FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Jian Wu – xcb479@alumni.ku.dk

Deep Contact
Accelerating Rigid Simulation With Convolutional Networks

Supervisor: Kenny Erleben

August 6th 2018

Abstract

This is a master theis from

Contents

1 Introdustion **1**

1.1 Motivation 1

1.2 Thesis Overview 1

2 Rigid Body Dynamics Simulation **2**

2.1 Rigid dynamics Simulation 2

2.1.1 Simulation Basics 2

2.1.2 Rigid Body Concepts 3

2.1.3 Rigid Body Equations of Motions 3

2.2 Contact Forces Solver 4

2.2.1 Constraints 4

2.2.2 Solving Constraints 7

2.3 Simulation Results 9

3 Partcle-grid-particle **10**

3.1 Grid-Based method 10

3.2 Smoothed Particle Hydrodynamics 10

3.2.1 Fundamentals 11

3.2.2 Kernels 12

3.2.3 Grid size and smoothing length 14

3.2.4 Neignbor Search 14

3.3 Grid to particle 15

3.3.1 Bilinear interpolation 16

3.4 Conclution 17

4 Deep Learning For Simulation **18**

4.1 Convolutional Neural Networks 18

4.2 CNN Constructure 18

4.2.1 Convolutional layers 18

4.2.2 Full-connected layers 18

4.3 Traing configuration 18

4.3.1 Loss Function 18
4.3.2 Optimizer(SGD) 18
4.4 Training Results 18
4.5 Simulation based on Trained model 18

5 Implementation 19

6 Conclusion 20
6.1 Future work 20

References 21

List of Figures

3.1 Grid description, $retrievedfromMIT(2011)$ 10

3.2 Visilaztion of SPH 11

3.3 Comparation of different kernels, we set smoothing length $h = 1$ here. 13

3.4 Comparation of gradient of different kernels, we set $h = 1$ here. 14

3.5 The figure shows visiualization of bilinear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate. 16

List of Tables

Chapter 1

Introdustion

1.1 Motivation

Movies studies have been pushing facial and character animation to a level where machine learning can automize a large part of this work in a production pipeline. Research community is exploring machine learning for gait control too and quite successfully or for upscaling of liquid simulation[1].

However, for rigid body problems it is not quite clear how to approach the technicalities in applying deep learning. Some work have been done in terms of inverse simulations or pilings to control rigid bodies to perform a given artistic 'target'. These thechniques are more in spirit of inverse problems that maps initial conditions to a well defined outcome(number of bounces or which face up on a cude) or level of detail idea replacing interiors of piles with stracks of cylinders pf decreasing radius to make an overall apparent pile have a given angle of of repose.

1.2 Thesis Overview

Chapter 2

Rigid Body Dynamics Simulation

This chapter mainly introduces rigid body simulation to help you understand how computer simulate rigid dynamics based on traditional newton-euler equations. For more details, some contact forces solvers are decribed in this chapter. Afterwards, we will use one of solver to run some simulation and get the image data for the next step, grids-transfer. All the discussion about rigid simulation and contacts solver are based on 2-D view.

2.1 Rigid dynamics Simulation

2.1.1 Simulation Basics

Simulating the motion of a rigid body is almost the same as simulating the motion of a particle, so I will start with partcle simulation. For particle simulation, we let function $x(t)$ describe the particle's location in world space at time t. Then we use $v(t) = \frac{\mathrm{d}}{\mathrm{d}(t)}x(t)$ to denote the velocity of the particle at time t. So, the state of a particle at a time t is the particle's position and velocity. We generalize this concept by defining a state vector $\mathbf{Y}(t)$ for a system: for a single particle,

$$\mathbf{Y}(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \end{pmatrix} \tag{2.1}$$

For a system with n particles, we enlarge $\mathbf{Y}(t)$ to be

$$\mathbf{Y}(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ ... \\ x_n(t) \\ v_n(t) \end{pmatrix} \tag{2.2}$$

However, to simulate the motion of particles actually, we need to know one more thing – the forces. $F(t)$ is defined as the force acting on the particle. If the mass of the particle is m, then the changes of $\mathbf{Y}(t)$ will be given by

$$\frac{\mathrm{d}}{\mathrm{d}(t)}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}(t)} \begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ F(t)/m \end{pmatrix} \tag{2.3}$$

2.1.2 Rigid Body Concepts

Unlike a particle, a rigid body occupies a volume of space and has a particular shape. Rigid bodies are more complicated, beside translating them, we can rotate them as well. To locate a rigid body, we use $x(t)$ to denote their translation and a rotation matrix $R(t)$ to describe their rotation.

2.1.3 Rigid Body Equations of Motions

Finally, we can covert all concepts we need to define the state $\mathbf{Y}(t)$ for a rigid body.

$$\mathbf{Y}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} \tag{2.4}$$

Like what is epressed in $\mathbf{Y}(t)$, the state of a rigid body is mainly consist by its position and orientation (describing spatial information), and its linear and angualr momentum(describe velocity information). Since mass M and bodyspace inertia tensor I_{body} are constants, we can the auxiliary quantities $I(t)$, $\omega(t)$ at any given time.

$$v(t) = \frac{P(t)}{M} \quad I(t) = R(t)I_{body}R(t)^T \quad \omega(t) = I(t)^{-1}L(t) \tag{2.5}$$

The derivative $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{Y}(t)$ is

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x(t) \\ R(t) \\ Mv(t) \\ L(t) \end{pmatrix} = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{pmatrix} \tag{2.6}$$

Then, we can conclude the simulation algorithm

```
Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while running the simulation world do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end
```

Algorithm 1: How to write algorithms

2.2 Contact Forces Solver

2.2.1 Constraints

In general form, a constraint is a scalar equation equal to some value (usually zero).

$$C(l_1, a_1, l_2, a_2) = 0 \tag{2.7}$$

The l and a terms in (2.7) are my own notation: l refers to linear while a refers to angular. The subscripts 1 and 2 refer to the two objects within the constraint. As you can see, there exist linear and angular inputs to a constraint equation, and each must be a scalar value.

Let's take a step back to look at the distance constraint. The distance constraint wants to drive the distance between two anchor points on two bodies to be equal to some scalar value:

$$C(l_1, a_1, l_2, a_2) = \frac{1}{2}[\|\mathbf{P}_2 - \mathbf{P}_1\|^2 - L^2] = 0 \tag{2.8}$$

L is the length of the rod connecting both bodies; $\mathbf{P}_1$ and $\mathbf{P}_2$ are the positions of the two bodies.

In its current form, this constraint is an equation of position. This sort of position equation is non-linear, which makes solving it very hard. A method of solving this equation can be to instead derive the position constraint (with respect to time) and use a velocity constraint. Resulting velocity equations are linear, making them solvable. Solutions can then be integrated using some sort of integrator back into positional form.

In general form, a velocity constraint is of the form:

$$\dot{C}(l_1, a_1, l_2, a_2) = 0 \tag{2.9}$$

The dot above the C in (2.9) refers to the derivative of C with respect to time. This is common notation when dealing with the study of physics.

During the derivative, a new term J appears via chain rule:

$$\dot{C}(l_1, a_1, l_2, a_2) = \mathbf{JV} = 0 \tag{2.10}$$

The time derivative of C creates a velocity vector and a Jacobian. The Jacobian is a 1x6 matrix containing scalar values corresponding to each degree of freedom. In a pairwise constraint, a Jacobian will typically contain 12 elements (enough to contain the l and a terms for both bodies A and B.

A system of constraints can form a joint. A joint can contain many constraints restricting degrees of freedom in various ways. In this case, the Jacobian will be a matrix where the number of rows is equal to the number of constraints active in the system.

The Jacobian is derived offline, by hand. Once a Jacobian is acquired, code to compute and use the Jacobian can be created. As you can see from (2.10), the velocity $\mathbf{V}$ is transformed from Cartesian space to constraint space. This is important because in constraint space the origin is known. In fact, any target can be known. This means that any constraint can be derived to yield a Jacobian that can transform forces from Cartesian space to constraint space.

In constraint space, given a target scalar, the equation can move either towards or away from the target. Solutions can easily be obtained in constraint space to move the current state of a rigid body towards a target state. These solutions can then be transformed out of constraint space back into Cartesian space like so:

$$\mathbf{F} = \lambda \mathbf{J}^T \tag{2.11}$$

F is a force in Cartesian space, where J^T is the inverse (transposed) Jacobian. λ (lambda) is a scalar multiplier.

Think of the Jacobian as a velocity vector, where each row is a vector itself (of two scalar values in 2D, and three scalar values in 3D):

$$\mathbf{J} = \begin{bmatrix} l_1 \\ a_1 \\ l_2 \\ a_2 \end{bmatrix} \tag{2.12}$$

To multiply **V** by **J** mathematically would involve matrix multiplication. However, most elements are zero, and this is why we treat the Jacobian as a vector. This allows us to define our own operation for computing JV, as in (2.10).

$$\mathbf{JV} = \begin{bmatrix} l_1 & a_1 & l_2 & a_2 \end{bmatrix} \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} \tag{2.13}$$

Here, **v** represents linear velocity, and ω represents angular velocity. (2.13) can be written down as a few dot products and multiplications in order to provide a more efficient computation compared to full matrix multiplication:

$$\mathbf{JV} = l_1 \cdot v_1 + a_1 \cdot \omega_1 + l_2 \cdot v_2 + a_2 \cdot \omega_2 \tag{2.14}$$

The Jacobian can be thought of as a direction vector in constraint space. This direction always points towards the target in the direction that requires the least work to be done. Since this "direction" Jacobian is derived offline, all that needs to be solved for is the magnitude of the force to be applied in order to uphold the constraint. This magnitude is called λ. λ can be known as the Lagrange Multiplier. I myself have not formally studied Lagrangian Mechanics, however a study of Lagrangian Mechanics is not necessary in order to simply implement constraints. (I am proof of that!) λ can be solved using a constraint solver (more on this later).

2.2.2 Solving Constraints

In Erin Catto's paper[2], there exists a simple outline for hand-deriving Jacobians. The steps are:

Start with constraint equation C Compute time derivative $\dot{C}$ Isolate all velocity terms Identify **J** by inspection

The only hard part is computing the derivative, and this can come with practice. In general, hand-deriving constraints is difficult, but gets easier with time.

Let's derive a valid Jacobian for use in solving a distance constraint. We can start at Step 1 with (2.8). Here are some details for Step 2:

$$\dot{C} = (P_2 - P_1)(\dot{P}_2 - \dot{P}_1) \tag{2.15}$$

$$\dot{C} = (P_2 - P_1)((v_2 + \omega_2 \times r_2) - (v_1 + \omega_1 \times r_1)) \tag{2.16}$$

r_1 and r_2 are vectors from the center of mass to the anchor point, for bodies 1 and 2 respectively.

The next step is to isolate the velocity terms. To do this, we'll make use of the scalar triple product identity:

$$(P_2 - P_1) = d \tag{2.17}$$

$$\dot{C} = (d \cdot v_2 + d \cdot \omega_2 \times r_2) - (d \cdot v_1 + d \cdot \omega_1 \times r_1) \tag{2.18}$$

$$\dot{C} = (d \cdot v_2 + \omega_2 \cdot r_2 \times d) - (d \cdot v_1 + \omega_1 \cdot r_1 \times d) \tag{2.19}$$

The last step is to identify the Jacobian by inspection. In order to do this, all the coefficients of all velocity terms (V and ω) will be used as the Jacobian elements. Therefore:

$$J = \begin{bmatrix} -d & -r_1 \times d & d & r_2 \times d \end{bmatrix} \tag{2.20}$$

Some more Jacobians

Contact constraint (interpenetration constraint), where n is the contact normal:

$$J = \begin{bmatrix} -n & -r_1 \times n & n & r_2 \times n \end{bmatrix} \tag{2.21}$$

Friction constraint (active during penetration), where t is an axis of friction (2D has one axis, 3D has two):

$$J = \begin{bmatrix} -t & -r_1 \times t & t & r_2 \times t \end{bmatrix} \tag{2.22}$$

Now that we have an understanding of what a constraint is, we can talk about how to solve them. As stated earlier, once a Jacobian is hand-derived, we only need to solve for λ. Solving a single constraint in isolation is easy, but solving many constraints simultaneously is hard, and very inefficient (computationally). This poses a problem, as games and simulations will likely want to have many constraints active all at once.

An alternative method to solving all constraints simultaneously (globally solve) would be to solve the constraints iteratively. By solving for approximations of the solution, and feeding in previous solutions to the equations, we can converge on the solution.

One such iterative solver is known as Sequential Impulses, as dubbed by Erin Catto. Sequential Impulses is very similar to Projected Gauss Seidel. The idea is to solve all constraints, one at a time, multiple times. The solutions will invalidate each other, but over many iterations each individual constraint will converge and a global solution can be achieved. This is good! Iterative solvers are fast.

Once a solution is achieved, an impulse can be applied to both bodies in the constraint in order to enforce the constraint.

To solve a single constraint, we can use the following equation:

$$\lambda = \frac{-(JV + b)}{JM^{-1}J^T} \tag{2.23}$$

M^{-1} is the mass of the constraint; b is the bias (more on this later).

This is a matrix containing the inverse mass and inverse inertia of both rigid bodies in the constraint. The following is the mass of the constraint; note that m^{-1} is the inverse mass of a body, while I^{-1} is the inverse inertia of a body:

$$M^{-1} = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} \tag{2.24}$$

Although M^{-1} is theoretically a matrix, please do not actually model it as such (most of it is zeroes!). Instead, be smart about what sort of calculations you do.

$JM^{-1}J^T$ is known as the constraint mass. This term is calculated one time and used to solve for λ. We calculate it for a system like so:

$$JM^{-1}J^T = (l_1 \cdot l_1) * m_1^{-1} + (l_2 \cdot l_2) * m_2^{-1} + a_1 * (I_1^{-1} a_1) + a_2 * (I_2^{-1} a_2) \tag{2.25}$$

Please note that you must invert (2.25) in order to compute (2.23).

The above information is all that is needed to solve a constraint! A force in Cartesian space can be solved for and used to update the velocity of an object, in order to enforce a constraint. Please recall (2.11):

$$F = \lambda J^T V_{final} = V_{initial} + m^{-1} * F \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} + = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} \begin{bmatrix} \lambda * l_1 \\ \lambda * a_1 \\ \lambda * l_2 \\ \lambda * a_2 \end{bmatrix} \tag{2.26}$$

2.3 Simulation Results

Chapter 3

Partcle-grid-particle

The basic method for generating training data which is more accessible to learning is that we will map a discrete element method(DEM) into a continuum setting use techniques from smooth particle hydrodynamics. Given a set of bodies δ and a set of contacts between these bodies C.

3.1 Grid-Based method

Traditional rigid motion simulation mainly use particle-based method. However, if we want to replace traditional contact solver with deep learning model, it is hard for cnn model to recognize the original image and do learning. Grid-based methos is a good to transfer original image to a grid-cells and then use

3.2 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) was invented to simulate nonaxisymmetric phenoma in astrophysis initially. The principal

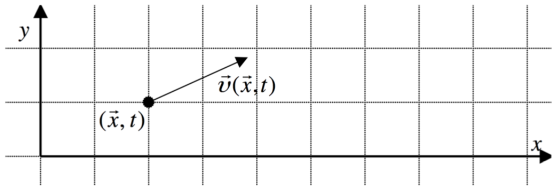

Figure 3.1: Grid description, $retrievedfromMIT$(2011)

idea of SPH is to treat hydrodynamics in a completely mesh-free fashion, in terms of a set of sampling particles. It turns out that the particle presentation of SPH has excellent conservation properties. Energy, linear momentum, angular momentum, mass and velocity.

3.2.1 Fundamentals

At the heart of SPH is a kernel interpolation method which allows any function to be expressed in terms of its values at a set of disordered points - the particles[3]. For ant field $A(\mathbf{r})$, a smoothed interpolated version $A_I(\mathbf{r})$ can be defined by a kernel $W(\mathbf{r}, h)$,

$$A_I(\mathbf{r}) = \int A(\mathbf{r}')W(\|\mathbf{r} - \mathbf{r}'\|, h)\, \mathrm{d}\mathbf{r}' \tag{3.1}$$

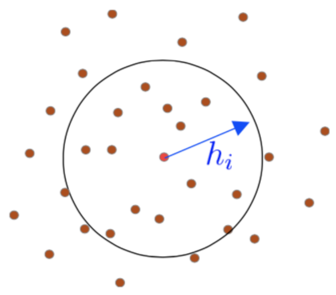

Figure 3.2: Visilaztion of SPH

where the integration is over the entire space, and W is an interpolating kernel with

$$\int W(\|\mathbf{r} - \mathbf{r}'\|, h)\, \mathrm{d}\mathbf{r}' = 1 \tag{3.2}$$

and

$$\lim_{h \to 0} W(\|\mathbf{r} - \mathbf{r}'\|, h)\, \mathrm{d}\mathbf{r}' = \delta(\|\mathbf{r} - \mathbf{r}'\|) \tag{3.3}$$

Normally, we want the kenel to be Non-negative and rotational invariant.

$$W(\|\mathbf{x}_i - \mathrm{x}_j\|, h) = W(\|\mathrm{x}_j - \mathrm{x}_i\|, h) \tag{3.4}$$

$$W(\|\mathbf{r} - \mathbf{r}'\|, h) \geq 0 \tag{3.5}$$

For numerical work, we can use midpoint rule,

$$A_I(\mathbf{x}) \approx A_S(\mathbf{x}) = \sum_i A(\mathbf{x}_i) W(\|\mathbf{x}_i - \mathbf{x}\|, h) \Delta V_i \tag{3.6}$$

Since $V_i = m_i/\rho_i$

$$A_S(\mathbf{x}) = \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) W(\|\mathbf{x}_i - \mathbf{x}\|, h) \tag{3.7}$$

The default, gradient and Laplacian of A are:

$$\begin{aligned} \nabla A_S(\mathbf{x}) &= \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) \nabla W(\|\mathbf{x}_i - \mathbf{x}\|, h) \\ \nabla^2 A_S(\mathbf{x}) &= \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) \nabla^2 W(\|\mathbf{x}_i - \mathbf{x}\|, h) \end{aligned} \tag{3.8}$$

3.2.2 Kernels

Smoothing kernels functions are one of the most important points in SPH. Stability, accurancy and speed of the whole method depends on these fuctions. Different kernels are being used for different purposes. One possibilyty for W is a Gaussian. However, most current SPH implementations are based on kernels with finite support. We mainly introduce gaussian, poly6 and spicky kernel here. And compare the different kernels and their property.

Poly6

The kernel is also known as the 6th degree polynomial kernel.

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)^3 & 0 \le \|\mathbf{r}\| \le h \\ 0 & \text{Otherwise} \end{cases} \tag{3.9}$$

Then, the gradient of this kernel function can be

$$\nabla W_{poly6}(\mathbf{r}, h) = -\frac{945}{32\pi h^9} \begin{cases} \mathbf{r}(h^2 - \|\mathbf{r}\|^2)^2 & 0 \le \|\mathbf{r}\| \le h \\ 0 & \text{Otherwise} \end{cases} \tag{3.10}$$

The laplacian of this kenel can be expressed by,

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = -\frac{945}{16\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)(3h^2 - 7\|\mathbf{r}\|^2) & 0 \le \|\mathbf{r}\| \le h \\ 0 & \text{Otherwise} \end{cases} \tag{3.11}$$

Spicky

The kernel proposed by Desbrum[4]

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\mathbf{r}\|)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.12}$$

Then, the gradient of spiky kernel can be described by,

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45\mathbf{r}}{\pi h^6 \|\mathbf{r}\|} \begin{cases} (h - \|\mathbf{r}\|)^2 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.13}$$

The laplacian of spiky can be expressed by,

$$\nabla^2 W_{spiky}(\mathbf{r}, h) = \frac{90}{\pi h^6} \begin{cases} h - \|\mathbf{r}\| & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.14}$$

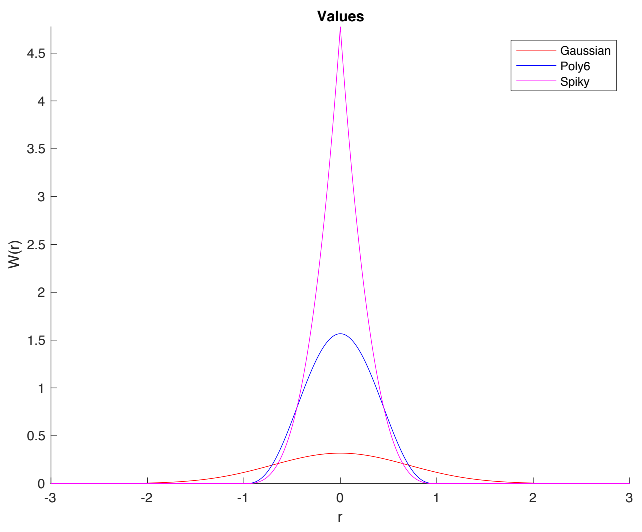

Figure 3.3: Comparation of different kernels, we set smoothing length $h = 1$ here.

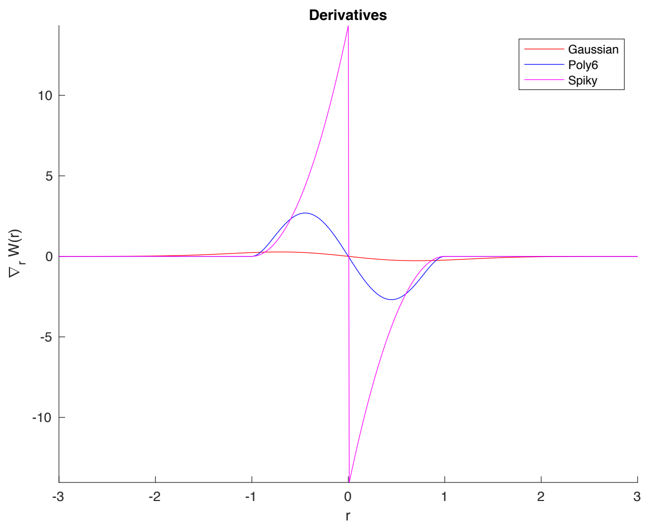

Figure 3.4: Comparation of gradient of different kernels, we set $h = 1$ here.

3.2.3 Grid size and smoothing length

The grid should be also fine enough to capture the variation in our simulation. In our case, it is reasonable to have a grid fine enough such that no two contact points are mapped into the same cell.

Smoothing length, h, is one of the most important parameters that affects the whole SPH method by changing the kernel value results abd neighbor searching results. Too small or too big values might cause lose essencail information in the simulation.

3.2.4 Neignbor Search

Neighbor search is one of the most crucial procedures in SPH method considersing all interpolation equations, $A(\mathbf{r})$, needs the neighbor list for every particle (refer to equation 3.8). A naive neighbor searching approach would end up with a complexity of $o(n^2)$. The complexity is not good enough since it is impossoble to reach any interactive speed when the particle count increses. With an efficient nearest neighbor searching(NNS) algorithm, it is possible to have a significant performance increase since it is the most time consuming procedure in SPH computation. In order to decrease the complexity, we choose to use $k - d$ tree data structure to store the particla spatial information and then do the nearest neighbor searching.

Hierarchical Tree

Using an adaptive hierarchy tree search is proposed by Paiva[5] to find particle neighbors. Since the simulation takes place in two dimensions, $k - d$ tree data structure was used in this approach.

An octree structure has been adapted from Macey (b) Octree Demo. The hierarchy tree is formed with a pre-defined height. The simulation box is divided recursively into eight pieces, nodes. The nodes at height = 1 are called leaves. Each node has a surrounding box for particle query which is used to check if the particle is inside the node. Each parent node, contains the elements that are divided through its children. The particle is being checked at each level of the tree if it's intersecting the node. If it does, descend one level down in that node. After reaching to leaves, bottom level, particle has been checked if its within the search distance, h. If the particle is in the range, add it to the neighbor list. Neighbor searching is done when the whole tree has been traversed. The search distance set to be smoothing length in this implementation.

The complexity of this tree search method is $O(nlog(n))$, n being the number of particles. The performance of this algorithm is worse than Spatial Hashing method. In addition, the results obtained using this NNS algorithm wasn't accurate and stable for this implementation. Therefore as mentioned above, Spatial Hashing method was preferred.

3.3 Grid to particle

After getting the grid image for simulation state in time t, we will use the grid cells as input and renew the grid image based on trained model. Once the grid image in $t + \Delta t$ have been renewed, the next step is to transfer the grid to particles which will instore all information including velocity,

3.3.1 Bilinear interpolation

We applied bilinear interpolation in our case, since we did mainly research on a rectilinear $2-D$ grid. The key idea is to perform lin-

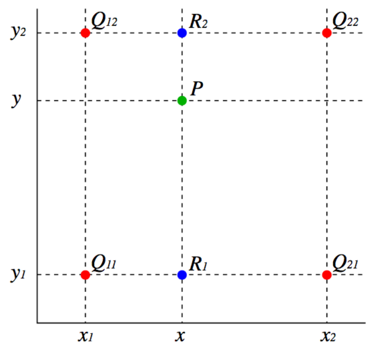

Figure 3.5: The figure shows visiualization of bilinear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate.

ear interpolation first in one direction, and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

As shown in Figure 3.5, We have known $Q_{a,b} = (x_a, y_b)$ and $a \in \{1,2\} \quad b \in \{1,2\}$. Then, we can firstly do linear interpolation in the x-direction. This yields

$$\begin{aligned} f(x,y_1) &\approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \\ f(x,y_2) &\approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \end{aligned} \tag{3.15}$$

After getting the two values in $x-$direction $f(x,y_1)$ and $f(x,y_2)$, we can use these values to do interpolation in $y-$ direction.

$$f(x,y) \approx \frac{y_2 - y}{y_2 - y_1} f(x,y_1) + \frac{y - y_1}{y_2 - y_1} f(x,y_2) \tag{3.16}$$

Combine $f(x,y_1)$ and $f(x,y_2)$ defined in equation 3.15, we can get,

$$
\begin{aligned}
f(x,y) \approx{}& \frac{y_2-y}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})\right)\\
&+\frac{y-y_1}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})\right)\\
={}&\frac{1}{(x_2-x_1)(y_2-y_1)}\big(\\
&f(Q_{11})(x_2-x)(y_2-y)+f(Q_{21})(x-x_1)(y_2-y)\\
&+f(Q_{12})(x_2-x)(y-y_1)+f(Q_{22})(x-x_1)(y-y_1)\big)\\
={}&\frac{1}{(x_2-x_1)(y_2-y_1)}\begin{bmatrix}x_2-x & x-x_1\end{bmatrix}\\
&\cdot\begin{bmatrix}f(Q_{11}) & f(Q_{12})\\ f(Q_{21}) & f(Q_{22})\end{bmatrix}\begin{bmatrix}y_2-y\\ y-y_1\end{bmatrix}
\end{aligned}
\tag{3.17}
$$

3.4 Conclution

Chapter 4

Deep Learning For Simulation

4.1 Convolutional Neural Networks

4.2 CNN Constructure

4.2.1 Convolutional layers

4.2.2 Full-connected layers

4.3 Traing configuration

4.3.1 Loss Function

4.3.2 Optimizer(SGD)

4.4 Training Results

4.5 Simulation based on Trained model

Chapter 5

Implementation

Chapter 6

Conclusion

6.1 Future work

References

[1] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, “Accelerating Eulerian Fluid Simulation With Convolutional Networks”, *ArXiv e-prints*, Jul. 2016. arXiv: `1607.03597 [cs.CV]`.

[2] E. Catto, “Modeling and solving constraints”, in *Game Developers Conference*, 2009, p. 16.

[3] J. J. Monaghan, “Smoothed particle hydrodynamics”, *Annual review of astronomy and astrophysics*, vol. 30, no. 1, pp. 543–574, 1992.

[4] M. Desbrun and M.-P. Gascuel, “Smoothed particles: A new paradigm for animating highly deformable bodies”, in *Computer Animation and Simulation’96*, Springer, 1996, pp. 61–76.

[5] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares, “Particle-based non-newtonian fluid animation for melting objects”, in *Computer Graphics and Image Processing, 2006. SIBGRAPI’06. 19th Brazilian Symposium on*, IEEE, 2006, pp. 78–85.