PARTICLE STRENGTH EXCHANGE METHOD

Project Report
Prerna Patil

May 2016

Contents

1 Introduction to particle methods

Advantages of using vortex methods:
1. Vortex Methods have no numerically dissipative truncation errors.Thus vortex methods are better for computations at higher Reynolds numbers. Grid based schemes suffer from numerical dissipation. In Lagrangian frame the vortex particles can be convected without introducing this dissipation.
2. Δt can be high.
3. Pressure drops out of the governing equations and thus only needs to be solved for when the requirement for force calculation arises.

2 Particle Strength Exchange Method

This method for infinite domain was introduced by Raviart(1987) and his co-workers (Cottet (1987), Huberson(1987), Mas Gallic (1987)). [1] This method in general simulates the diffusion process accurately.

Characteristices of Particle Strength Exchange Method:
(a) Based on the exchange of circulation among particles to approximate diffusion
(b) Involves approximating the Laplacian at a particle's location based on nearby particles
(c) Formulated grid-free but requires frequent remeshing of the particle field onto a well-ordered field [2]

3 Formulation

The particle strength exchange method is formulated in terms of vorticity. We carry out the spatial discretization of vorticity instead of velocity. The Curl eliminates the gradient of pressure and we don't have to deal with pressure unless required for force calculations. For sake of simplicity incompressible flow is considered for calculations and simulations in this project.

Continuity Equation:

$$\nabla \cdot u = 0 \tag{1}$$

$$\nabla \times u = \omega \tag{2}$$

where ω is vorticity.
The velocity field can be obtained from the Poisson's Equation.

$$\nabla \times \omega = \nabla \times (\nabla \times u) = \nabla(\nabla \cdot u) - \nabla^2 u = -\Delta u \tag{3}$$

Momentum Equation:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\frac{\partial p}{\partial x} + \nu \nabla^2 u \tag{4}$$

Taking curl of the equation:

$$\frac{\partial \omega}{\partial t} + (u \cdot \nabla)\omega = \nu \nabla^2 \omega \tag{5}$$

which can be concisely written as:

$$\frac{D\omega}{Dt} = \nu \nabla^2 \omega \tag{6}$$

The solution to the diffusion equation can be written by using the Green's function approach:

$$\omega(x,t) = \int G(x-y, \nu t)\omega_o(y)dy \tag{7}$$

The discretized form can be written as:

$$\omega_p^{n+1} = \sum_{q=1}^{N} v_q \omega_q^n G[x_p^n - x_q^n, \nu \delta t] \tag{8}$$

$$\omega_p^{n+1} = \omega_p^n + \sum_{q=1}^{N} v_p(\omega_q^n - \omega_p^n) G[x_p^n - x_q^n, \nu \delta t] \tag{9}$$

In 1990, Pepin observed that when Gaussian regularization function is used with time step $\delta t = \frac{\epsilon^2}{4\nu}$ the results are independent of the particles used to resolve the equation.

$$G(x-y, \epsilon) = \frac{1}{4\pi\epsilon} e^{\frac{-|x-y|^2}{4\epsilon}} \tag{10}$$

Convection equation:

$$\frac{dx_i}{dt} = u_i \tag{11}$$

Using the Adam Bashforth Second order scheme for convection:

$$x_{n+1} = x_n + \Delta t(1.5u_n - 0.5u_{n-1}) \tag{12}$$

The solution for velocities is obtained by using the **Biot-Savart law** for induced velocity at a point:

$$V(z) = u - iv = \frac{i}{2\pi} \sum_{m=1}^{N} \frac{-\Gamma_m}{z - z_m} \tag{13}$$

where $z = x + iy$

4 Problem Definition

Problem 1:

Vortex Diffusion: Considering a Gaussian vortex field with core size $\sigma = 1.0$ centred at the origin. The diffusion is studied for T= 2 Time Units.

Problem 2:

Vortex Interaction: Considering two Gaussian vortices placed in a flow field. The circulation strengths of both the vortices is same. To study the effects of vortex merging.

5 Code Algorithm

Table 1: Algorithm for PSE

	Particle Strength Exchange Method
1. Initialize the domain	$\omega_0 \rightarrow \Gamma_0$
2. Remesh	Remeshing can be done at every step or after 5-10 steps
3. Solve Diffusion and Continuity	(1) We have the value of circulation Γ^{n+1} from the formulation (2) We have values of (u,v) from the Biot-Savart law
4. Convect and diffuse the particles	Adam-Bashforth 2nd Order scheme (other higher order schemes can be used)
5. Return to (2)	while $(t < T)$

6 Remeshing

Most vortex methods consist of restarting the particle field on a regular grid every few time steps and re-calculating particle strengths based on the diffused particle field by using interpolation methods. (This is an active area of research in vortex methods). The need for remeshing arises due to the fact that Lagrangian time evolution results in the loss of discretisation accuracy due to distortion of particle distribution. Particle distortion can result in the creation and evolution of spurious vortical structures due to the inaccurate resolution of areas of high shear.To control these inaccuracies introduced in the flow we introduce remeshing by using higher order interpolation schemes. This evokes a controversy regarding the meshlessness of the method.
Although we will see during the error calculation about the interpolation errors introduced in the flow field due to remeshing these errors are lower compared to the loss of discretisation accuracy introduced if we do not remesh. This also makes the simulations computationally expensive by $O(N^2)$ since we need to interpolate the values of circulation on the nearby points. To reduce the computational cost we can remesh particles after 5-10 time steps. For the computations carried out in this project, remeshing is employed at every time step.

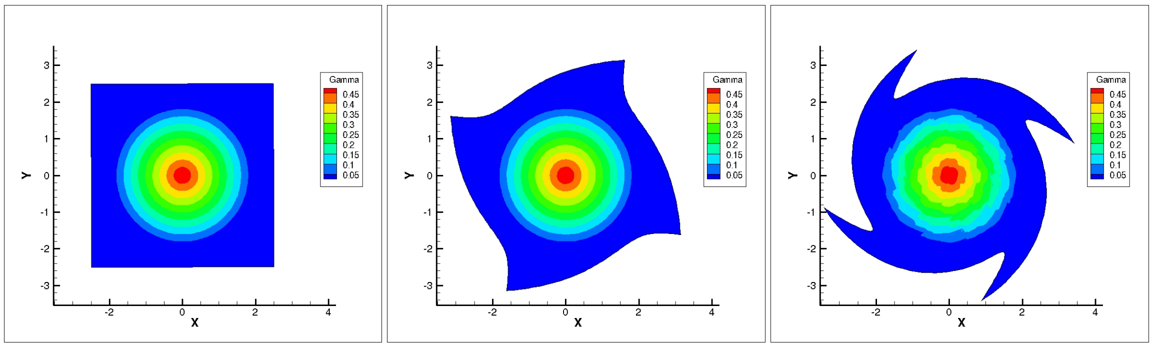
The need for remeshing

Following is an example of simulation to demonstrate the need for remeshing. The problem is taken from ”Simulations using particles: HPCSE I Class notes” [Koumoutsakos, 2005]
We consider vorticity equation without viscosity ($\nu = 0$). The vorticity evolves according to Euler eqaution: $\frac{D\omega}{Dt} = 0$. The initial flow field is set to radial function given by:

$$\omega_o(x) = \omega_{max} \cdot max(0, 1 - \frac{||x||}{R}) \tag{14}$$

Simulation details: For the purpose of simulation the following are used:
$R = 2.0$, $\omega_{max} = 10$ $\delta t = 0.0001$
No-remeshing: Particles used - 4900
Remeshing: Particles used - 625

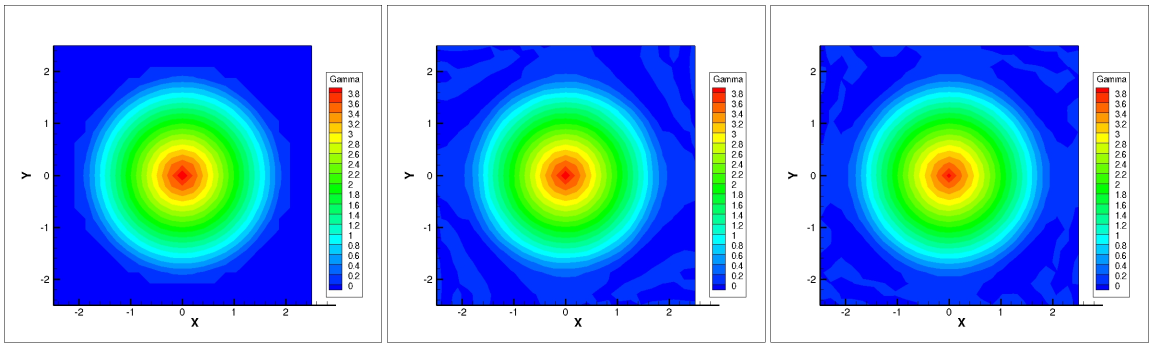
Since the vorticity is radially symmetric and system is in steady state the solution is just the initial condition for all time steps.



(a) Γ at T=0.0 (b) Γ at T=0.06 (c) Γ at T=0.2

Figure 1: Flow Field Evolution without remeshing

We see that without remeshing the radial symmetry breaks due to structures generated by the solver. These inaccuracies might build over time and give inaccurate numerical solutions. Now we see the results with remeshing



(a) Γ at T=0.0 (b) Γ at T=0.06 (c) Γ at T=0.2

Figure 2: Flow Field Evolution with remeshing: We see that the radial symmtery is preserved if remeshing is employed

Interpolation method for remeshing:

This method consists of interpolating the diffused particle field on Cartesian lattice by using interpolation kernals. For 2D or 3D interpolation kernals are built by taking a Cartesian tensor product of the 1D kernals. The commonly used interpolation kernals are given below:

$$\Lambda_o(u) = \begin{cases} 1, & \text{if } 0 \leq u < 0.5. \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

$$\Lambda_1(u) = \begin{cases} 1-|u|, & \text{if } 0 \leq u < 1. \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

$$M_4(u) = \begin{cases} \frac{1}{6}(2-u)^3 - \frac{4}{6}(1-u)^3, & \text{if } 0 \leq u \leq 1. \\ \frac{1}{6}(2-u)^3, & \text{if } 1 \leq u \leq 2. \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

$$M_4'(u) = \begin{cases} 1 - \frac{5u^2}{2} + \frac{3u^3}{2}, & \text{if } 0 \leq u \leq 1. \\ \frac{1}{2}(2-u)^3(1-u), & \text{if } 1 \leq u \leq 2. \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

The interpolation family of "M" is derived from the splines. The M_4' kernal is third order accurate kernal. The M_4' kernal conserves the first three invariants (total Circulation, linear impulse and angular impulse).This kernal is used by researchers for higher accuracy results (Lorena A. Barba, 2004) It has the advantage of being smooth and more accurate than the Λ_2 kernal which is of the same order.
For the numerical experiments carried out for this project, the M_4' kernal has been adopted.

Formulation:

$$\Delta\Gamma_{ij} = \Gamma_i M_4(\frac{\tilde{x_j} - x_i}{h}) M_4'(\frac{\tilde{y_j} - y_i}{h}) \tag{19}$$

Here $(\tilde{x_j}, \tilde{y_j})$ is the point on the new ordered Cartesian mesh and (x_i, y_i) are the old diffused particles whose circulation we need to map on the cartesian grid points.
$\Delta\Gamma_{ij}$ is the contribution to circulation from the i^{th} vortex (old) to the new mesh point $(\tilde{x_j}, \tilde{y_j})$.

Time stepping after remeshing: In the simualtions carried out for this project Adam-Bashforth Scheme was employed for time-stepping in convection equation. Remeshing creates an issue for availibility of velocities for the particles. To solve this issue two methods can be used:
1.) We can remesh the values of the velocites on the ordered grid as well. However this will violate the

relation $\omega = \nabla \times U$ for the remeshed vorticity field.
2.) Second method is to recalulate the velocity field from the Biot-Savart law on all the grid points. This however increases the computational expense by $O(N^2)$.

7 Results

Vortex Diffusion:
Simulation details: Vortex field at T=0.0:

$$\omega_o(x, y, t = 0) = \omega_{max} \cdot e^{\frac{-(x^2+y^2)}{2}} \tag{20}$$

Parameters: $\delta t = 0.001$, $\nu = 0.1$, $T_{final} = 2$, N=1600 (Total particles)
Analytical Solution:
Using separation of variables we get:

$$\frac{1}{T}\frac{\partial T}{\partial t} = -\lambda^2 \qquad and \qquad (\frac{1}{r}\frac{\partial R}{\partial r} + \frac{\partial^2 R}{\partial r^2}) = \frac{-\lambda^2 R}{\nu} \tag{21}$$

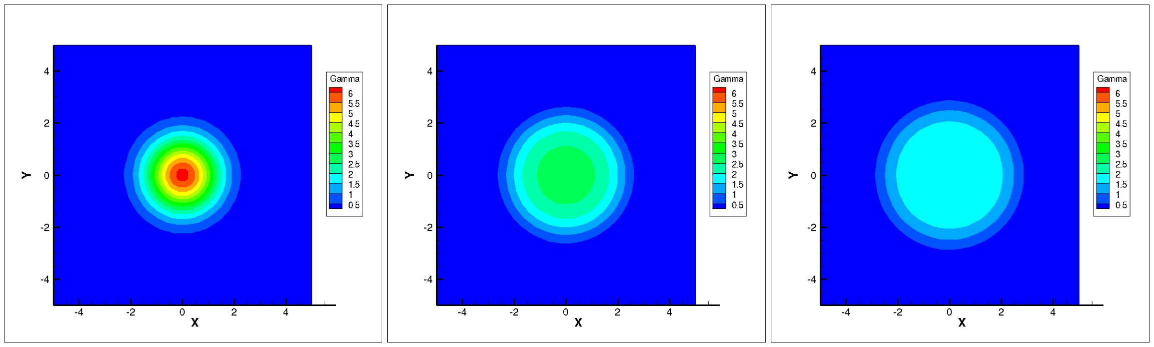
Thus, $T(t) = e^{-\lambda^2 t}$ whereas we see that the second separated equation is in the Bessel's differential equation form $[x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0]$ whose solution is given by the bessels's functions of the first kind. (In this case $\alpha = 0$)

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\,\Gamma(m+\alpha+1)}(\frac{x}{2})^{2m+\alpha} \tag{22}$$

For the zeros we require $J_o(\frac{\lambda_n r_o}{\sqrt{\nu}}) = 0$ To simplyfy the simulations we take $r_o = 10$
The first few roots are given by: $\frac{\lambda_1 r_o}{\sqrt{\nu}} = 2.405$,$\frac{\lambda_2 r_o}{\sqrt{\nu}} = 5.520$, $\frac{\lambda_3 r_o}{\sqrt{\nu}} = 8.654$, $\frac{\lambda_4 r_o}{\sqrt{\nu}} = 11.79$
The particular solution is given by:

$$T_n(r, t) = R_n(r)T(t) = \sum_{n=1}^{\infty} A_n J_o(\frac{\lambda_n r_o}{\sqrt{\nu}})e^{-\lambda_n^2 t} \tag{23}$$

$$A_n = \frac{2}{\frac{r_o^2}{\nu}J_1^2(\frac{\lambda_n r_o}{\sqrt{\nu}})}\int_0^{r_o/\nu} rJ_o(\lambda_n r)\omega(r, t = 0)dr \tag{24}$$



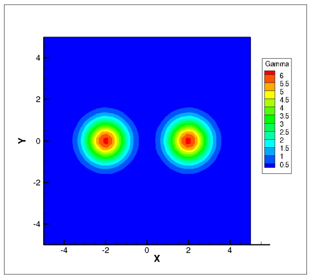
(a) Γ at T=0 (b) Γ at T=1 (c) Γ at T=2

Figure 3: Diffusion of Gaussian Vortex

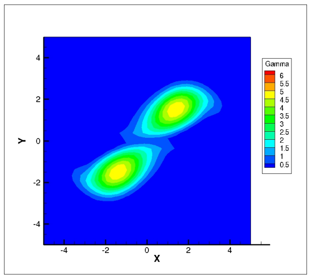
Vortex Merging: We place two gaussian vortices in the flow field at a distance of $L = 4\sigma$ where σ is the variance of the gaussian distribution.
Vortex field at T=0.0:

$$\omega_o(x, y, t = 0) = \omega_{max} \cdot e^{\frac{-((x-2\sigma)^2+y^2)}{2}} + \omega_{max} \cdot e^{\frac{-((x+2\sigma)^2+y^2)}{2}} \tag{25}$$

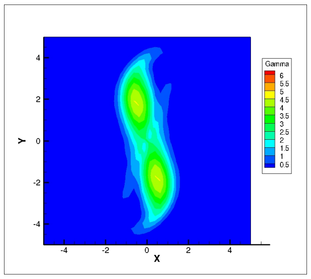
Evolution of the vorticity field:



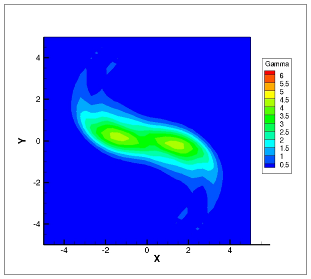
(a) Γ at T=0



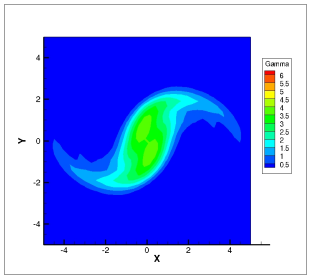
(b) Γ at T=0.15



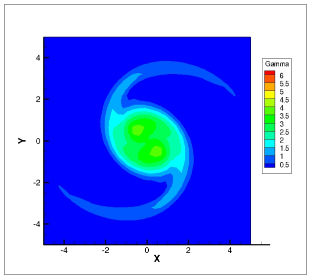
(c) Γ at T=0.30



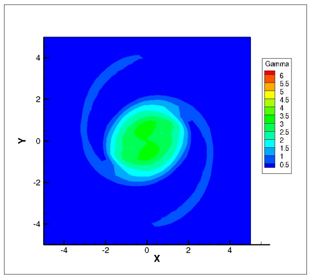
(d) Γ at T=0.45



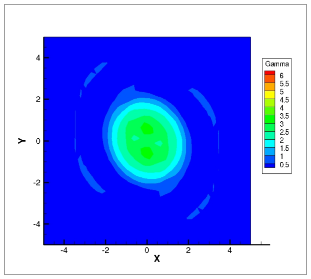
(e) Γ at T=0.60



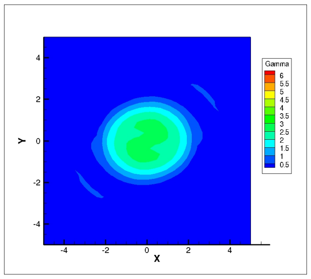
(f) Γ at T=0.75



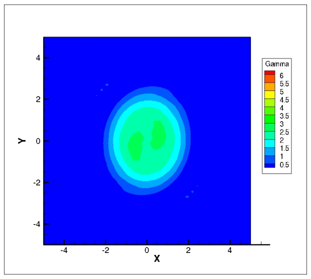
(g) Γ at T=0.90



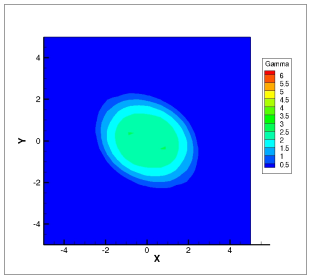
(h) Γ at T=1.05



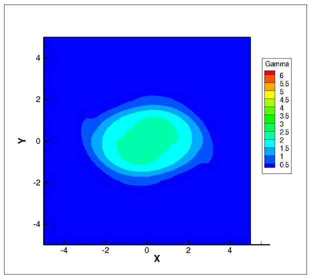
(i) Γ at T=1.20



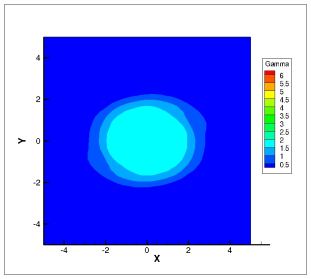
(j) Γ at T=1.35



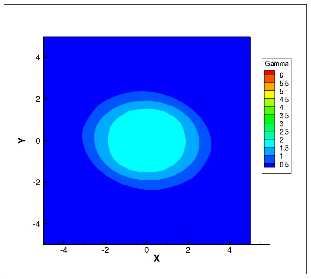
(k) Γ at T=1.50



(l) Γ at T=2.00



(m) Γ at T=2.50



(n) Γ at T=3.00

Figure 4: Evolution of vorticity field

8 Error Estimation

Truncation error due to Adam-Bashforth 2nd order: Adam Bashforth is second order accurate.

$$x_{n+1} = x_n + \Delta t(1.5u_n - 0.5u_{n-1}) \tag{26}$$

$$TruncationError = \frac{5}{12}\frac{d^3x}{dt^3}\delta t^2 \tag{27}$$

Interpolation errors: Order of accuracy of the interpolation fuunction is $O(h^2)$

Approximation Error: The approximation of the vorticity function introduces an error. This error depends on the smoothing function.

$$\omega(x,t) = \int G(x-y, \nu t)\omega_o(y)dy \tag{28}$$

$$G(x-y, \sigma^2) = \frac{1}{4\pi\sigma^2}e^{\frac{-|x-y|^2}{4\sigma^2}} \tag{29}$$

The gaussian kernal is accurate to $O(\sigma^2)$

Table 2: Error Calculation (For steady case): T=0.001

Particles	Error
25x25	0.00575403
40x40	0.00192992
50x50	0.00115641
60x60	0.000693931
70x70	0.0005064
80x80	0.000352878
90x90	0.000263211

9 Disadvantages of the Particle Strength Method:

1.) Computational cost.
2.) Distortion of the associated Lagrangian grid and the need for remeshing after every time step which introduces a computational cost of $O(N^2)$

10 Codes Used

C++ CODE VORTEX DIFFUSION:

```
//Code to solve the flow field vortex and compare the results to analytical solution:
//Written by: Prerna Patil
//Last edited: 12th May 2016
#define _USE_MATH_DEFINES

#include<cmath>
#include<stdio.h>
#include<iostream>
#include<math.h>
#include<iomanip>
#include "M4.h"
#include<fstream>
#include<iomanip>
#include<string.h>

using namespace std;
/**********************CLASS PARTICLE*************************************/
//Creating class Particle whcih stores information about X, Y, vorticity, u, v :
class particle
{
double X, Y, vorticity, U, V, Gamma; //Gamma= Circulation

public:
//Default constructor:
particle()
{
X=0.0; Y=0.0; vorticity =0.0; U=0.0; V=0.0; Gamma=0.0;
}
void set_position(double, double);
void set_velocity(double, double);
void set_vorticity(double);
void set_Gamma(double);
double getX();
double getY();
double getU();
double getV();
double getvor();
double getGamma();

//Operator overloading:
void operator=(const particle& par)
{

this->X = par.X;
this->Y = par.Y;
this->U = par.U;
this->V = par.V;
this->vorticity = par.vorticity;
this->Gamma = par.Gamma;
}
```

```
};
double particle::getX(void)
{
return X;
}
double particle::getY(void)
{
return Y;
}
double particle::getU(void)
{
return U;
}
double particle::getV(void)
{
return V;
}
double particle::getvor(void)
{
return vorticity;
}
double particle::getGamma(void)
{
return Gamma;
}
void particle::set_position(double xin, double yin)
{
X= xin;
Y= yin;
}
void particle::set_velocity(double uin, double vin)
{
U= uin;
V= vin;
}
void particle::set_vorticity(double vor)
{
vorticity =vor;
}
void particle::set_Gamma(double gam)
{
Gamma= gam;
}
/********************** CLASS PARTICLE END *************************************/
int main()
{
//Initialisation of the domain:
//Initialisaing a Gaussian vortex field
//\Sigma = 1 (centred at the origin)
//Infinite domain??
// x E [-3, 3] y E [-3, 3]
double xstart = -5.0, xend = 5.0;
double temp;
double dt= 0.001, t=0.0, nu =0.1, epsilon = nu*dt; //nu= viscosity
double T = 2.0; //Final time
/************************** INITIALIZATION *************************************/
//Eulerian mesh
int N=40;
double h = (xend - xstart) / static_cast<double>(N-1);
double x[N], y[N];
//Volume remains unchanged for all time steps(incompressible flow)
double Vol = pow((xstart - xend),2.0)/(N*N);
for(int i=0; i<N; i++)
{
x[i] = xstart + i*h;
y[i] = x[i] ;
//cout<<"x: "<<x[i] <<endl;
}
clock_t begin, end;
double time_spent;
begin = clock();

//Initialising particles on heap:
//For using Adam-Bashforth second order for time stepping
particle *par = new particle [N*N]; //Particle data at n+1 time step (current time step)
particle *par_n = new particle [N*N]; //Particle data at n time step
particle *par_n1 = new particle [N*N]; //Particle data at n-1 time step
particle *par_diffuse = new particle[N*N]; //particles field data after diffusion
/*
//Initialise the reset mesh (all quatiies zero and set position)
//This initialisation needs to be done before every reset
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
par_reset[N*i+j].set_velocity(0.0, 0.0);
par_reset[N*i+j].set_Gamma(0.0);
par_reset[N*i+j].set_vorticity(0.0);
par_reset[N*i+j].set_position(x[i], y[j]);
}
}
*/
//Initialise the field at t=0;
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
par_n1[N*i+j].set_position(x[i], y[j]);
temp= 100.0* exp(-0.5*(pow(par_n1[N*i+j].getX(),2.0)+ pow(par_n1[N*i+j].getY(),2.0)));
par_n1[N*i+j].set_vorticity(temp); //Setting the vorticity value for all the particles
temp= par_n1[N*i+j].getvor() * Vol;
par_n1[N*i+j].set_Gamma(temp);    //Setting the Circulation value for all the particles
//cout<<setw(2)<<par_n1[N*i+j].getGamma()<<"  ";
}
//cout<<endl;
}
double kernal, r2, conX, conY, Circ;
//We need to obtain the value of velocity at the grid points using the greens function:
```

```
for(int p=0; p<N*N; p++)
{
conX = 0.0, conY = 0.0;
for(int q=0; q< N*N; q++)
{
if(p==q){q=q+1;}
r2 = pow(par_n1[q].getX() - par_n1[p].getX() ,2.0) + pow(par_n1[q].getY() - par_n1[p].getY() ,2.0);
conX = conX - par_n1[q].getGamma() * (par_n1[p].getY() - par_n1[q].getY())/(2*M_PI*r2);
conY = conY + par_n1[q].getGamma() * (par_n1[p].getX() - par_n1[q].getX())/(2*M_PI*r2);
}
par_n1[p].set_velocity(conX, conY);
}
//wrting initial data to file:
ofstream outputtec;
outputtec.open("tec.dat");
outputtec<<"TITLE=\"Simple\""<<endl;
outputtec<<"VARIABLES=\"X\" \"Y\" \"U\" \"V\" \"Gamma\""<<endl;
outputtec<<"ZONE I="<<N<<" J="<<N<<endl;
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
outputtec<<setw(6)<<par_n1[N*i+j].getX()<<"  "<<setw(6)<<par_n1[N*i+j].getY()<<"  "
<<setw(6)<<par_n1[N*i+j].getU()<<"  "<<setw(6)<<par_n1[N*i+j].getV()<<"  "
<<setw(6)<<par_n1[N*i+j].getGamma()<<endl;

}
}
//Calculating the Gamma using the difussion equation: (for the first time step)
/*
//Convection by Adam bashforth second order:
conX = par_n[p].getX() + dt* (1.5*par_n[p].getU() - 0.5*par_n1.getU());
conY = par_n[p].getY() + dt* (1.5*par_n[p].getV() - 0.5*par_n1.getV());
par[p].set_position(ConX, ConY);
//Finding the velocities by MPE:
r2 = pow(par_n1[q].getX() - par_n1[p].getX() ,2) + pow(par_n1[q].getY() - par_n1[p].getY() ,2);
conX = -par[p].getGamma()*(par[q].getY() - par[p].getY())/(2*M_PI*r2);//Sum over p
conY = par[p].getGamma() * (par[q].getX() - par[p].getX())/(2*M_PI*r2);//Sum over p
par[p].set_velocity(conX, conY);
*/

for(int p=0; p<N*N; p++)
{
Circ = par_n1[p].getGamma();
conX = 0.0, conY = 0.0;
for(int q=0; q<N*N; q++)
{
if(p==q){q=q+1;}
r2 = pow(par_n1[q].getX() - par_n1[p].getX() ,2.0) + pow(par_n1[q].getY() - par_n1[p].getY() ,2.0);
kernal = 1.0/(4.0*epsilon*M_PI) * exp(-r2/(4.0*epsilon));
Circ = Circ + nu*dt * kernal * Vol*(par_n1[q].getGamma() - par_n1[p].getGamma());
conX = conX - par_n1[q].getGamma() * (par_n1[p].getY() - par_n1[q].getY())/(2.0*M_PI*r2);
conY = conY + par_n1[q].getGamma() * (par_n1[p].getX() - par_n1[q].getX())/(2.0*M_PI*r2);
}
```

```
par_diffuse[p].set_Gamma(Circ);
par_diffuse[p].set_velocity(conX, conY);

//Convection of the particles: (For the first time step: Euler forward)
conX = par_n1[p].getX() + dt*par_n1[p].getU();
conY = par_n1[p].getY() + dt*par_n1[p].getV();
par_diffuse[p].set_position(conX, conY);

}
//Remeshing:
//Remesh the diffused particles to the stable grid:
//Interpolate the values of the circulation over the nearest grid points and regularize the grid:
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
par_n[N*i+j].set_position(x[i], y[j]); //Setting the value of x and y of the new remesh grid
 //Since the first time step all values are initialised to zero by the default constructor
//par_n[N*i+j].set_velocity(0.0, 0.0);
//par_n[N*i+j].set_Gamma(0.0);
//par_n[N*i+j].set_vorticity(0.0);
temp =0.0;
for(int p=0; p<N*N; p++)
{
//Using the M4 Third order accuracy
temp = temp + par_diffuse[p].getGamma()* M4(par_n[N*i+j].getX() , par_diffuse[p].getX(), h) *
M4(par_n[N*i+j].getY() , par_diffuse[p].getY(), h);
}
par_n[N*i+j].set_Gamma(temp); //Setting the value of the circulation at the 1st time step
//cout<<setw(2)<<par_n[N*i+j].getGamma()<<"  ";
}
//cout<<endl;
}
//We need to obtain the value of velocity at the grid points using the greens function:
for(int p=0; p<N*N; p++)
{
conX = 0.0, conY = 0.0;
for(int q=0; q< N*N; q++)
{
if(p==q){q=q+1;}
r2 = pow(par_n[q].getX() - par_n[p].getX() ,2.0) + pow(par_n[q].getY() - par_n[p].getY() ,2.0);
kernal = 1.0/(4.0*epsilon*M_PI) * exp(-r2/(4.0*epsilon));
conX = conX - par_n[q].getGamma() * (par_n[p].getY() - par_n[q].getY())/(2.0*M_PI*r2);
conY = conY + par_n[q].getGamma() * (par_n[p].getX() - par_n[q].getX())/(2.0*M_PI*r2);
}
par_n[p].set_velocity(conX, conY);
}
//Starting the time stepping using Adam Bashforth Second Order:
t= dt;

while(t < T)
{
//Initialise diffuse to zero:
//Initialise the reset mesh (all quatiies zero and set position)
```

```
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
par_diffuse[N*i+j].set_velocity(0.0, 0.0);
par_diffuse[N*i+j].set_Gamma(0.0);
par_diffuse[N*i+j].set_vorticity(0.0);
par_diffuse[N*i+j].set_position(x[i], y[j]);
}
}
//Diffuse the particles according to adam bashforth second order
for(int p=0; p<N*N; p++)
{
Circ = par_n[p].getGamma();
conX = 0.0, conY = 0.0;
for(int q=0; q<N*N; q++)
{
if(p==q){q=q+1;}
r2 = pow(par_n[q].getX() - par_n[p].getX() ,2.0) + pow(par_n[q].getY() - par_n[p].getY() ,2.0);
kernal = 1.0/(4.0*epsilon*M_PI) * exp(-r2/(4.0*epsilon));
Circ = Circ + nu*dt * kernal * Vol*(par_n[q].getGamma() - par_n[p].getGamma());
conX = conX - par_n[q].getGamma() * (par_n[p].getY() - par_n[q].getY())/(2.0*M_PI*r2);
conY = conY + par_n[q].getGamma() * (par_n[p].getX() - par_n[q].getX())/(2.0*M_PI*r2);
}
par_diffuse[p].set_Gamma(Circ);
par_diffuse[p].set_velocity(conX, conY);

//Convection of the particles: (Using Adam Bashforth 2nd order)
conX = par_n[p].getX() + dt*(1.5*par_n[p].getU() - 0.5*par_n1[p].getU());
conY = par_n[p].getY() + dt*(1.5*par_n[p].getV() - 0.5*par_n1[p].getV());
par_diffuse[p].set_position(conX, conY);

//cout<<par_diffuse[p].getU()<<"  "<<endl;

}
//Remesh the particles to stable grid:
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
par[N*i+j].set_position(x[i], y[j]); //Setting the value of x and y of the new remesh grid
//Since the first time step all values are initialised to zero by the default constructor
//par[N*i+j].set_velocity(0.0, 0.0);
//par[N*i+j].set_Gamma(0.0);
//par[N*i+j].set_vorticity(0.0);
temp =0.0;
for(int p=0; p<N*N; p++)
{
//Using the M4 Third order accuracy
temp = temp + par_diffuse[p].getGamma()* M4(par[N*i+j].getX() , par_diffuse[p].getX(), h) *
M4(par[N*i+j].getY() , par_diffuse[p].getY(), h);
}
par[N*i+j].set_Gamma(temp); //Setting the value of the circulation at the 1st time step
```

```
//cout<<par[N*i+j].getGamma()<<" ";
}
//cout<<endl;
}
//cout<<"***********"<<endl;
//We need to obtain the value of velocity at the grid points using the greens function:
for(int p=0; p<N*N; p++)
{
conX = 0.0, conY = 0.0;
for(int q=0; q< N*N; q++)
{
if(p==q){q=q+1;}
r2 = pow(par[q].getX() - par[p].getX() ,2.0) + pow(par[q].getY() - par[p].getY() ,2.0);
kernal = 1.0/(4.0*epsilon*M_PI) * exp(-r2/(4.0*epsilon));
conX = conX - par[q].getGamma() * (par[p].getY() - par[q].getY())/(2.0*M_PI*r2);
conY = conY + par[q].getGamma() * (par[p].getX() - par[q].getX())/(2.0*M_PI*r2);
}
par[p].set_velocity(conX, conY);
}
//Shift the data for the particles:
for(int p=0; p<N*N; p++)
{
par_n1[p] = par_n[p];
par_n[p] = par[p];
}

t=t+dt; //Advance in the time step
cout<<"t:"<<t<<endl;
if(static_cast<int>(t/dt) %10 == 0)
{

//Write data to a file:
//ofstream outputvor, outputU, outputV, outputGamma;
//outputvor.open("Vorticity_"+ std::to_string(static_cast<int>(t/dt))+".dat");
//outputU.open("U_"+ std::to_string(static_cast<int>(t/dt))+".dat");
//outputV.open("V_"+ std::to_string(static_cast<int>(t/dt))+".dat");
//outputGamma.open("Gamma_"+ std::to_string(static_cast<int>(t/dt))+".dat");

outputtec<<"ZONE I="<<N<<" J="<<N<<endl;
for(int i=0; i<N; i++)
{
for(int j=0; j<N; j++)
{
// outputvor<<setw(6)<<par[N*i+j].getvor()<<"  ";
// outputU<<setw(6)<<par[N*i+j].getU()<<"  ";
// outputV<<setw(6)<<par[N*i+j].getV()<<"  ";
outputtec<<setw(6)<<par[N*i+j].getX()<<"  "<<setw(6)<<par[N*i+j].getY()<<"  "
<<setw(6)<<par[N*i+j].getU()<<"  "<<setw(6)<<par[N*i+j].getV()<<"  "<<setw(6)<<par[N*i+j].getGamma()<<e
// outputGamma<<setw(6)<<par[N*i+j].getGamma()<<"  ";

}
// outputvor<<endl;
// outputU<<endl;
// outputV<<endl;
// outputGamma<<endl;
}

}

}
end = clock();
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
cout<<"Timetaken:"<<time_spent<<endl;
return 0;
}
```

C++ CODE INTERPOLATION FUNCTION:

```
//Function for remeshing:
#include "M4.h"
#include <math.h>
//Function code:
double M4(double x_tilda, double x, double h )
{
double u= fabs(x_tilda - x)/(h);
double value = 0.0;
if( u>= 0.0 && u <=1.0)
{
value = 1- 2.5*u*u + 1.5*u*u*u;
return value;
}
else if( u>=1.0 && u<=2.0)
{
value = 1.0/2.0*(2.0-u)*(2.0-u)*(1.0-u);
return value;
}
else
{
return 0.0;
}
}
```

References

[1] Petros D. Koumoutsakos *Direct Numerical Simulations of Unsteady separated flows using Vortex Methods.* Ph.D Thesis, California Institute of technology, 1993.

[2] Lorena A. Barba *Vortex Method for computing high-Reynolds number flows: Increased accuracy with a fully mesh-less formulation* Ph.D Thesis, California Institute of technology, 2004.

[3] Petros D. Koumoutsakos *Simulations using particles* HPCSE I Class Notes.

[4] R.W. Hockney and J.W. Eastwood *Computer Simulations Using Particles* IOP Publishing Limited, 1998.

[5] Rajesh Ramaswamy and Ivo F. Sbalzarini *Particle methods: Computing with particles* Lecture Notes: TU Dresden, Faculty of Computer Science Chair of Scientific Computing for Systems Biology, August 2015.

[6] Particle Methods
`http://mosaic.mpi-cbg.de/?q=education/courses/particlemethods`