Complex Flows - Hand in 2

Finnur Mauritz Einarsson

January 22, 2024

1

During the collision of two particles in the simulation there is a possibility that the particles will stick together instead of repelling. The reason being, that the simulation works in specified time steps and if the particles are in collision, they are assumed to have left the collision after an adjustment has been made to the velocity. Originally, the velocities were adjusted in such a way that a Δv was calculated using equation (2), where $\mathbf{v}_1^{\{0\}}$ and $\mathbf{v}_2^{\{0\}}$ are the particle's velocities before the collision. Then, Δv is added to both particle's velocities with opposite signs. The relative velocity, $\mathbf{G}^{\{0\}}$ is described in equation (1).

$$\mathbf{G}^{\{0\}} = \mathbf{v}_1^{\{0\}} - \mathbf{v}_2^{\{0\}} \tag{1}$$

$$\Delta v = \mathbf{n}(\mathbf{G}^{\{0\}} \cdot \mathbf{n}) \tag{2}$$

This works fine as long as during the next time step, the particles are no longer intersecting. If they are, the signs are reversed again, causing them to stick together and with higher speed, they can appear to vibrate. A simple, and perhaps a lazy solution, would be to make a matrix that records if two particles are in a collision and using that to only allow the velocities to update once per collision of two particles.

Using this method, a significant number of *sticking* can be averted. This problem will however reappear as the assignment progresses.

2

Before applying friction on sliding particles and restitution coefficient a few properties of the collision have to be defined.

Using $\mathbf{G}^{\{0\}}$ from equation (1) the relative velocity of the contact point before the collision can be defined [1].

$$\mathbf{G}_c^{\{0\}} = \mathbf{G}^{\{0\}} + r_1\omega_1^{\{0\}} \times \mathbf{n} + r_2\omega_2^{\{0\}} \times \mathbf{n} \tag{3}$$

In equation (3), the radius of both particles, r_1 and r_2, are multiplied with the corresponding angular velocity, $\omega_1^{\{0\}}$ and $\omega_2^{\{0\}}$. The cross product of these terms with the normal vector, $\mathbf{n}$, provides a tangential contribution to the relative velocity at the contact point, $\mathbf{G}_c^{\{0\}}$ [1]. The tangential component of $\mathbf{G}_c^{\{0\}}$ is defined in equation (4)

$$\mathbf{G}_{ct}^{\{0\}} = \mathbf{G}_c^{\{0\}} - \left(\mathbf{G}_c^{\{0\}} \cdot \mathbf{n}\right)\mathbf{n} \tag{4}$$

Using the length of the vector $\mathbf{G}_{ct}^{\{0\}}$, the tangential unit vector can be defined in equation (5)

$$\mathbf{t} = \frac{\mathbf{G}_{ct}^{\{0\}}}{|\mathbf{G}_{ct}^{\{0\}}|}. \tag{5}$$

Now, the coefficient of restition is defined in equation (6), in order to better simulate particle to particle and particle to wall collisions.

$$\mathbf{n} \cdot \mathbf{G}^{\{0\}} = -e\left(\mathbf{n} \cdot \mathbf{G}^{\{0\}}\right) \tag{6}$$

the normal and tangential components of the Impulsive force are given by equation (7) and equation (8).

$$J_n = -\frac{m_1 m_2}{m_1 + m_2}(1+e)(\mathbf{n} \cdot \mathbf{G}^{\{0\}}) \tag{7}$$

$$J_t = fJ_n \tag{8}$$

During particle to particle collision, two scenarios have to be considered: If the particles slide through collision, or not. The velocities and angular velocities of the particles are determined depending on if they slide through collision or not. The condition can be described with equation (9) [1].

$$J_t > -\left(\frac{2}{7}\right)\frac{m_1 m_2}{m_1+m_2}|\mathbf{G}_{ct}^{\{0\}}| \tag{9}$$

If the particles slide through the collision the velocities and angular velocities after the collision can be evaluated using equation (10), equation (11), equation (12) and equation (13) [1].

$$\mathbf{v_1} = \mathbf{v_1}^{\{0\}} - (\mathbf{n} + f\mathbf{t})(\mathbf{n}\cdot\mathbf{G}^{\{0\}})(1+e)\frac{m_2}{m_1+m_2} \tag{10}$$

$$\mathbf{v_2} = \mathbf{v_2}^{\{0\}} + (\mathbf{n} + f\mathbf{t})(\mathbf{n}\cdot\mathbf{G}^{\{0\}})(1+e)\frac{m_1}{m_1+m_2} \tag{11}$$

$$\omega_\mathbf{1} = \omega_\mathbf{1}^{\{0\}} - \frac{5}{2r_1}(\mathbf{n}\cdot\mathbf{G}^{\{0\}})(\mathbf{n}\times\mathbf{t})(1+e)\frac{m_2}{m_1+m_2} \tag{12}$$

$$\omega_\mathbf{2} = \omega_\mathbf{2}^{\{0\}} - \frac{5}{2r_2}(\mathbf{n}\cdot\mathbf{G}^{\{0\}})(\mathbf{n}\times\mathbf{t})(1+e)\frac{m_2}{m_1+m_2} \tag{13}$$

If the condition is not fulfilled, that is, if the particles do not slide through the condition the particles' properties are found using equation (14), equation (15), equation (16) and equation (17) [1].

$$\mathbf{v_1} = \mathbf{v_1}^{\{0\}} - \left((1+e)(\mathbf{n}\cdot\mathbf{G}^{\{0\}})\mathbf{n} + \frac{2}{7}|\mathbf{G}_{ct}^{\{0\}}|\mathbf{t}\right)\frac{m_2}{m_1+m_2} \tag{14}$$

$$\mathbf{v_2} = \mathbf{v_2}^{\{0\}} + \left((1+e)(\mathbf{n}\cdot\mathbf{G}^{\{0\}})\mathbf{n} + \frac{2}{7}|\mathbf{G}_{ct}^{\{0\}}|\mathbf{t}\right)\frac{m_1}{m_1+m_2} \tag{15}$$

$$\omega_\mathbf{1} = \omega_\mathbf{1}^{\{0\}} - \frac{5}{7r_1}|\mathbf{G}_{ct}^{\{0\}}|(\mathbf{n}\times\mathbf{t})\frac{m_2}{m_1+m_2} \tag{16}$$

$$\omega_\mathbf{2} = \omega_\mathbf{2}^{\{0\}} - \frac{5}{7r_2}|\mathbf{G}_{ct}^{\{0\}}|(\mathbf{n}\times\mathbf{t})\frac{m_1}{m_1+m_2} \tag{17}$$

This way, particle to particle collisions are modelled more accurately. For particle to wall collisions, the velocity and angular velocity of the particle is scaled by the coefficient of restitution.

Applying these conditions greatly affects the average velocity of the particles over time. In order to quantify these changes, the ratio of the average particle speed to the initial average particle speed is calculated, equation (18), for each time step.

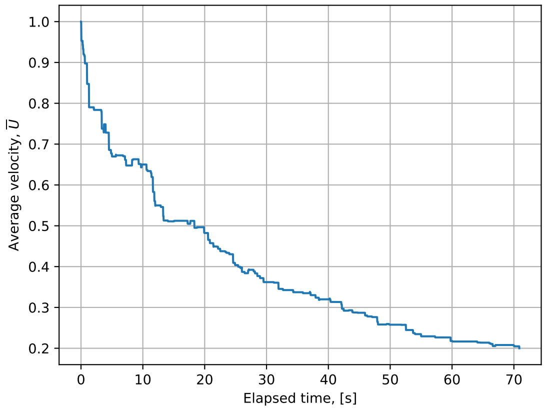
$$\overline{U} = \frac{U_{average}}{U_{initial}} \tag{18}$$

The time it takes for $\overline{U}$ to reach 0.2 will be the variable of interest. At that point, the average velocity of the particles in the system will have lowered by 80% of the initial value. The ratio of 0.2 is chosen based on observations of the simulation and was found to be a reasonable ratio as the rate at which the average velocities decrease slows down significantly after that point. This variable of interest is defined by equation (19).

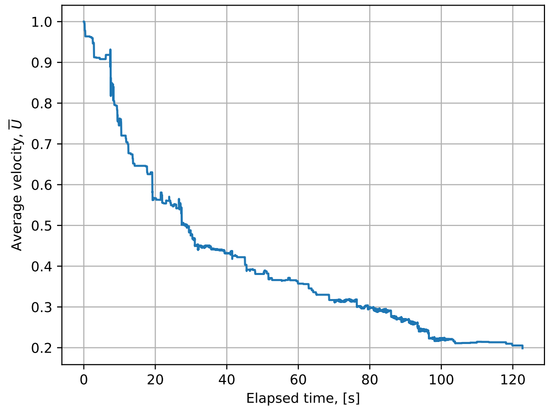
$$\tau = t_{\overline{U}=0.2} \tag{19}$$

Since the initial velocities are randomized, there is a non predictive factor in the model, making the variable τ inconsistent to some degree. However, in general, τ is relatively consistent between models.

Comparing figure 1b and figure 1a, the improved model shows consistently a $\tau \approx 62$s while the original model has a consistent $\tau \approx 118$s. This shows that the implementation of the coefficients of restitution and friction significantly speed up the rate at which the kinetic energy in the system is lost. The kinetic energy of the system is determined by calculating the kinetic energy of each particle and adding them up. The kinetic energy of the system is plotted on figure 2.



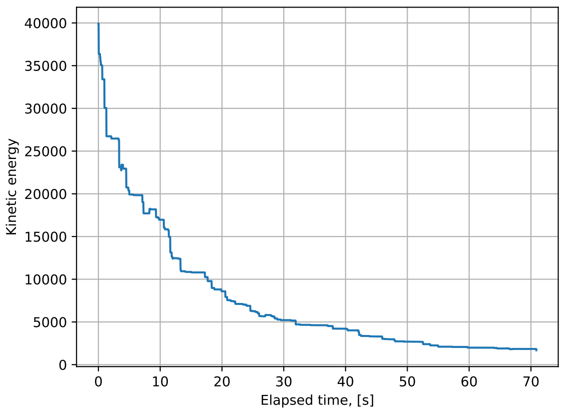
(a) Normalized average velocity - with friction



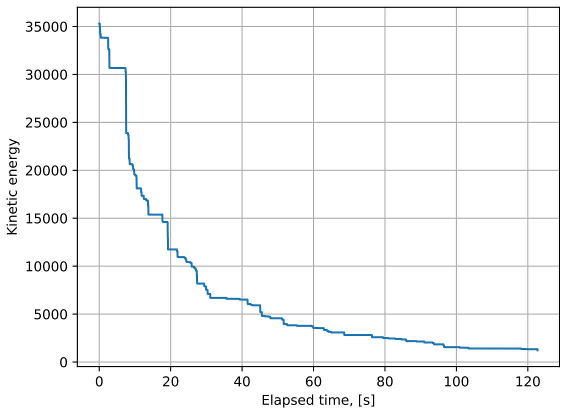
(b) Normalized average velocity - without friction

Figure 1: Normalized average velocity

As mentioned, the velocity is normalized while the kinetic energy is not which is why the y-axis on figure 2a and figure 2b does not match. Due to this, the normalized velocity is a more descriptive property of the flow to analyse than the kinetic energy of the system.



(a) Kinetic energy - with friction



(b) Kinetic energy - without friction

Figure 2: Kinetic energy

3

In it's current state the particles are only affected by their interactions between other particles and the wall. Implementing passive forces will change the model drastically. For this assignment the velocity of each particle will be adjusted by three different accelerations in the form of their respective forces. The sum of the forces are used to adjust the velocity. Each force is represented by a matrix of the same size as the matrix that describes the velocity of the particles.

Force due to gravitational pull

The gravitational acceleration is implemented in the code by making the second column, or the y-component, of the force matrix equal to the gravitational acceleration of $-9.81\ \frac{\mathrm{m}}{\mathrm{s}^2}$ times the mass of each particle. For each particle, i, the force due to the gravitational pull is defined in equation (20), where M is the mass of the particle. In the current model the mass is the same for each particle and it's value is 1.

$$F_g^{\{i\}} = \begin{bmatrix} 0 \\ -9.81 \end{bmatrix} \cdot M \tag{20}$$

Force due to Drag

The Drag force on each particle is calculated using equation (21) [2] where u_i and v_i are the horizontal and vertical components of the particle's velocity, C_D is the coefficient of drag and A is the area normal to the flow direction, in this case $A = r^2 * \pi$ where r is the radius of the particle. The vector n_d is a directional unit vector of the velocity.

$$F_d^{\{i\}} = -\frac{1}{2}(\mathrm{C}_D \cdot A\left(\sqrt{u_i^2 v_i^2}\right)^2)\, \mathbf{n_d} \tag{21}$$

Force due to Coriolis acceleration

The Coriolis acceleration is calculated using equation (22) [2].

$$F_c^{\{i\}} = -2 \cdot (\omega_\mathbf{i} \times \mathbf{n_d}) \tag{22}$$

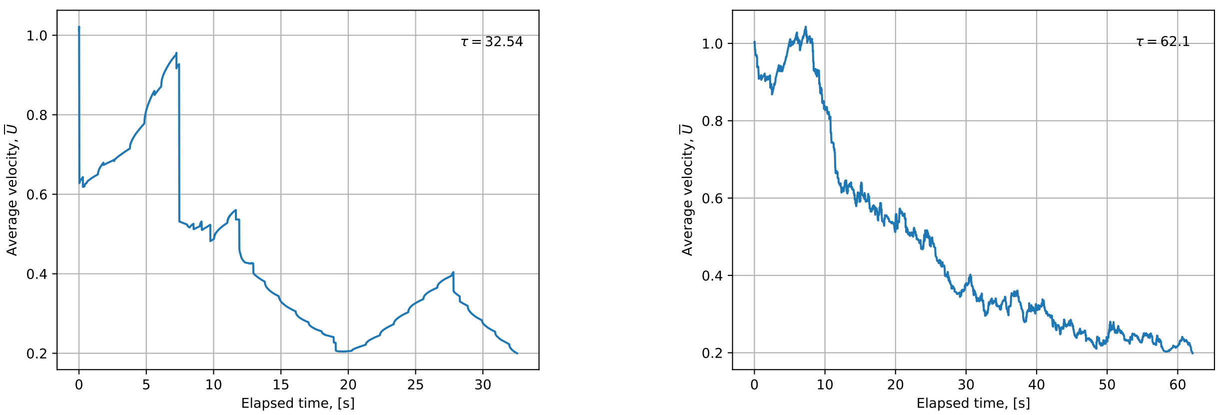
The sum of these forces, F_{sum}, are used to adjust the velocity before the particles are moved according to equation (23) where dt is the time increment by which the program moves each run.

$$\mathbf{v} = \mathbf{v}^{\{0\}} + \frac{F_{sum}}{M} \cdot dt \tag{23}$$

The simulation now shows the particles collect at the bottom, causing them to clip, similarly as in the first part of this assignment, however due to a different reason. Now, the particles are clipping since the particles settle at a lower velocity and are generally closer so when the velocity after collision is updated it is often not enough to expel the particles away from one another. This and the added effect of the gravitational acceleration, which accelerates the colliding particles in the same direction, while one might be at the bottom, gaining no velocity. Decreasing dt can reduce this effect.

4

Varying the number of particles provides an interesting comparison of the simulation's velocity plots. On figure 3a, the small number of particles makes particle to particle collision a rarity and with the added effect from the gravitational force, the particles bounce independently until the wall collision and other forces slow them down. The particle bounce can be seen on each plot in figure 3 where the average velocity increases as the particles accelerate towards the bottom, hit the bottom, the vertical velocity is reversed with relatively low loss, and then the particle slows down as it travels away from the bottom.



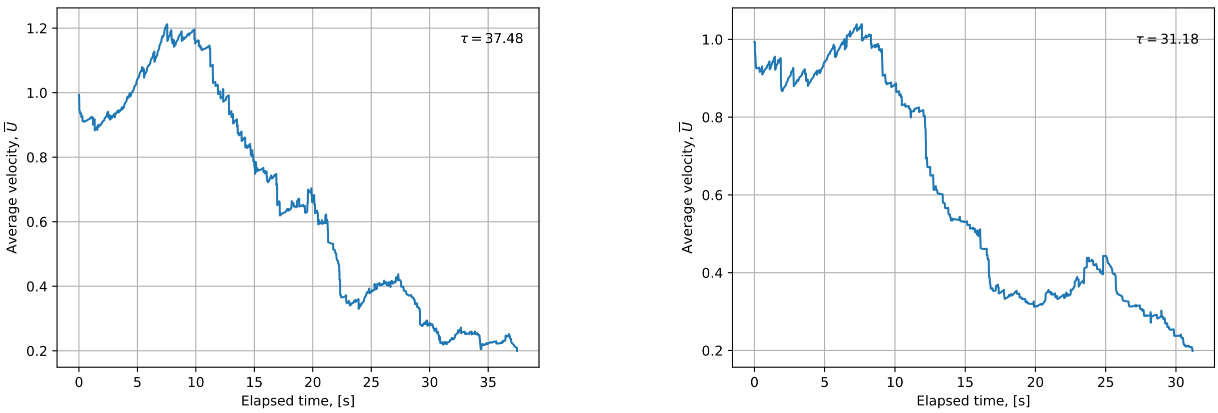
(a) Normalized average velocity, N = 5

(b) Normalized average velocity, N = 35

Figure 3: Normalized average velocity with varying number of particles, N

As the number of particles increase the value of τ increases, meaning the system takes a longer time to slow lose it's energy. With increased rate of particle to particle interaction, there are more instances where the particles slide through the collision, causing them to spin. Due to the Coriolis acceleration in combination with the gravitational acceleration as the particles begin to spin more while accelerating towards the bottom the particles begin to not only spin but travel in a circular pattern. This causes erraticity in the average velocity which can clearly be seen increasing with the number of particles on figure 3.

All previous simulations were made using the same fluid density of $\rho_{\text{air}} = 1.239 \frac{\text{kg}}{\text{m}^3}$. Increasing the density of the fluid should increase the effects of the drag force. In theory this should lower the τ value. The model simulated 20 particles in a fluid with the density of air, water, corn syrup and molasses. As can be seen on figure 4, the τ value does decrease however not by a large margin. This suggests either that even still the gravitational force has a much larger effect on the particles than the drag force or that the drag force is not correctly implemented.



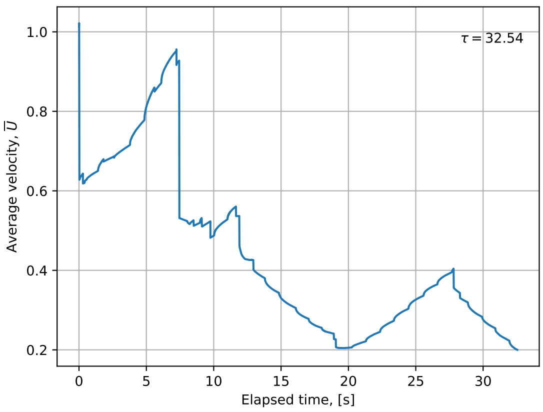
(a) Normalized average velocity, $\rho = 1.239 \frac{\text{kg}}{\text{m}^3}$

(b) Normalized average velocity, $\rho = 1600 \frac{\text{kg}}{\text{m}^3}$

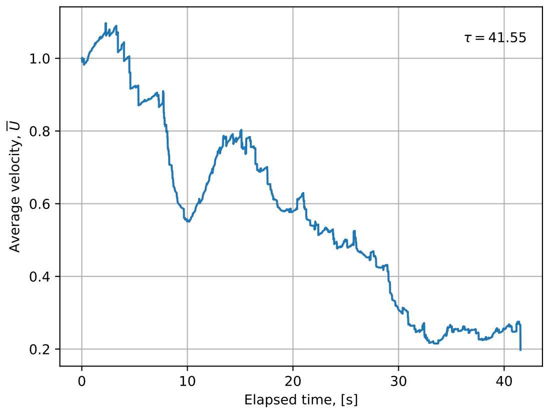
Figure 4: Normalized average velocity with varying fluid density, ρ

APPENDIX

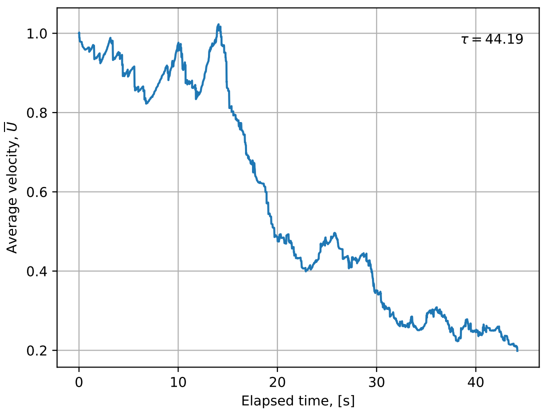
additional plots



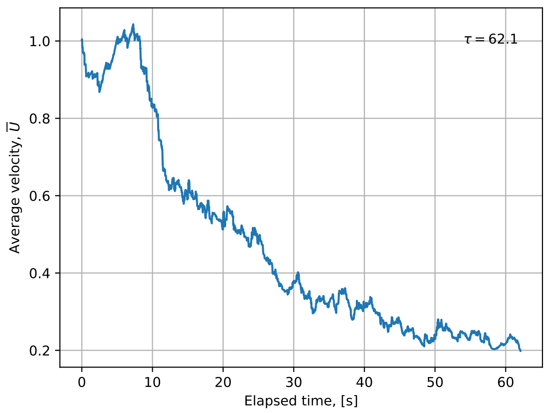
(a) Normalized average velocity, N = 5



(b) Normalized average velocity, N = 15

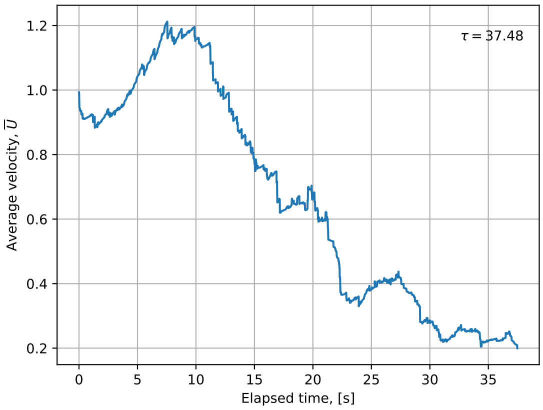


(c) Normalized average velocity, N = 25

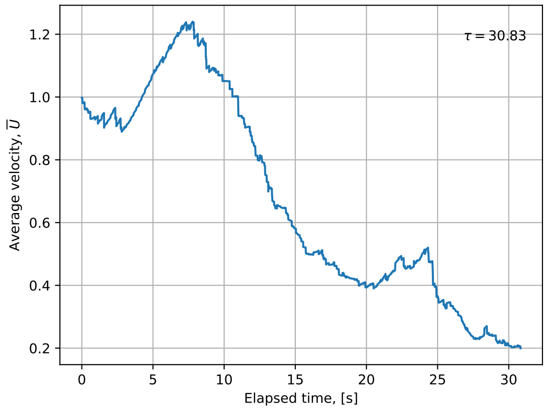


(d) Normalized average velocity, N = 35

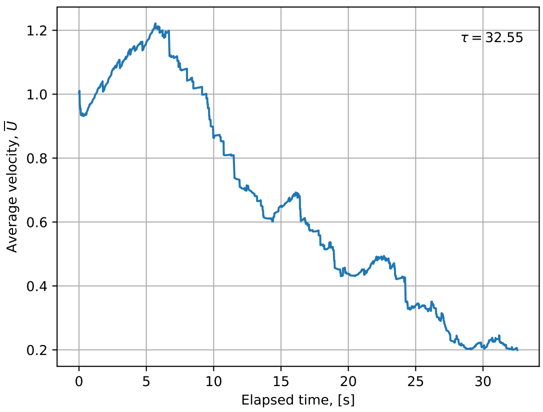
Figure 5: Normalized average velocity with varying number of particles, N



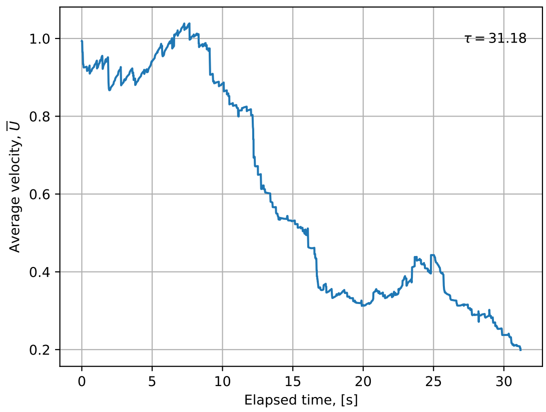
(a) Normalized average velocity, $\rho = 1.239\ \frac{\mathrm{kg}}{\mathrm{m}^3}$



(b) Normalized average velocity, $\rho = 997\ \frac{\mathrm{kg}}{\mathrm{m}^3}$



(c) Normalized average velocity, $\rho = 1380\ \frac{\mathrm{kg}}{\mathrm{m}^3}$



(d) Normalized average velocity, $\rho = 1600\ \frac{\mathrm{kg}}{\mathrm{m}^3}$

Figure 6: Normalized average velocity with varying fluid density, ρ

Model Code

```python
import numpy as np
from numpy import sqrt, cos, sin, array, zeros
import pyglet
import matplotlib.pyplot as plt

#parameters
window_size = np.array((800, 800))
nparticles = 35
radius = 20
max_velocity = 100  #Pixels per second
max_omega = 0.5
color1 = (20, 20, 250) # blue
color2 = (200, 200, 250) # light blue
prevents = 0

#Tracked variables
plotVel = []
plotKE = []
collision = np.zeros((nparticles,nparticles))
timevalue = float(0)
dt = 1/20000
initialV = 0
AvgVel = 0
KE = 0

#Physical properties
A  = (radius*10**(-3))**2*np.pi
mass = 1
f = 0.5          # friction coefficient
```

```python
e = 0.8          # coefficient of restitution
CD = 0.5         #coefficient of drag  - F_D = CD*1/2*rho*U^2*A
CL = 0.1         #coefficient of lift
FD = np.zeros((nparticles,2) )         #Drag force on particles
Fg = np.zeros((nparticles,2) )
FC = np.zeros((nparticles,2) )
FR = np.zeros((nparticles,2) )
rho = 1.239   #density of air
#rho_W =  997    #density of water
#rho_S = 1500
#rho_M = 1200
#t0 = time()

# initialize global variables for particles
position = zeros((nparticles, 2))
velocity = zeros((nparticles, 2))
angle = zeros(nparticles)
omega = zeros(nparticles)
circles = []

# make window and batch
window = pyglet.window.Window(window_size[0], window_size[1])
batch = pyglet.graphics.Batch()

# functions for particle simulation

def avg_velocity(velocity):
    Sum = 0
    for i in range(len(velocity)):
        Sum += np.sqrt(velocity[i,0]**2+velocity[i,1]**2)
    avgVel = Sum/len(velocity)
    return avgVel

def totKE(velocity):
    KE = 0
    for i in range(nparticles):
        KE += 0.5*length(velocity[i])**2*mass
    return KE

def length(vector):
    length = np.sqrt(vector[0]**2+vector[1]**2)
    return length

def make_particles(position, velocity, omega, circles):
    # make particles in a grid at least one diameter from walls
    xstart =  np.ones(2) * radius * 2
    xlength = window_size - 2 * xstart
    n = np.ceil(np.sqrt(nparticles))
    m = np.ceil(nparticles / n)
    xg, yg = np.meshgrid(np.arange(n)/(n-1), np.arange(m)/(m-1))
    # make random velocities and rotation
    velangle = np.random.rand(nparticles) * 2 * np.pi
    velocity += np.random.rand(nparticles).reshape(-1,1) * max_velocity \
                * array([cos(velangle), sin(velangle)]).T
    omega += (np.random.rand(nparticles) - 0.5) * 2 * max_omega
    global initialV
    initialV = avg_velocity(velocity)
    # prepare particles for simulation
    for i in range(nparticles):
        position[i,0] = xg.flatten()[i] * xlength[0] + xstart[0]
        position[i,1] = yg.flatten()[i] * xlength[1] + xstart[1]
        # generate circle for particle
        circles.append(pyglet.shapes.Circle(position[i,0], position[i,1],
                                            radius, color=color1, batch=batch))
        # generate spot on particle to track rotation
        x, y = position[i,:] + 0.5 * radius * array([cos(angle[i]), sin(angle[i])])
        circles.append(pyglet.shapes.Circle(x, y, 0.25*radius,
                                            color=color2, batch=batch))

def Forces(dt, velocity, omega, rho):
    for i in range(nparticles):
        nF = velocity[i]/length(velocity[i])
        nF3 = [nF[0], nF[1], 0]
        #tF= np.array(nF[:,2],[-nF[:,1], nF[:,0]]).T
        temp_omega = [0, 0, omega[i]]
        FC[i,0] = -2*np.cross(temp_omega,nF3)[0]
```

```
        C3 = -2*np.cross(temp_omega,nF3)        #Coriolis acceleration
        FC[i,0] = C3[0]*mass                    #
        FC[i,1] = C3[1]*mass
        FD = -(CD*0.5*length(velocity[i])**2*A)*nF
        Fg[i,1] = -9.81*mass

    Fsum = FD+Fg+FC
    velocity +=  Fsum/mass*dt

def move_particles(dt, position, angle, circles):
    position += velocity * dt
    position = np.round(position,decimals=2)
    angle += omega * dt
    # update circle positions
    for i in range(nparticles):
        circles[i*2].position = position[i]
        circles[i*2+1].position = position[i] + 0.5 * radius * array([cos(angle[i]), sin(angle[i])])

def wall_collision(velocity):
    # handle particle collition with the walls
    for i in range(nparticles):
        if position[i,0] < radius:
            velocity[i,0] = e*abs(velocity[i,0])
            omega[i] = omega[i]*-e
        if position[i,0] > window_size[0] - radius :
            velocity[i,0] = -e*abs(velocity[i,0])
            omega[i] = omega[i]*-e
        if position[i,1] < radius:
            velocity[i,1] = e*abs(velocity[i,1])
            omega[i] = omega[i]*-e
        if position[i,1] > window_size[1] - radius :
            velocity[i,1] = -e*abs(velocity[i,1])
            omega[i] = omega[i]*-e

def particle_collision(velocity):
    for i in range(nparticles-1):
        for j in range(i+1, nparticles):
            distance = sqrt(((position[j] - position[i])**2).sum())
            if  distance < 2 * radius and collision[i,j] == 0:
                # collision! - apply textbook eq. 5.14
                collision[i,j] = 1
                collision[j,i] = 1
                n = (position[j] - position[i]) / (distance)

                G0 = velocity[i] - velocity[j]
                G03 = np.zeros(3)
                G03[:-1] = G0
                n3 = np.zeros((1,3))
                n3[0,:-1] = n
                omega3 = np.zeros((nparticles,3))
                omega3[:,2] = omega
                Gc3 = G03+np.cross(radius*omega3[i],n3) + np.cross(n3,radius*omega3[j])
                Gc = Gc3[0, :2]
                Gct = Gc - (np.dot(Gc, n))*n
                Gc3 = np.squeeze(np.asarray(Gc3))
                n3 = np.squeeze(np.asarray(n3))
                Gct3 = Gc3 - (np.dot(Gc3, n3))*n3
                lengthGct = length(Gct)
                t = Gct/ lengthGct
                t3 = Gct3 / length(Gct3)
                Jn = -(mass**2/(2*mass))*(1+e)*(np.dot(n,G0))
                Jt = Jn * f
                if Jt > -(2/7)*mass**2/(mass*2)* lengthGct:
                    dveli = (n+f*t)*(np.dot(n,G0))*(1+e)*mass/(mass*2)
                    dvelj = (n+f*t)*(np.dot(n,G0))*(1+e)*mass/(mass*2)
                    domegi = (5)/(2*radius) *np.dot(n3,G03)*(np.cross(n3,t3))*f*(1+e)*mass/(mass*2)
                    domegj = (5)/(2*radius) *np.dot(n3,G03)*(np.cross(n3,t3))*f*(1+e)*mass/(mass*2)

                else:
                    dveli = ((1+e)*(np.dot(n,G0))*n + 2/7*lengthGct * t)*mass/(2*mass)
                    dvelj = ((1+e)*(np.dot(n,G0))*n + 2/7*lengthGct * t)*mass/(2*mass)
                    domegi = 5/(7*radius)*length(Gct3) * (np.cross(n3,t3)) * mass/(mass*2)
                    domegj = 5/(7*radius)*length(Gct3) * (np.cross(n3,t3)) * mass/(mass*2)

                velocity[i] -= dveli
                velocity[j] += dvelj
```

```python

                domegi = domegi[2]      #Since the simulation is two-dimensional, only spin around the z-axis affects the problem.
                domegj = domegj[2]

                omega[i] -= np.sign(omega[i]) * domegi
                omega[j] += np.sign(omega[j]) * domegj
            elif  distance < 2 * radius and collision[i,j] == 1:

                global prevents
                prevents += 1
                #print('t: ', time()-t0, '   prevented events: ', prevents)
            else:
                collision[i,j] = 0
                collision[j,i] = 0

def particle_collisionoriginal(velocity):
    # handle collision between particles using simple loops
    for i in range(nparticles-1):
        for j in range(i+1, nparticles):
            distance = sqrt(((position[j] - position[i])**2).sum())
            if  distance < 2 * radius:
                # collision! - apply textbook eq. 5.14
                n = (position[j] - position[i]) / distance
                G0 = velocity[i] - velocity[j]
                dvel = n * np.dot(n, G0)
                velocity[i] -= dvel
                velocity[j] += dvel

def plot(i, xi, yi, xlabel, ylabel, tau):
    tau = np.round(tau, decimals=2)
    plt.figure(i)
    plt.clf()
    plt.text(0.87*max(xi),0.95*max(yi), r'$\tau = {}$'.format(tau))
    plt.plot(xi,yi)
    plt.grid()
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show(block = False)


# modify pyglet draw command to draw our particles
@window.event
def on_draw():
    window.clear()
    batch.draw()

# This is want we do in each time step
def update(dt):
    global timevalue
    global initialV
    global AvgVel
    timevalue += dt
    Forces(dt, velocity, omega, rho)
    move_particles(dt, position, angle, circles)
    wall_collision(velocity)
    particle_collision(velocity)

    AvgVel = avg_velocity(velocity)
    KE = totKE(velocity)
    normVel = AvgVel/initialV
    plotVel.append(normVel)
    plotKE.append(KE)

    if normVel < 0.2:
        pyglet.app.exit()
        return


# run the following if this is the main script
if __name__ == "__main__":

    # Update the game 120 times per second
    pyglet.clock.schedule_interval(update, dt)

    # Create particles with random position and velocity
    make_particles(position, velocity, omega, circles)
```

```python

    # Tell pyglet to do its thing
    pyglet.app.run()
    x = np.linspace(0, timevalue, len(plotVel))
    plot(1,x,plotVel,'Elapsed time, [s]', r'Average velocity, $\overline{U}$', timevalue)
    plot(2,x,plotKE, 'Elapsed time, [s]', 'Kinetic energy', timevalue)
    print(r'\tau',timevalue)
    del window
    del batch
```

Bibliography

[1] Clayton T. Crow et al. *Multiphase Flows with Droplets and Particles*. 2012.

[2] Pijush K. Kundu, Ira M. Cohen, and David R. Downling. *Fluid Mechanics*. 2016.