UNIVERSITY OF LEICESTER

Report.

For Particle Methods in Scientific Computing PA7113

Gleb Vorobchuk

Leicester

May 2018

UNIVERSITY OF
LEICESTER

Abstract.

Smoothed particle hydrodynamics or SPH is the Lagrangian method in computational physics that is used for fluid dynamics in astrophysics.

Unlike particle in cell (PIC) method SPH does not need a grid to calculate partial derivatives. Instead, they are found by analytical differentiation of interpolation formula.[]

SPH simplifies the PDE's making it easier to understand and translate astrophysics phenomena in terms of hydrodynamics. SPH work with particle-based fluid properties such as: mass, density, velocity, pressure, and acceleration to describe forces and simulate the fluid motion.

The implementation of SPH density estimator was done in C++.

Basics

In following section I want to describe some SPH basic formalism in the historical brief of SPH developing.

In Lagrangian methods fluid used to be described as set of the particles or molecules that filling some volume.

First approaches on SPH was studied by Gingold, Monaghan and Lucy in 1977. [1]

The idea was to use smoothing kernels to interpolate any function *A* at any point r:

$$A(r) = \sum_j m_j \frac{A j_j}{\rho_j} W(|r - r_j|, h)$$

Where m is the mass of particle j, A is the value of quantity A for particle j, ρ is the density around particle j, r is the position of particle W is the smoothing kernel function.

In order to find derivatives of this interplant we don't need to use finite differences method or construct a grid. Simply applying gradient.(Derivation of summation interplant affects only smoothing kernel):

$$\nabla A(r) = \sum_j m_j \frac{A j_j}{\rho_j} \nabla W(|r - r_j|, h)$$

Smoothing kernel W is the function that adjust calculations according to distances between particles. In last decades decent amount of smoothing kernels was proposed. Kernel choice depends on demotions of the particle system and property that needs to be estimated.

According to [13] appropriate smoothing kernels should:

Be normalized:

$$\int_{\Omega} W(r, h) dr = 1$$

Be positive:

$$W(r, h) dr \geq 0$$

Be even:

$$W(r, h) = W(-r, h)$$

In order to be sure that outside outside the computational range kernel nullifies any interactions suitable kernel should have a compact radius

So this implies:

[1]R.A. **Gingold**, J.J. **Monaghan**. Mon. Not. Roy. Astr. Soc., 181 (1977), p. 375.

$$W(r,h) = 0,\ ||r|| < h$$

Properties Estimation

In this section explanation on Particle System properties estimation is given.

To simulate physical processes, it is always necessary to represent the properties of the modelled system. And particle systems are no exception. Particle systems in SPH are represented by a set of particles in a given volume. Each of these particles has the following set of properties that partially determine the properties of the entire system:

1)Grid size or Volume.

2)Mass-Density.

3)Position.

4)Velocity.

In SPH particle system the only properties that remains are Grid Size and Mass of the particles.

Mass is the most important important for density estimation and pressure.

Density Estimation

Particle density is the can be estimated using following kernels.

Pressure can be used

The most used SPH smoothing kernels are:

B-spline functions:

$w(r) = Cb_n(r)$

$b_n = \frac{1}{2\pi}\int_{-\infty}^{\infty}\left(\frac{sin k/n}{k/n}\right)^n coskrdk$

Wendland functions:

$w(r) = C\psi_{l,k}(r), \psi_{l,k} = \mathscr{L}^k(1-r)_+^l$

$\mathscr{L}[f](r) = \int_r^{\infty} sf(s)ds$

Kernels used:

Qubic:

$b_4 = (1-r)_+^2 - 4(\frac{1}{2}-r)_+^2$

With Cnorm: $\frac{80}{7\pi}$,

WendC2:

$\psi_{3,1} = (1 - r)_+^4 - (1 + 4r)_+$

With Cnorm: $\frac{7}{\pi}$,

WendC6

$\psi_= (1 - r)_+^4 - (1 + 4r)_+$

With Cnorm: $\frac{78}{7\pi}$,

Pressure Estimation

Pressure is the internal force that arise within the fluid[13].

In SPH it can be calculated after estimating pressure at each particle:

$f_i^{pressure} = -\nabla p(r_i) = -\sum_{j \neq i} p_j \frac{m_j}{\rho_j} \nabla W(r_i - r_j, h)$

Since pressure depends on density, at the every particle pressure values are different. This implies that pressure is asymmetric force. SPH allows us to make it symmetric(Muller et al. (2003)):

$$f_i^{pressure} = -\sum_{j \neq i} \frac{p_i + p_j}{2} \frac{m_j}{\rho_j} \nabla W(r_i - r_j, h)$$

Which is symmetrical, stable and action-reaction law is conserved.

Where W is the spiky kernel:

$$\nabla W_{spiky}(r, h) = -\frac{30}{\pi h^4} \frac{(1 - q_{ij})}{q_{ij}} r$$

Since particles can be clustered at the high pressure areas computational model can be inaccurate as well. So we need to use smoothing kernels for pressure estimation. For this purposes we can use spiky kernel proposed by Desbrun[13].

So substituting:

$$f_i^{pressure} = \frac{15k}{\pi h^4} \sum_j m_j \frac{p_i + p_j - 2p_0}{p_j} \frac{(1 - q_{ij})^2}{q_{ij}} r$$

Viscosity Estimation

Viscosity is the measure of fluid resistance to deformation by stress. Viscosity defined by viscosity coefficient μ. In SPH we can estimate viscosity as:

$$f_i^{viscosity} = \mu \sum_{j \neq i} u_j \frac{m_j}{\rho} \nabla^2 W(r_i - r_j, h)$$

Müller proposed symmetric equation[23]:

$$f_i^{viscosity} = \mu \sum_{j \neq i} (u_j - u_i) \frac{m_j}{\rho} \nabla^2 W(r_i - r_j, h)$$

That in SPH formalism become placing density inside the Laplacian:

$$f_i^{viscosity} = \frac{\mu}{\rho_i} \sum_{j \neq i} (u_j - u_i) \frac{m_j}{\rho} \nabla^2 W(r_i - r_j, h)$$

Where W is the viscosity kernel:

$$\nabla W_{viscosity}(r, h) = \frac{40}{\pi h^4}(1 - q)$$

The Laplacian for the smoothing kernel has to be positive to prevent increasing of the velocity. Müller Kernel that take this condition into account:

$$W(\mathbf{r}, h) = \frac{1}{Ch^d} \begin{cases} f(q), & 0 \leq q \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

Gravity Estimation

Gravity is the external force that can be easily calculated after density:

$$f_i^{gravity} = p_i g$$

Smoothing kernel length

Smoothing kernel length, also known as kernel support radius is the scaling parameter that define a smoothness of the kernel function. In SPH this parameter directly affects the accuracy of the computations obtained. For small $h \to 0$ not enough particles will be involved in calculation. For greater $h \to \infty$ too many particles will involved and kernel weights near the centre particle will greater than h. For both cases results obtained will be imprecise.

Neighbour Finding Algorithm

One of the most critical part of the SPH simulation is the Neighbour Searching Algorithm(NSA).Computational complexity is rely on the number of particles. So since every SPH term iterates trough all the particles so the complexity is bounded around $O(n^2)$. Our goal is to find all the neighbours within the area bounded by radius h.

Several NSA's was developed up to date. One of the fastest are Spatial Hashing and Compact Hashing.

Spatial Hashing was proposed by Teschner M., Heidelberger B., Muller M.(2003)[2]

This method uses hash function to map every grid cell to the hash cell. Here domain is mapped to list. Infinite domain to finite list.

Hash function (for 2D grid) for position $x = (x, y)$ can be written as:

$$c = \left[\left(\left[\frac{x}{d}\right] * p_1\right) xor \left(\left[\frac{y}{d}\right] * p_2\right)\right] \mod m$$

Where d - grid cell size, m - is the hash table size, p_1 and p_2 are prime numbers 73856093, 19349663 respectively[3]

The disadvantage of the algorithm is that different grid cells can be mapped to one cell in a hash table, which inevitably leads to a collision of the hash. Hash collision leads to a algorithm slowdown. We can increase the size of the hash table in order to reduce the collision but that will leads to memory consumption increasing. Average memory consumption is around $O(n * k + m)$, where k is the number of entries.

Compact hashing on the other had store in hash cell just links to the cells that was used.

So memory for this data structure allocates only if cell contains particle. If particle is not stored in a cell then memory deallocating. During the neighbours searching used cells list is queried. The memory consumption is reduced down to $O(m + n * k), m \to n$.[4]

[2] 2, Teschner M., Heidelberger B., Müller M., Pomeranets D., Gross M.: Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proc. of Vision, Modeling, Visualization (VMV)* (2003), pp. 47–54.

[4] hmsen, M., Akinci, N., Becker, M. and Teschner, M., 2011, March. A Parallel SPH Implementation on Multi-Core CPUs. In *Computer Graphics Forum* (Vol. 30, No. 1, pp. 99-112). Blackwell Publishing Ltd.

References

1)R.A. **Gingold**, J.J. **Monaghan**. Mon. Not. Roy. Astr. Soc., 181 (1977), p. 375.

2,3) Teschner M., Heidelberger B., Müller M., Pomeranets D., Gross M.: Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proc. of Vision, Modeling, Visualization (VMV)* (2003), pp. 47–54.

4) hmsen, M., Akinci, N., Becker, M. and Teschner, M., 2011, March. A Parallel SPH Implementation on Multi-Core CPUs. In *Computer Graphics Forum* (Vol. 30, No. 1, pp. 99-112). Blackwell Publishing Ltd.

Appendix:
C++ implementation.

Snapshot.h:

```
//
// Created by Глеб Воробчук on 21/01/2018.
//

#ifndef UNTITLED_POINT_H
#define UNTITLED_POINT_H

namespace sph {
    struct Point
    {
        double x,y;
        Point&operator+=(Point const&other)
        { x+=other.x; y+=other.y; return*this; }
        Point operator+(Point const&other) const
        { auto tmp=*this; return tmp+=other; };
        Point operator-=(Point const&other)
        { x-=other.x; y-=other.y; return*this; }
        Point operator-(Point const&other) const
        { auto tmp=*this; return tmp-=other; };
        Point&operator*=(double fac)
        { x*=fac; y*=fac; return*this; }
        Point operator*(double fac) const
        { auto tmp=*this; return tmp*=fac;}
        double operator*(Point const&other) const
        { return x*other.x + y*other.y; }
        friend double norm(Point const&p)
        { return p.x*p.x + p.y*p.y; }
    };
    inline Point operator*(double fac, const Point& p)
    { return p*fac; }

}
#endif //UNTITLED_POINT_H
```

```
//
// Created by Глеб Воробчук on 21/01/2018.
//

#ifndef UNTITLED_SNAPSHOT_H
#define UNTITLED_SNAPSHOT_H

#include <vector>
#include <fstream>
#include "Point.h"

namespace sph {
//
/// class to hold all particles and the time
```

Point.h

Grid.h

```
//
// Created by Глеб Воробчук on 12/02/2018.
//

#ifndef UNTITLED_GRID_H
#define UNTITLED_GRID_H

#include <vector>
#include <cassert>
#include <random>
#include "Point.h"
#include "Snapshot.h"
#include "kernel.h"
#include <cmath>
#include <iostream>
#include <vector>

using namespace sph;
using std::vector;

class Grid {
public:

    /// creates a snapshot with positions on or near centroids of a
square grid
/// \param[in] Nside                   # particle per dimension
/// \param[in] Side                    length of grid per dimension
/// \param[in] sigma_over_delta        add random component with
stdev=sigma_over_delta * grid distance
    Snapshot make_square_grid(size_t Nside, double Side = 1, double
sigma_over_delta = 0) {
        assert(Nside);
        assert(Side > 0);
        Snapshot shot(Nside * Nside);
        const auto delta = Side / Nside;
        double m = (double)1/(double)(Nside*Nside);
        auto p = shot.begin();
        for (size_t ix = 0; ix != Nside; ++ix) {
            const auto x = (0.5 + ix) * delta;
            for (size_t iy = 0; iy != Nside; ++iy, ++p) {
                p.pos() = Point{x, (0.5 + iy) * delta};
                p.vel() = Point{0.015625, 0.015625};
                p.mass() = m;
                p.density()=0.0;
            }
        }
        if (sigma_over_delta > 0) {
            std::random_device rdv;
            std::default_random_engine R(rdv());
            std::normal_distribution<double> Norm(0., sigma_over_delta *
delta);
            for (p = shot.begin(); p != shot.end(); ++p) {
                p.pos() += Point{Norm(R), Norm(R)};
                while (pos(p).x < 0) p.pos().x += Side;
                while (pos(p).y < 0) p.pos().y += Side;
                while (pos(p).x > Side) p.pos().x -= Side;
                while (pos(p).y > Side) p.pos().y -= Side;
            }
        }
// float Snapshot::kernel(std::vector<float > x, float h);
        // read time and particle data; old data are lost
        void read(std::istream &);

        void plot(std::ostream &) const;
        void plot_density(std::ostream &,const double &H) const;


    };

    using Particle = Snapshot::Particle;

}
#endif //UNTITLED_SNAPSHOT_H
```

Kernel.h

Snapshot.cpp

Main.cpp

UNIVERSITY OF LEICESTER

```
#include <iostream>
#include <fstream>
#include <random>
#include <array>
#include "Snapshot.h"
#include "Grid.h"

//#include "Solver.h"
using namespace std;
using namespace sph;

int main() {
    //properties of particle
    size_t N=1024;
    ofstream out;//cubic
    ofstream out2;//wendC2
    ofstream out3;//quadric
    ifstream in;
    ofstream plot;
    ofstream plot_density_cubic;
    ofstream plot_density_wc2;

    //properties of the grid
    size_t size=32;

    out.open ("particles.txt");
    out2.open ("particles_wc2.txt");
    out3.open("particles_wc6.txt");
    plot.open("plot.txt");

   // plot_density_cubic.open("cubic_density.txt");
    //plot_density_wc2.open("wc2_density.txt");

    sph::Snapshot snap(N);


    Grid init_grid;

    auto sq_grid = init_grid.make_square_grid(size);
    auto h_step = Grid::linspace(0.,0.5,40); //h less than half of the
grid
    sq_grid.plot(plot);

    for(auto h=h_step.begin()+1; h != h_step.end(); ++h){
        init_grid.density_estimator_cubic(sq_grid,*h);
       //v sq_grid.plot_density(plot_density_cubic,*h);
        sq_grid.write(out,*h);
        init_grid.density_estimator_wendC2(sq_grid,*h);
       // sq_grid.plot_density(plot_density_wc2,*h);
        sq_grid.write(out2,*h);
        init_grid.density_estimator_wendC6(sq_grid,*h);
       // sq_grid.plot_density(plot_density_wc2,*h);
        sq_grid.write(out3,*h);
    }


    return 0;
}
```

```
return shot;
}

Snapshot make_random_grid(size_t Nside, double Side = 1, double
sigma_over_delta = 0) {
    assert(Nside);
    assert(Side > 0);
    const auto delta = Side / Nside;
    std::random_device rdv;
    std::default_random_engine R(rdv());
    std::normal_distribution<double> Norm2(-0.25, 0.25);
    Snapshot shot(Nside * Nside);
    double m = (double)1/(double)(Nside*Nside);
    auto p = shot.begin();
    for (size_t ix = 0; ix != Nside; ++ix) {
        for (size_t iy = 0; iy != Nside; ++iy, ++p) {
            p.pos() = Point{(0.5 + ix+Norm2(R)) * delta, (0.5 +
iy+Norm2(R)) * delta};
            p.vel() = Point{0.015625, 0.015625};
            p.mass() = m;
            p.density()=0.0;
        }
    }
    if (sigma_over_delta > 0) {
        std::random_device rdv;
        std::default_random_engine R(rdv());
        std::normal_distribution<double> Norm(0., sigma_over_delta *
delta);
        for (p = shot.begin(); p != shot.end(); ++p) {
            p.pos() += Point{Norm(R), Norm(R)};
            while (pos(p).x < 0) p.pos().x += Side;
            while (pos(p).y < 0) p.pos().y += Side;
            while (pos(p).x > Side) p.pos().x -= Side;
            while (pos(p).y > Side) p.pos().y -= Side;
        }
    }
    return shot;
}
/// periodic box [0,X] x [0,Y]
struct PeriodicBox {
    const Point P, Ph;
    explicit PeriodicBox(Point const&p)
    : P(p), Ph(0.5*P)
    { assert(P.x>0 && P.y>0); }
    void ensure_periodic(Point&p) const
    {
        while(p.x < 0. ) p.x+=P.x;
        while(p.x > P.x) p.x-=P.x;
        while(p.y < 0. ) p.y+=P.y;
        while(p.y > P.y) p.y-=P.y;
    }
    /// distance vector, shifted to [-X/2,X/2] x [-Y/2,Y/2]
    Point distance(Point const&p, Point const&q) const {
        auto d = p - q;
        //std::cout<<"D:"<<d.x<<" "<<d.y<<std::endl;
        if (d.x < -Ph.x) d.x += P.x;
        if (d.x > Ph.x) d.x -= P.x;
        if (d.y < -Ph.y) d.y += P.y;
        if (d.y > Ph.y) d.y -= P.y;
       // std::cout<<"D:"<<d.x<<" "<<d.y<<std::endl;
        return d;
    }
};
```

```
void density_estimator_cubic(Snapshot&snap,const double &H, Point
const&period={1.,1.})
{
    const auto H_sq= H*H;
    const auto iH_sq = 1/H_sq;
    const PeriodicBox pBox(period);
    for(auto pi=snap.begin();pi!=snap.end();++pi) {
        double rho = 0;
        const auto Xi = pos(pi);
        //std::cout<<"Xi: "<<Xi.x<<" "<<Xi.y<<std::endl;
        for (auto pj = snap.begin(); pj != snap.end(); ++pj) {
            auto d = pBox.distance(Xi,pos(pj));//check this
            auto d_sq = norm(d); //here is the error; return 0; problem
with periodic box;
           // std::cout<<d_sq<<std::endl;
            if (d_sq < H_sq) {

                rho += mass(pj) *
kern<kernel::cubic>::W(sqrt(d_sq*iH_sq));
            }
        }
        pi.density() = iH_sq * rho * kern<kernel::cubic>::Cnorm();
    }
}

void density_estimator_wendC2(Snapshot&snap, const double &H, Point
const&period={1.,1.})
{
    const auto H_sq= H*H;
    const auto iH_sq = 1/H_sq;
    const PeriodicBox pBox(period);
    for(auto pi=snap.begin();pi!=snap.end();++pi) {
        double rho = 0;
        const auto Xi = pos(pi);
        //std::cout<<"Xi: "<<Xi.x<<" "<<Xi.y<<std::endl;
        for (auto pj = snap.begin(); pj != snap.end(); ++pj) {
            auto d = pBox.distance(Xi,pos(pj));//check this
            auto d_sq = norm(d); //here is the error; return 0; problem
with periodic box;
            //std::cout<<d_sq<<std::endl;
            if (d_sq < H_sq) {
                rho += mass(pj) *
kern<kernel::wendC2>::W(sqrt(d_sq*iH_sq));
            }
        }
        pi.density() = iH_sq * rho * kern<kernel::wendC2>::Cnorm();
    }
}
```

```
void density_estimator_wendC6(Snapshot&snap, const double &H, Point
const&period={1.,1.})
    {
        const auto H_sq= H*H;
        const auto iH_sq = 1/H_sq;
        const PeriodicBox pBox(period);
        for(auto pi=snap.begin();pi!=snap.end();++pi) {
            double rho = 0;
            const auto Xi = pos(pi);
            //std::cout<<"Xi: "<<Xi.x<<" "<<Xi.y<<std::endl;
            for (auto pj = snap.begin(); pj != snap.end(); ++pj) {
                auto d = pBox.distance(Xi,pos(pj));//check this
                auto d_sq = norm(d); //here is the error; return 0;
problem with periodic box;
                //std::cout<<d_sq<<std::endl;
                if (d_sq < H_sq) {
                    rho += mass(pj) *
kern<kernel::wendC6>::W(sqrt(d_sq*iH_sq));
                }
            }
            pi.density() = iH_sq * rho * kern<kernel::wendC6>::Cnorm();
        }
    }
    void pressure_estimator_spiky(Snapshot&snap, const double &H, Point
const&period={1.,1.})
    {
        const auto H_sq= H*H;
        const auto H_4= H*H*H*H;

        //const auto iH_sq = 1/H_sq;
        const auto iH_4 = 1/H_4;

        const PeriodicBox pBox(period);
        for(auto pi=snap.begin();pi!=snap.end();++pi) {
            double pressure = pi.pressure();
            double rho_i = pi.density();
            const auto Xi = pos(pi);
            //std::cout<<"Xi: "<<Xi.x<<" "<<Xi.y<<std::endl;
            for (auto pj = snap.begin(); pj != snap.end(); ++pj) {
                double rho_j = pi.density();
                auto d = pBox.distance(Xi,pos(pj));//check this
                auto d_sq = norm(d); //here is the error; return 0;
problem with periodic box;
                //std::cout<<d_sq<<std::endl;
                if (d_sq < H_sq) {
                    pressure+=mass(pj) * ((rho_i + rho_j)/2/
rho_j)*kern<kernel::spiky>::W(sqrt(d_sq*iH_4));
                }
            }
            pi.pressure() = iH_4 * pressure *
kern<kernel::spiky>::Cnorm();
        }
    }
    template <typename T>
    static std::vector<T> linspace(T a, T b, size_t N) {
        T h = (b - a) / static_cast<T>(N-1);
        std::vector<T> xs(N);
        typename std::vector<T>::iterator x;
        T val;
        for (x = xs.begin(), val = a; x != xs.end(); ++x, val += h)
            *x = val;
        return xs;
    }
};
#endif //UNTITLED_GRID_H
```

```
template<>
struct kern<kernel::viscosity>
{
    static double Cnorm() { return Pi/40; }
    static double W(double q) // assumes q <= 1
    {
        return square(q)/4-cube(q)/9-1/6*log(q)-5/36;
    }
};
template<>
struct kern<kernel::spiky>
{
    static double Cnorm() { return -30/Pi; }
    static double W(double q) // assumes q <= 1
    {
        return square(q)/4-cube(q)/9-1/6*log(q)-5/36;
    }
};
template<>
struct kern<kernel::viscosity>
{
    static double Cnorm() { return 40/Pi; }
    static double W(double q) // assumes q <= 1
    {
        return (1-q);
    }
};
```

```
double i=0;
   while (std::getline(input, line))
   {

       if (line.find(part) != std::string::npos) {
           std::istringstream iss(line);
           iss.ignore(12);
           iss >> i;//get the mass
           i=i-1;
       }
       if (line.find(mass) != std::string::npos) {
           double m;
           std::istringstream iss(line);
           iss.ignore(7);
           iss >> m;//get the mass
           this->M[i]=m;//set mass

       }
       if (line.find(pos) != std::string::npos) {
           double X,Y;
           input.ignore();//get line with X
           std::getline(input,line);
           std::istringstream iss(line);
           iss.ignore(7);
           iss >> X;
           input.ignore();//get line with Y
           std::getline(input,line);
           std::istringstream iss2(line);
           iss2.ignore(7);
           iss2 >> Y;
           this->X[i].x=X;
           this->X[i].y=Y;
       }
       if (line.find(vel) != std::string::npos) {
           double VX,VY;
           input.ignore();//get line with X
           std::getline(input,line);
           std::istringstream iss(line);
           iss.ignore(7);
           iss >> VX;
           input.ignore();//get line with Y
           std::getline(input,line);
           std::istringstream iss2(line);
           iss2.ignore(7);
           iss2 >> VY;
           this->V[i].x=VX;
           this->V[i].y=VY;
       }
       if (line.find(dens) != std::string::npos) {
           double density;
           std::istringstream iss(line);
           iss.ignore(7);
           iss >> density;//get the density
           this->RHO[i]=density;//set density
       }
   }
```

```
void Snapshot::plot(std::ostream & plot) const {
    if (!plot) {
        std::cerr << "error: open file for input failed!";
        abort();
    }

    for (auto i : this->X) {
        plot<< i.x<<"\t"<< i.y<<"\n";
    }


}

void Snapshot::plot_density(std::ostream & plot, const double &H) const
{
    if (!plot) {
        std::cerr << "error: open file for input failed!";
        abort();
    }

    plot<<"h:\t"<<H<<"\n";
    for (auto i : this->RHO) {
        plot<<i<<"\n";
    }


}
```