Interpolated water retention after two-body collisions using Neural Networks and linear interpolation methods

Bachelor's Thesis

Lukas Winkler

supervised by

Mag. Dr. Thomas Maindl

1. September 2019

a01505981@unet.univie.ac.at

Contents

Abstract

To get a closer estimate on how much water remains after the collision of two protoplanets or asteroids covered in water, a total of 1375 SPH simulations have been conducted. Six parameters like impact velocity, angle and mass have been varied between the simulations to estimate for many possible collision scenarios. To interpolate the resulting water retention fraction for collisions in between the simulations, the three methods gridbased linear interpolations, Radial Basis Functions and Artificial Neural Networks have been used. This allows to predict the remaining water for arbitrary collisions within the simulated parameter range.

1. Introduction

One important question for planet formation is how water got to the earth. The part of the protoplanetary disk closest to the sun was too hot to make it possible that water can condense on Earth during formation. And while there are theories that the region where ice is possible inside the snow-line moved during Earth's formation[1], the most popular theory is that water moved inwards in the solar system through collisions of water-rich proto-planets.

To better understand how this process works, large n-body simulations over the lifetime of the solar systems have been conducted[2]. Most of these neglect the physical details of collisions when two bodies collide for simplicity and instead assume that a perfect merging occurs. So the entire mass of the two progenitor bodies and especially all of their water (ice) is retained in the newly created body. Obviously this is a simplification as in real collisions perfect merging is very rare and most of the time either partial accretion or a hit-and-run encounter occurs.[3] Therefore, the amount of water retained after collisions is consistently overestimated in these simulations. Depending on the parameters like impact angle and velocity, a large fraction of mass and water can be lost during collisions.[4]

To understand how exactly the water transport works, one has to find an estimate of the mass and water fractions that are retained during two-body simulations depending on the parameters of the impact. First, I will be shortly describing the simulation setup, the important parameters and the post-processing of the results (Chapter 2). Next I will summarize the results of the simulations and their properties (Chapter 3). In the main section I will then be describing three different approaches to interpolate and generalize these results for arbitrary collisions (Chapter 4). Finally I'll compare the three methods and show their advantages and disadvantages (Chapter 5).

[1]Martin and Livio 2012.
[2]for example Dvorak, Eggl, et al. 2012
[3]Stewart and Leinhardt 2012.
[4]T. I. Maindl et al. 2017.

2. Simulations

2.1. Model

For a realistic model of two gravitationally colliding bodies the smooth particle hydrodynamics (`SPH`) code `miluphCUDA` as explained in C. Schäfer et al. 2016 and C. M. Schäfer et al. 2019 is used. It is able to simulate brittle failure and the interaction between multiple materials.

In the simulation two celestial bodies are placed far enough apart so that tidal forces can affect the collision (5 times the sum of the radii). Both objects consist of a core with the physical properties of basalt rocks and an outer mantle made of water ice. These two-body-collisions are similar to those that happen between protoplanets or the collision that created the Earth's Moon.[5]

To keep the simulation time short and make it possible to do many simulations with varying parameters, 20k SPH particles are used and each simulation is run for 12 hours in which every 144 seconds the current state is saved.

2.2. Parameters

Six parameters have been identified that have a major influence on the result of a two-body collision. All selected parameter ranges are inspired by the physical properties that occurred in collisions in a simulation of an early solar system.[6] (Table 2.1)

2.2.1. Impact velocity

The collision velocity v_0 is defined in units of the mutual escape velocity v_{esc} of the projectile and the target.[7] Simulations have been made from $v_0 = 1$ to $v_0 = 5$. As one expects, a higher velocity results in a stronger collision and more and smaller fragments.

$$v_{esc} = \sqrt{\frac{2G(M_p + M_t)}{r_p + r_t}} \tag{2.1}$$

2.2.2. Impact angle

The impact angle is defined in a way that $\alpha = 0°$ corresponds to a head-on collision and higher angles increase the chance of a hit-and-run encounter. The simulation parameters range from $\alpha = 0°$ to $\alpha = 60°$

[5]Dvorak, Loibnegger, and Thomas I. Maindl 2015.
[6]Thomas I. Maindl and Dvorak 2014.
[7]T. I. Maindl et al. 2017.

v_0	1	1.5	2	3	5
α	0°	20°	40°	60°	
m	10^{21} kg	10^{23} kg	10^{24} kg	10^{25} kg	
γ	0.1	0.5	1		
water fraction target	10 %	20 %			
water fraction projectile	10 %	20 %			

Table 2.1.: parameter set of the first simulation run

2.2.3. Target and projectile mass

The total masses in these simulations range from about two Ceres masses (1.88×10^{21} kg) to about two earth masses (1.19×10^{25} kg). In addition to the total mass m, the mass fraction between projectile and target γ is defined. As the whole setup is symmetrical between the two bodies, only mass fractions below and equal to one have been considered.

2.2.4. Water fraction of target and projectile

The last two parameters are the mass fraction of the ice to the total mass of each of the bodies. To keep the numbers of parameter combinations and therefore required simulations low, only 10 % and 20 % are simulated in the first simulation set.

2.3. Execution

In the first simulation run for every parameter combination from Table 2.1 a separate simulation has been started. First, the parameters and other configuration options are written in a `simulation.input` text file. Afterwards the relaxation program described in Burger, T. I. Maindl, and C. M. Schäfer 2018, pp. 24 sqq. generates relaxed initial conditions for all 20k particles and saves their state to `impact.0000`. Finally, `miluphcuda` can be executed with the following arguments to simulate starting from this initial condition for 300 timesteps which each will be saved in a `impact.XXXX` file.

```
miluphcuda -N 20000 -I rk2_adaptive -Q 1e-4 -n 300  -a 0.5 -H -t 144.0 -f impact.0000 -m material.cfg -s -g
```

This simulation ran on the `amanki` server using a `Nvidia GTX 1080` taking about 30 min per simulation as the `Nvidia GTX 1080` is the fastest consumer GPU for this simulation set in a comparison of 13 tested GPUs.[8] Of these 960 simulations, 822 succeed and were used in the analysis.

2.4. Post-processing

After the simulation the properties of the SPH particles needs to be analyzed. To do this, the `identify_fragments` C program by Christoph Burger (part of the post-processing tools of `miluphCUDA`) uses a friends-of-friends algorithm to group the final particles into fragments. Afterwards `calc_aggregates` calculates the mass of the two largest fragments together with their gravitationally bound fragments and its output is written into a simple text file (`aggregates.txt`).

[8]Dorninger 2019.

2.5. Resimulation

To increase the amount of available data and especially reduce the errors caused by the grid-based parameter choices (Table 2.1), a second simulation run has been started. All source code and initial parameters have been left the same apart from the six main input parameters described above. These are set to a random value in the range listed in Table 2.2 apart from the initial water fractions. As they seem to have little impact on the outcome (see Section 3.1), they are set to 15 % to simplify the parameter space.

This way, an additional 553 simulations have been calculated on `Nvidia Tesla P100` graphics cards on `Google Cloud`. (Of which 100 simulations are only used for comparison in Chapter 5)

	min	max
v_0	1	5
α	$0°$	$60°$
m	1.88×10^{21} kg	1.19×10^{25} kg
γ	0.1	1
water fraction target	15 %	15 %
water fraction projectile	15 %	15 %

Table 2.2.: parameter ranges for the resimulation

3. Simulation outcome

For the large set of simulations, we can now extract the needed values. The output of the relaxation program (`spheres_ini_log`) gives us the precise values for impact angle and velocity and the exact masses of all bodies. As these values differ slightly from the parameters explained in Section 2.2 due to the setup of the simulation, in the following steps only the precise values from `spheres_ini_log` are considered. From the `aggregates.txt` explained in Section 2.4 the final masses and water fractions of the two largest fragments are extracted. From these, the main output considered in this analysis, the water retention of the two fragments, can be calculated.

3.1. Correlations

One very easy, but sometimes flawed[9] way to look at the whole dataset at once is calculating the *Pearson correlation coefficient* between the input parameters and the output water fraction (Figure 3.1). This shows the expected result that a higher collision angle (so a more hit-and-run like collision) has a higher water retention and a higher collision speed results in less water left on the two largest remaining fragments. In addition, higher masses seem to result in less water retention. The initial water fractions of the two bodies does seem to have very little influence on the result of the simulations.

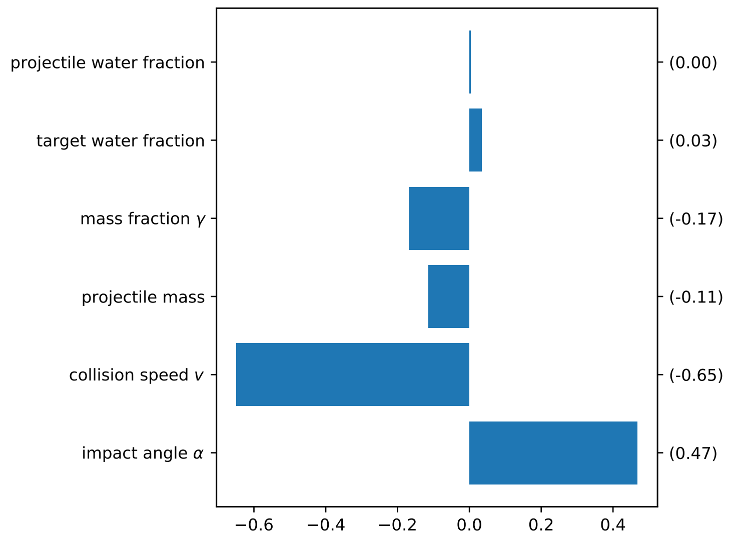


Figure 3.1.: The Pearson correlation coefficient visualized as a bar graph

[9]The Pearson correlation coefficient only measures linear correlations. With a value close to zero there can still be a non-linear correlation between the two dimensions. In addition, the coefficient gives no information about the steepness of the correlation, only about which fraction of the values conform to it.

4. Interpolations

4.1. Multidimensional linear interpolation

4.1.1. Theory

One of the easiest ways to interpolate a new value between two known values is linear interpolation. It takes the closest values and creates a linear function between them.

In one dimension, linear interpolation is pretty trivial. For example, let's assume that we have 20 random points P between 0 and 1 (● and ● in Figure 4.1a) and have a new point I (●) at 0.4 for which we want to interpolate. Finding the two closest points (●) above and below is trivial as there is only one dimension to compare. Now, if we have measured a value $f(P)$ for each of these points, a straight line (|) between the two closest values can be drawn and an interpolated value for $f(I)$ can be found.

In two dimensions things get more complicated as we now have a set of points with X and Y coordinates (Figure 4.1b). One fast way to find the closest points to the point that should be interpolated is using Delaunay triangulation. This is a method to separate the space between the points into triangles while trying to maximize their smallest angle and to make sure no other point is inside the circumcircle of the triangles[10]. Connecting the centers of the circumcircles results in a Voronoi diagram.

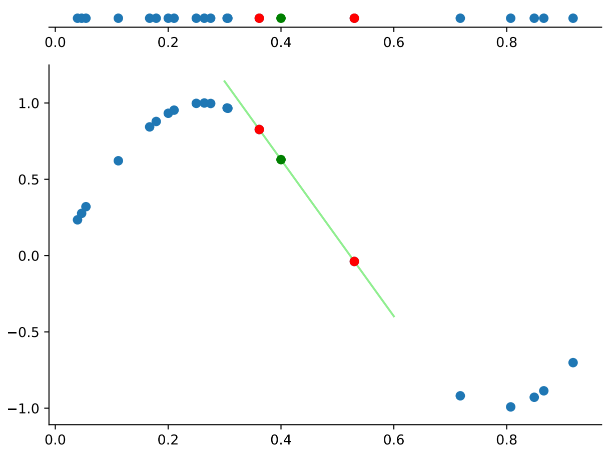
Afterwards, the closest three points can be found very quickly by checking the nodes of the surrounding triangle (Figure 4.2a). If we now again have a function $f(X,Y)$ similar to the one-dimensional example, we can create a unique plain through the three points and get the interpolated value for any pair of X and Y on this layer. (Figure 4.2b)

This approach has the advantage that it can be extended in more than two dimensions by replacing the triangle in the Delaunay triangulation with an n-simplex in n dimensions. The `scipy.spatial.Delaunay` python function allows to quickly calculate it thanks to the `Qhull` library[11]. One noticeable limitation of this method is that data can't be extrapolated. Therefore, the possible output is limited to the convex hull of the input parameter space (as seen in Figure 4.2a).

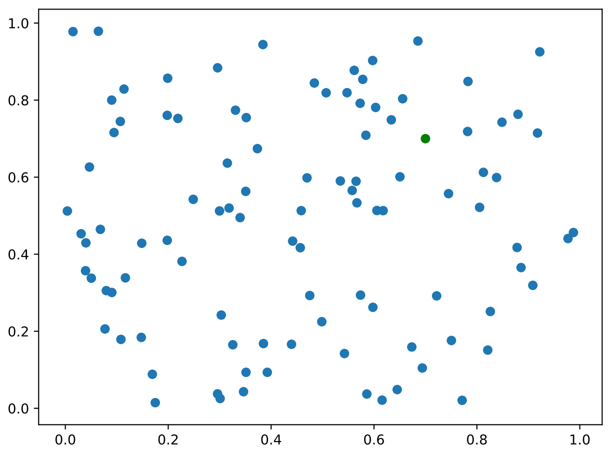
4.1.2. Implementation

For doing the actual interpolations, the `scipy.interpolate.griddata` function is used with the `method="linear"` argument which itself uses `scipy.interpolate.LinearNDInterpolator` to do an interpolation similar to the one described above. The function is given a $6 \times n$ matrix of the six parameters and an n-sized list of the water retention fraction for those n simulations. In addition, `griddata` supports not only calculating interpolations for one set of parameters, but also for lists of parameters which allows to quickly generate 2d diagrams as seen in Figure 4.3.

[10]Delaunay 1934.
[11]`http://www.qhull.org/`

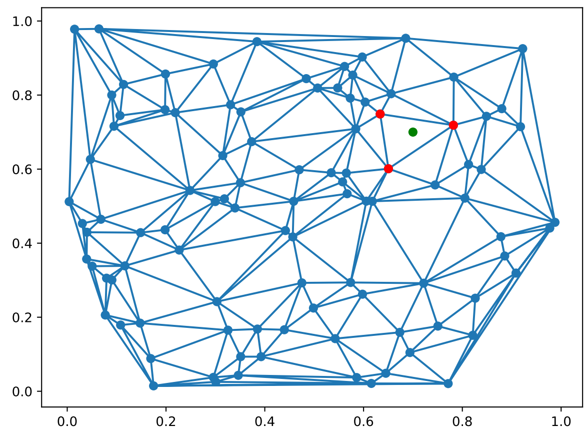


(a) A one-dimensional example of linear interpolation

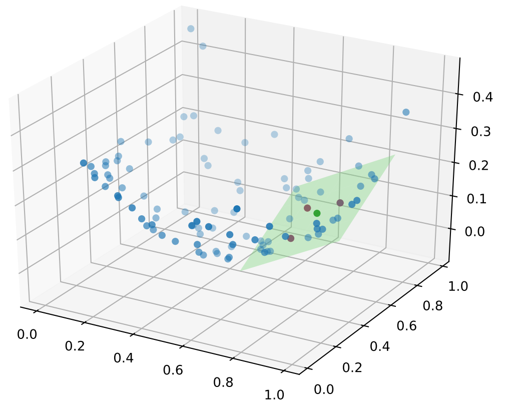


(b) A set of two-dimensional datapoints

Figure 4.1.



(a) A Delaunay triangulation of the points from Figure 4.1b

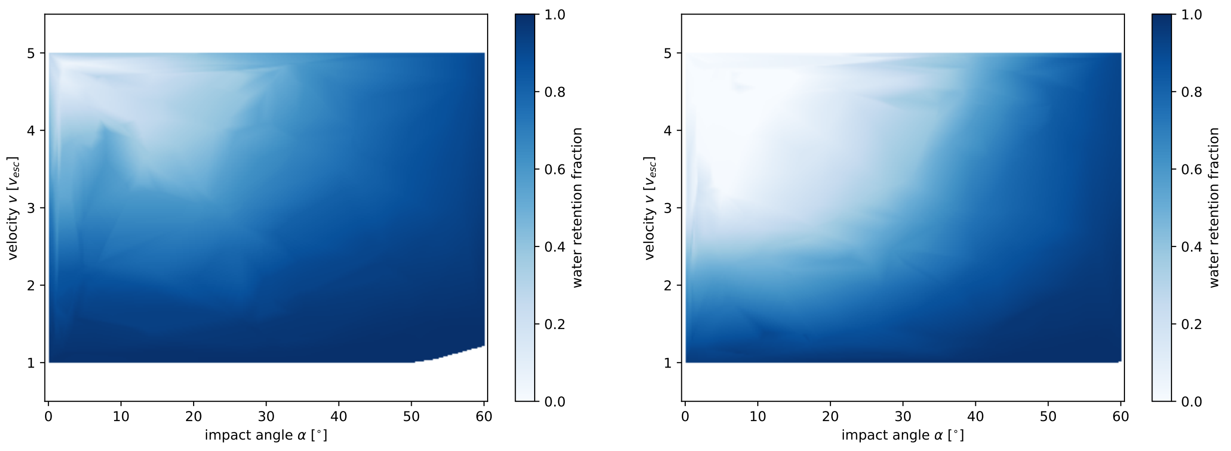


(b) A $f(X, Y)$ interpolated via the green plane

Figure 4.2.

4.1.3. Results

Most notable about the results of the griddata interpolation (Figure 4.3) are the many fine details that can be seen. This is mostly caused by the fact that this method only uses the closest values for interpolations and therefore there is no smoothing. These details might just be random derivations of the simulation and not a higher resolution of the data. Another thing that can be seen in the bottom right corner of Figure 4.3a is that griddata can't extrapolate data.



(a) $m_{total} = 10^{22}$, $\gamma = 0.6$, $wt = wp = 0.15$

(b) $m_{total} = 10^{24}$, $\gamma = 0.6$, $wt = wp = 0.15$

Figure 4.3.: Interpolation result using griddata

4.2. RBF interpolation

4.2.1. Theory

Another approach to interpolate data is using *Radial Basis Functions* (RBF). A very good explanation on how they work is given in Du Toit 2008 which is shortly summarized below:

A function ϕ for which $\phi(x) = \phi(\|x\|)$ is true is called *radial*. Now to be able to interpolate, we need to find the interpolation function $s(x)$ which is the same as the given values p_i in all points.

$$s(x_i) = p_i, \quad i = 1, 2, \ldots, n \tag{4.1}$$

The RBF interpolation now consists of a linear combination of $\phi(\|x - x_i\|)$ for a chosen radial function ϕ with n constants λ_i.

$$s(x) = \sum_{i=1}^{n} \lambda_i \phi(\|x - x_i\|) \tag{4.2}$$

$$p_j = \sum_{i=1}^{n} \lambda_i \phi(\|x_j - x_i\|), \quad j = 1, 2, \ldots, n \tag{4.3}$$

Therefore, this can be written as a linear matrix equation:

$$\begin{bmatrix} \phi(\|x_1 - x_1\|) & \phi(\|x_2 - x_1\|) & \dots & \phi(\|x_n - x_1\|) \\ \phi(\|x_1 - x_2\|) & \phi(\|x_2 - x_2\|) & \dots & \phi(\|x_n - x_2\|) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\|x_1 - x_n\|) & \phi(\|x_2 - x_n\|) & \dots & \phi(\|x_n - x_n\|) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \tag{4.4}$$

or simply

$$\Phi\lambda = p \tag{4.5}$$

with Φ being a symmetric $n \times n$ matrix as $\|x_j - x_i\| = \|x_i - x_j\|$. There are many possibilities for the radial basis function $\phi(r)$. It can be for example linear (r), gaussian (e^{-r^2}) or multiquadric ($\sqrt{\left(\frac{r}{\epsilon}\right)^2 + 1}$) with ϵ being a constant that defaults to the approximate average distance between nodes.

As an example, consider the three points $x_1 = 0$, $x_1 = 3$ and $x_1 = 5$ with $p(x_1) = 0.2$, $p(x_2) = 0.8$ and $p(x_3) = 0.1$ and choose a gaussian function for ϕ to get the following:

$$\begin{bmatrix} \phi(0) & \phi(3) & \phi(5) \\ \phi(3) & \phi(0) & \phi(2) \\ \phi(5) & \phi(2) & \phi(0) \end{bmatrix} \lambda = \begin{bmatrix} 1 & 1.23 \times 10^{-4} & 1.39 \times 10^{-11} \\ 1.23 \times 10^{-4} & 1 & 1.83 \times 10^{-2} \\ 1.39 \times 10^{-11} & 1.83 \times 10^{-2} & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} 0.22 \\ 0.8 \\ 0.1 \end{bmatrix} \tag{4.6}$$

Solving this linear matrix equation using `numpy.linalg.solve` gives us the solution for λ:

$$\lambda = \begin{bmatrix} 0.200 \\ 0.798 \\ 0.085 \end{bmatrix} \tag{4.7}$$

Combined we get the following linear combination for the interpolated function $s(x)$ (Figure 4.4a):

$$s(x) = 0.200\phi(\|x\|) + 0.798\phi(\|x - 3\|) + 0.085\phi(\|x - 5\|) \tag{4.8}$$

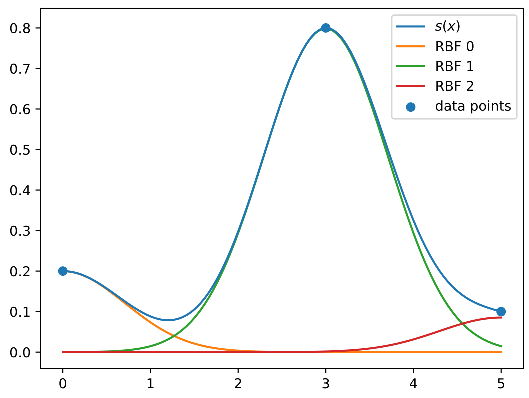
Applying the same method to a list of random points allows to interpolate their values for arbitrary other points like the green point on the sinus-like curve in Figure 4.4b. This can also be trivially extended in m dimensions by replacing x with an $x \in \mathbb{R}^m$ and using a norm in $\mathbb{R}^m$ for $\| \ \|$.

4.2.2. Implementation

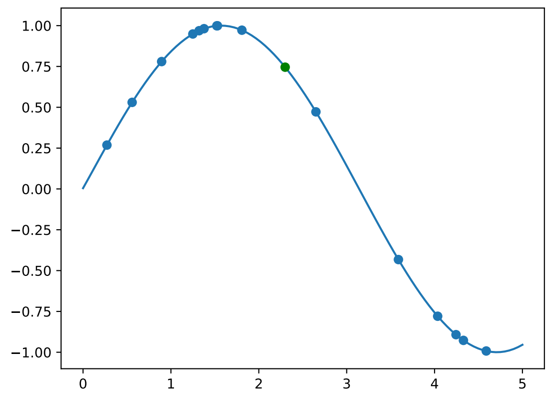
The scipy function `scipy.interpolate.Rbf` allows directly interpolating a value similar to `griddata` in Section 4.1.2 while using the linear function as the Radial Basis Function ($\phi(r) = r$). A difference in usage is that it only allows interpolating a single value, but as it is pretty quick it is possible to calculate multiple values sequentially.

4.2.3. Results

The results from RBF interpolations can be seen in Figure 4.5. It is far smoother with a gradient from 0 % to 100 % from the top left to the bottom right corner. Only the lower mass (Figure 4.5a) has a view outliers. Unlike griddata it is also possible to extrapolate to close values and still get realistic results.

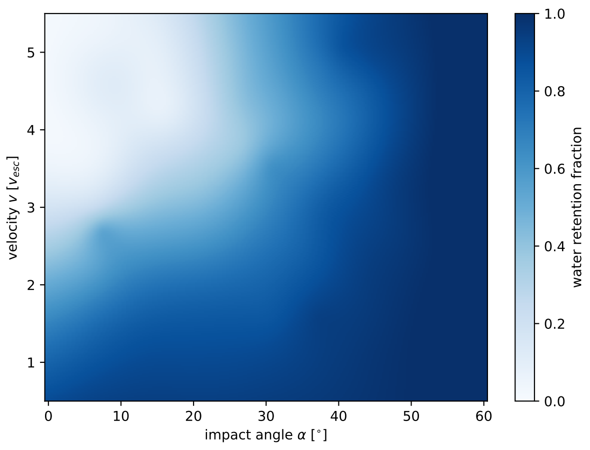


(a) The three functions making up the RBF interpolation from Equation (4.8)

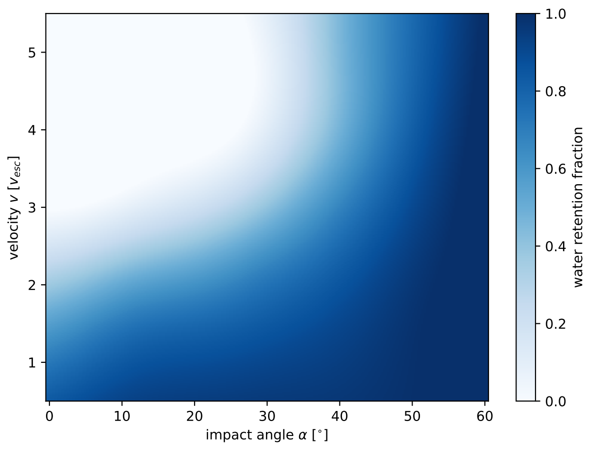


(b) 15 points following a sinus-like function with one interpolated value (●)

Figure 4.4.: Two examples for simple RBF interpolation in one dimension



(a) $m_{total} = 10^{22}$, $\gamma = 0.6$, $wt = wp = 0.15$



(b) $m_{total} = 10^{24}$, $\gamma = 0.6$, $wt = wp = 0.15$

Figure 4.5.: Interpolation result using Radial Basis Functions

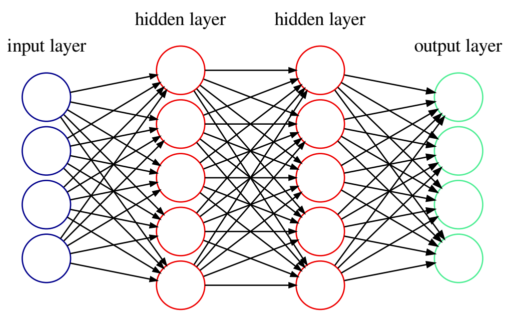
4.3. Artificial Neural Networks

Another method that is good at taking pairs of input and output values and then able to predict the output for arbitrary input sets is using *Artificial neural networks* (ANNs).

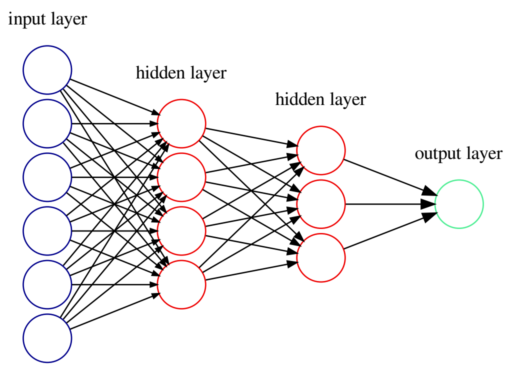
4.3.1. Theory

The idea behind artificial neural networks is trying to emulate the functionality of neurons by having nodes that are connected to each others. The weights w of these connections are modified during the training to represent the training data and can then be used to predict new results for input values not seen in the training data.

Every neural network needs an input layer with as many nodes as input parameters and an output layer with a node for every output value. In between, there can be multiple hidden layers with an



(a) An example for a neural network



(b) The network used for interpolation

Figure 4.6.

arbitrary number of nodes. (Figure 4.6a)

If we first only consider a single neuron, then on every iteration it calculates the sum over all input values multiplied with their weight w. Afterwards an activation function g is applied to the sum z to get the prediction $\hat{y}$.

$$z = \sum_i w_i x_i \qquad \hat{y} = g(z) \tag{4.9}$$

The non-linear activation function allows the network to be able to approximate all types of functions instead of being just a linear function itself. Popular activation functions are the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and the ReLU function (*rectified linear unit*, $f(x) = \max(0, x)$).[12]

After this first step (the *feedforward*) is done, the weights can be modified by comparing the prediction with the real output (the *backpropagation*). The function that describes the error between them is called the Loss function and one possible form is the mean squared error function:

$$L(\hat{y}, y) = \sum_i (\hat{y}_i - y_i)^2 \tag{4.10}$$

To update the weights, the derivative of the Loss function with respect to the weights is calculated and added to the existing weights.[13]

4.3.2. Implementation

As building a neural network from scratch gets complex very quickly, it is easier to use `Keras`[14] which provides easy to use high-level functions over the calculations provided by `TensorFlow`[15]. To

[12]Skalski 2018.
[13]Loy 2018.
[14]`https://keras.io`
[15]`https://www.tensorflow.org/`

build our network, we only need to specify the structure of the layers, take our input and let the network train for 200 epochs (iterations of feedforward and backpropagation).

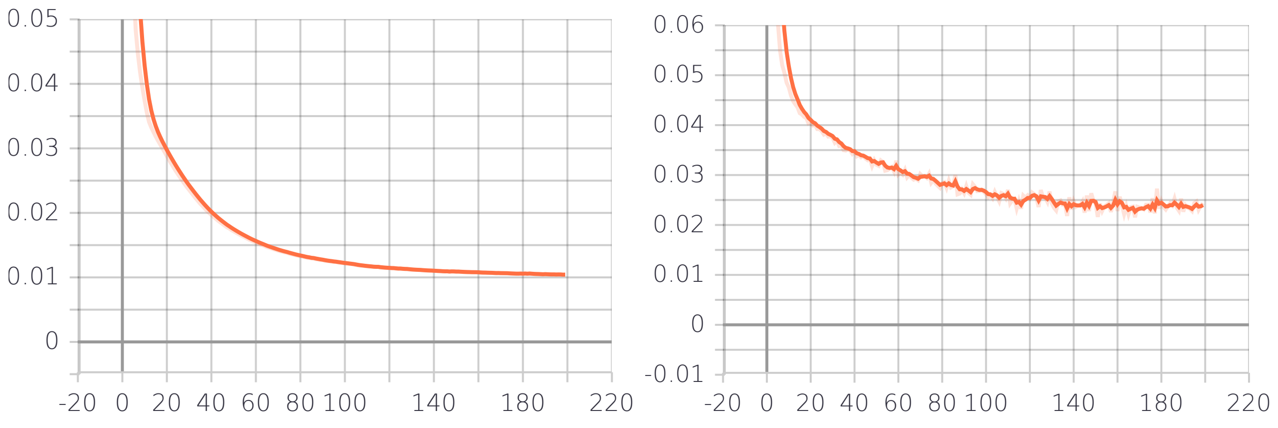
The network needs six nodes in the input layer for the input parameters and one node in the output layer for the prediction. In between, are two layers with decreasing numbers of nodes as this seems to give the best results. (Figure 4.6b)

```
from keras import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(6, input_dim=6, activation='relu'))
model.add(Dense(4, kernel_initializer='normal', activation='relu'))
model.add(Dense(3, kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal', activation="sigmoid"))
model.compile(loss='mean_squared_error', optimizer='adam')

model.fit(x, Y, epochs=200, validation_data=(x_test, Y_test))
```

Listing 4.1: The used model as Keras code

4.3.3. Training

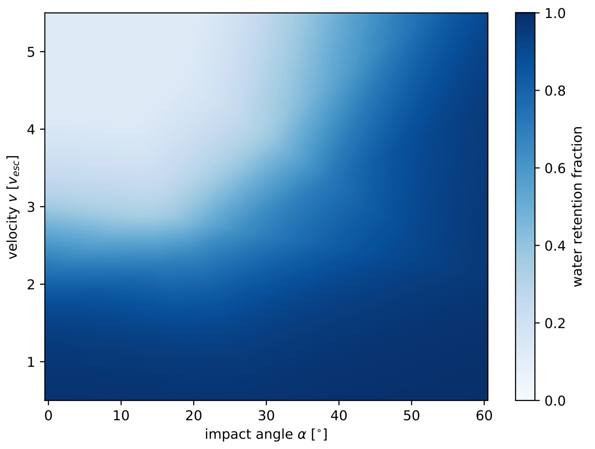
To find the ideal parameters to use, the simulation data (excluding the data from Chapter 5) is split into two groups: The complete original set of simulations and 80 % of the new simulation set is used to train the neural network while the remaining 20 % are used for validation. This means that after every epoch the loss function is not only calculated for the training data, but also for the separate validation data (Figure 4.7). Finally, the model with the lowest loss on the validation data set was chosen (Listing 4.1).



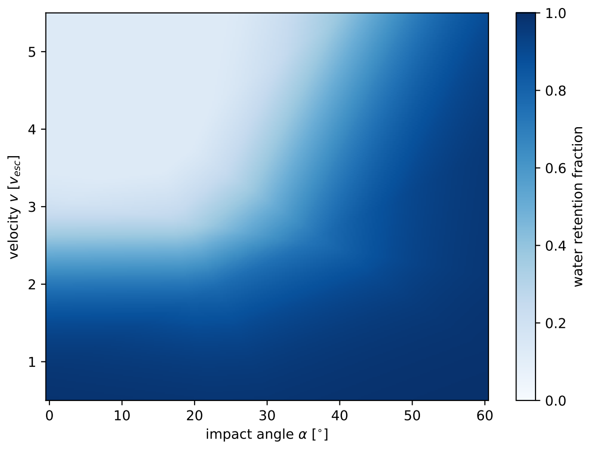
(a) loss function on the training data (b) loss function on the validation data

Figure 4.7.: During training the loss function (mean squared error) decreases with every epoch until it converges to a final value.

After the training, the resulting model is saved in a small `HDF5` file which can be used to evaluate the model very quickly (about 100 ms for 10 000 interpolations).



(a) $m_{total} = 10^{22}$, $\gamma = 0.6$, $wt = wp = 0.15$



(b) $m_{total} = 10^{24}$, $\gamma = 0.6$, $wt = wp = 0.15$

Figure 4.8.: Interpolation result using the trained neural network

4.3.4. Results

The output of the Neural Network (Figure 4.8) looks quite similar to the RBF output. There seem to be very few details visible, but just like in the other results the transition from high water fraction to low water fraction is clearly visible.

	mean squared error	mean error
griddata (only original data)	0.014	0.070
neural network	0.010	0.069
RBF	0.008	0.057
griddata	0.005	0.046

Table 4.1.: Prediction accuracy for the different interpolation methods

5. Comparison and Conclusion

All three methods for interpolation described above give results that follow the rough correlations from Section 3.1. So to compare them more precisely and measure their accuracy, an additional set of 100 simulations (with the same properties as the ones listed in Section 2.5) was created. These results are neither used to train or select the neural network, nor are in the dataset for griddata and RBF interpolation. Therefore, we can use them to generate predictions for their parameters and compare them with the real fraction of water that remained in those simulations. By taking the mean absolute difference and the mean squared error between the predictions and the real result, the accuracy of the different methods can be estimated (Table 4.1). As one of these parameter sets is outside the convex hull of the training data and griddata can't extrapolate, this simulation is skipped and only the remaining 99 simulations are considered for the griddata accuracy calculation.

Of the three methods, the trained neural network has the highest mean squared error. This seems to be at least partly caused by the fact that during training of the neural network, the data is strongly generalized, causing the final network to output the "smoothest" interpolations. While this causes the errors to be higher, it might be possible that the fine structured details in the simulation output are just an artifact of the simulation setup and doesn't represent real world collisions.

Another important aspect to compare is the interpolation speed. The neural network is able to give the 100 results in about 4 ms (after loading the trained model which takes approximately one second). RBF interpolation is still reasonably fast, taking about 8.5 s (85 ms per interpolation). But as `griddata` expects a grid-based parameter space, it becomes really slow when adding the resimulation data with random parameters. A single interpolation takes about 35 s totalling to about an hour for all 99 test cases. Using only the original dataset brings the run time down to around 10 s, but causes the results to be less accurate than all other methods. (first row in Table 4.1)

Interpolation using Radial Basis Functions all in all seems to be the most reliable method if there is enough input data and this input data is mostly spread randomly across the parameter space. It is easy to implement and quite fast to execute while still giving reasonable results. Neural Networks can also provide realistic output, but have lots more configurable parameters that need to be tuned to get usable results. Their main advantage would be more noticeable if the input set was by magnitudes larger. In this case only the training would take longer, while evaluating the trained model wouldn't change.

To sum up, it is possible to estimate the amount of water lost in two-body-collisions with arbitrary collision parameters by simulating the outcome of a large amount of collisions using `SPH` and then doing linear interpolations to get results for parameters in between the ones from the simulation set. While the amount of remaining water is overestimated in this analysis as thermal effects during the collision are ignored, the results are better than a perfect merging assumption.

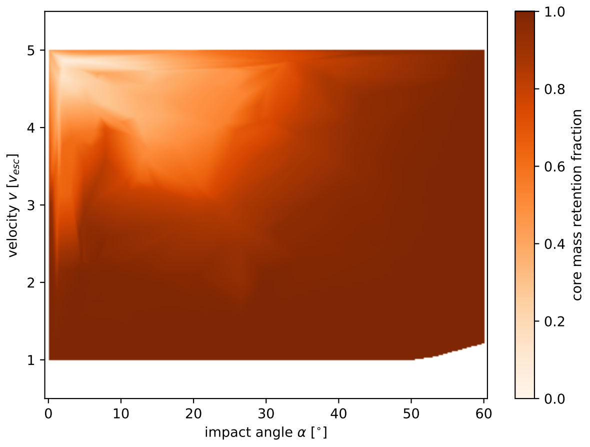
A. Core mass retention

While this thesis focuses on the water retention after the collisions, the same methods can be applied to the fraction of basalt from the core of the two bodies that remains after the collision. Using the same parameters for interpolation and a separately trained model with the same parameters results in similar results as for water retention. When plotting the same scenarios as in the previous chapters (Figure A.1), one can see that the results are quite similar. The main difference is that on average there is a slightly higher core mass retention, which can be explained by the fact that weaker collisions might be strong enough to throw the outer water layer into space, but keep the core intact. In addition, it seems like the transition between high and low core mass retention is faster.

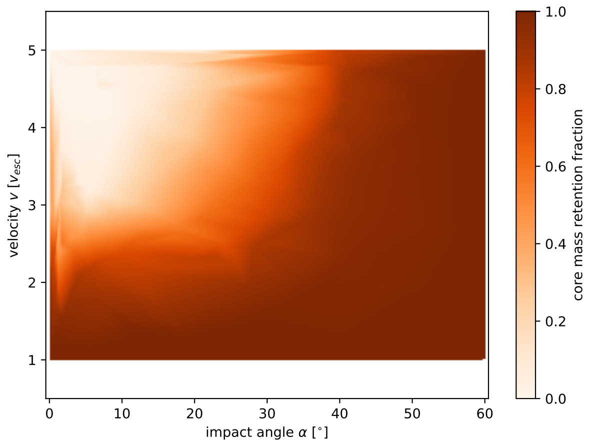
When applying the same comparison as described in Chapter 5 the interpolations seem to have a lower accuracy, but still RBF interpolation gives the best results considering the slow speed of griddata.

	mean squared error	mean error
neural network	0.043	0.167
RBF	0.032	0.149
griddata	0.041	0.169

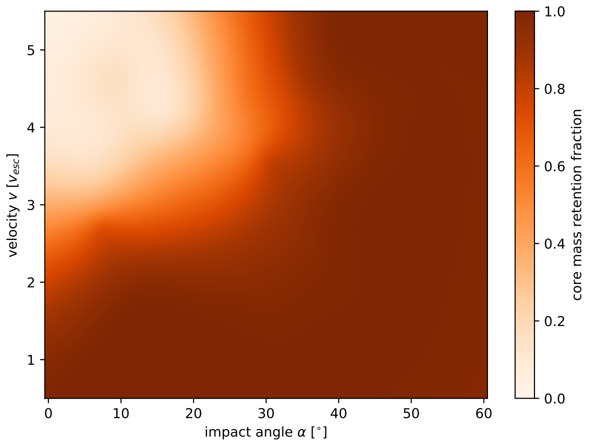
Table A.1.: prediction accuracy for the different interpolation methods



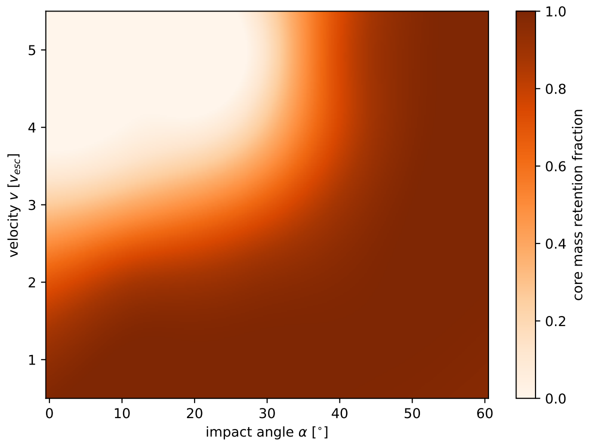
(a) Griddata with $m_{total} = 10^{22}$



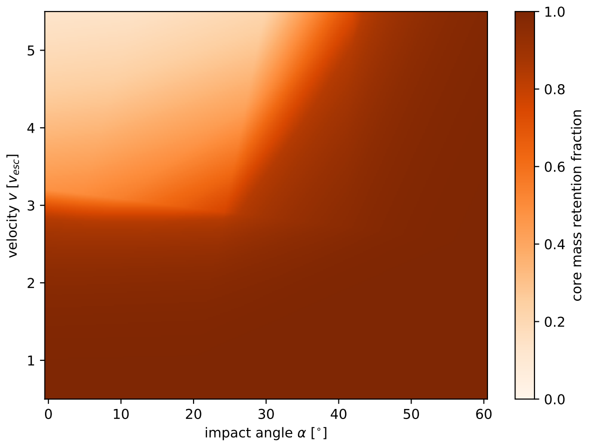
(b) Griddata with $m_{total} = 10^{24}$



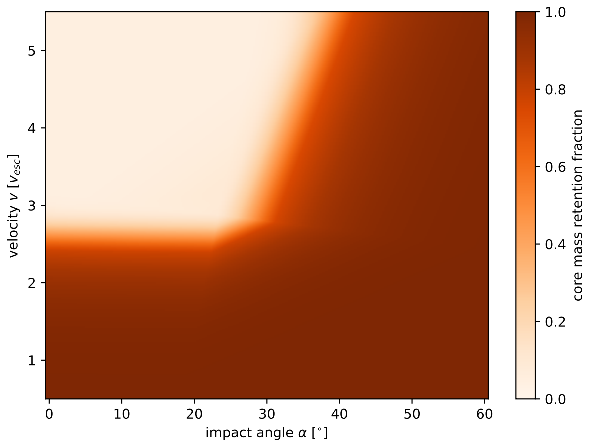
(c) RBF with $m_{total} = 10^{22}$



(d) RBF with $m_{total} = 10^{24}$



(e) Neural Network with $m_{total} = 10^{22}$



(f) Neural Network with $m_{total} = 10^{24}$

Figure A.1.

Bibliography

Burger, C., T. I. Maindl, and C. M. Schäfer (Jan. 2018). "Transfer, loss and physical processing of water in hit-and-run collisions of planetary embryos." In: *Celestial Mechanics and Dynamical Astronomy* 130.1, 2, p. 2. DOI: 10.1007/s10569-017-9795-3. arXiv: 1710.03669 [astro-ph.EP] (cit. on p. 5).

Delaunay, Boris (1934). "Sur la sphère vide. A la mémoire de Georges Voronoï." French. In: *Bull. Acad. Sci. URSS* (6), pp. 793–800. URL: http://mi.mathnet.ru/eng/izv4937 (cit. on p. 8).

Dorninger, Berndt (2019). "Realistic physical collision model in planet formation. Benchmark of GPU-environments." Bachelor's Thesis (cit. on p. 5).

Du Toit, Wilna (Mar. 2008). "Radial basis function interpolation." MA thesis. Stellenbosch: Stellenbosch University. URL: https://core.ac.uk/download/pdf/37320748.pdf (cit. on p. 10).

Dvorak, Rudolf, Siegfried Eggl, et al. (Aug. 2012). "Water delivery in the early Solar System." In: *American Institute of Physics Conference Series*. Ed. by Marko Robnik and Valery G. Romanovski. Vol. 1468. American Institute of Physics Conference Series, pp. 137–147. DOI: 10.1063/1.4745576. arXiv: 1506.01851 [astro-ph.EP] (cit. on p. 3).

Dvorak, Rudolf, Birgit Loibnegger, and Thomas I. Maindl (June 2015). "On the probability of the collision of a Mars-sized planet with the Earth to form the Moon." In: *arXiv e-prints*, arXiv:1506.09043, arXiv:1506.09043. arXiv: 1506.09043 [astro-ph.EP] (cit. on p. 4).

Loy, James (May 14, 2018). *How to build your own Neural Network from scratch in Python*. Towards Data Science. URL: https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6 (visited on 07/16/2019) (cit. on p. 13).

Maindl, T. I. et al. (Mar. 2017). "Collisional water transport and water-loss relevant to formation of habitable planets." In: *Proceedings of the First Greek-Austrian Workshop on Extrasolar Planetary Systems*, pp. 137–153 (cit. on pp. 3, 4).

Maindl, Thomas I. and Rudolf Dvorak (Jan. 2014). "Collision parameters governing water delivery and water loss in early planetary systems." In: *Exploring the Formation and Evolution of Planetary Systems*. Ed. by Mark Booth, Brenda C. Matthews, and James R. Graham. Vol. 299. IAU Symposium, pp. 370–373. DOI: 10.1017/S1743921313008971. arXiv: 1307.1643 [astro-ph.EP] (cit. on p. 4).

Martin, Rebecca G. and Mario Livio (Sept. 2012). "On the evolution of the snow line in protoplanetary discs." In: *MNRAS* 425.1, pp. L6–L9. DOI: 10.1111/j.1745-3933.2012.01290.x. arXiv: 1207.4284 [astro-ph.EP] (cit. on p. 3).

Schäfer, C. M. et al. (Aug. 2019). "A versatile smooth hydrodynamics code for graphics cards." In: *Computers and Mathematics with Applications*, submitted (cit. on p. 4).

Schäfer, C. et al. (May 2016). "A smooth particle hydrodynamics code to model collisions between solid, self-gravitating objects." In: *A&A* 590, A19. DOI: 10.1051/0004-6361/201528060. arXiv: 1604.03290 [astro-ph.EP] (cit. on p. 4).

Skalski, Piotr (Aug. 17, 2018). *Deep Dive into Math Behind Deep Networks*. Towards Data Science. URL: https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba (visited on 07/16/2019) (cit. on p. 13).

Stewart, Sarah T. and Zoë M. Leinhardt (May 2012). "Collisions between Gravity-dominated Bodies. II. The Diversity of Impact Outcomes during the End Stage of Planet Formation." In: *ApJ* 751.1, 32, p. 32. DOI: 10.1088/0004-637X/751/1/32. arXiv: 1109.4588 [astro-ph.EP] (cit. on p. 3).