D(r);

      pressureForceX +=
        other.mass * (pi / (rhoi * rhoi) + pj / (rhoj * rhoj)) * w_press;
      pressureForceY +=
        other.mass * (pi / (rhoi * rhoi) + pj / (rhoj * rhoj)) * w_press;
    }
  }
}
```

A Simple Solver

Density

```
double calculateDensity(const std::vector<Particle>& particles, const
    Particle& p)
{
  double density = 0.0;
  for(const auto& other : particles)
  {
    double r = distance(p, other);
    density += other.mass * wendlandKernel2D(r);
  }
  return density;
}
```

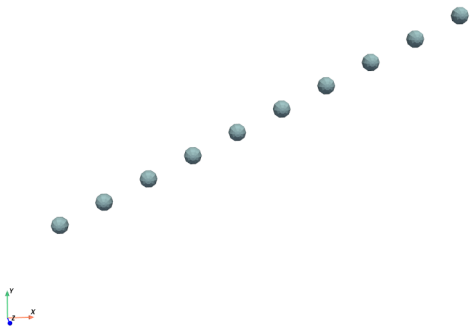
A Simple Solver

Main loop

```
for(auto& p : particles)
{
    // From https://www.cs.cmu.edu/~scoros/cs15467-s16/lectures/11-fluids2.pdf
    p.vx += dt * p.fx / p.density;
    p.vy += dt * (p.fy / p.density - g); //add gravity here
    p.x += dt * p.vx;
    p.y += dt * p.vy;
    //recalculate forces in the next step
    p.fx = 0.0;
    p.fy = 0.0;
  }
  // Calculate density and pressure forces
  for(auto& p : particles)
  {
    p.density = calculateDensity(particles, p);
  }
  calculatePressureForce(particles);
```

Just gravity

(Not much, but I'm glad I could get something basic up)



*Using PyVista for this

Dam Break

PyVista Visualization

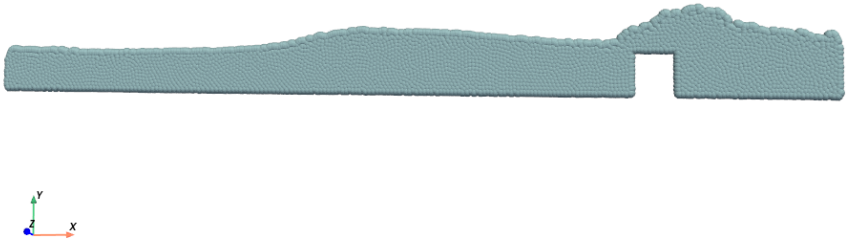


Figure: Domain at the end of the run

Dam Break

Generating CV Mesh

```
def write_to_stl():
  x_resolution = 10
  y_resolution = 10

  x = np.linspace(0.07, 0.17, x_resolution)
  y = np.linspace(0.07, 0.17, y_resolution)
  z = np.zeros(1)  # 2D box, so z dimension has only one value

  grid = pv.RectilinearGrid()
  grid.x = x
  grid.y = y
  grid.z = z

  mesh = grid.cast_to_structured_grid().extract_surface()
  mesh.save("output.stl")
  mesh.plot("show_edges=True")
```

Dam Break

Recovering Mesh in Nemo

```
def read_into_nemo(): #note I could only read back an STL, so it's going to be
    triangulated..
    mesh =  n.Mesh.load(n.File("output.stl"))
    segmentcloud = mesh.data
    areas =  mesh.get_surface_area_per_segment()
    normals = segmentcloud.get_segment_normals()

  #  print(segmentcloud.point_attributes) #-->empty dict, how to proceed?
  #  print(segmentcloud.point_attributes["face_normal"])# KeyError: 'face_normal
      '


    #Riemann sum of a unit vector
    dot_product = np.sum(normals, axis=1) #horizontally sum across normals -> 1
    riemann_sum = np.sum(dot_product * areas)

    print('Riemann sum is:')
    print(mesh.get_surface_area())
    print(riemann_sum)
```

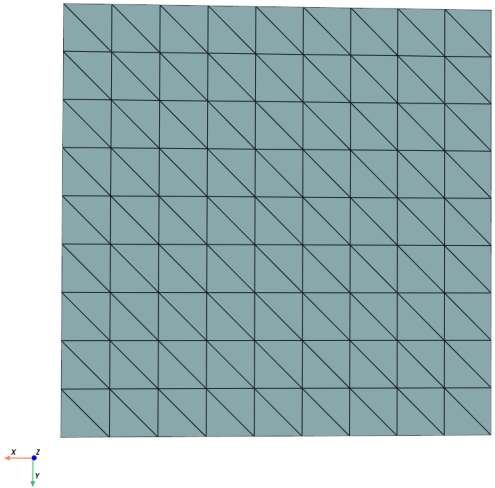


Figure: Displaying the triangulated CV

Interpolation

Pseudoalgorithm

(I think the time was beyond the scope for me to implement this, but this is what I considered. This should apply to all quantities - flux and primitives alike)

Pseudoalgorithm

1. Define a search radius = smoothing length h for each particle (not sure, but could be a start)
2. For each grid segment, look for neighbors within the radius h. Find atleast one point particle.
3. Calculate value of numerical quantity using Shephard normalization and use value for grid segment. This is essentially a projection onto the cell centroid.
4. Do Riemann sum over grid for total value on surface

Interpolation

Considerations

- Using the closest point (Voronoi) may not be the best way for consistency, instead anything that defines the local neighborhood more accurately
- A Shepherd interpolation, where the sum of the neigbourhood values is 1, can be more consistent for post-processing
- The problem with consistency might be highlighted at flow boundaries (but I need to read more about this to give you a strong opinion)
- In general, and for post-processing, the location of neighbours can be the expensive step, it probably makes sense to have something similar to a k-d tree as a pre-processing step