Permeability estimation of pore-scale images through a Convolutional Neural Net

A 3D Proof-of-concept

Author
Simon Penninga (1256017)
s.w.penninga@student.tue.nl

Supervisor
Prof.dr.ir. J.M.V.A. Koelman

April 6, 2021

Eindhoven University of Technology
Department of Applied Physics

Abstract

A Monte-Carlo method of generating binary porescale images is used to create 2.4×10^6 samples and their permeabilities are calculated through a two-step pressure-field calculation. A convolutional neural network is created and trained on these samples for permeability estimation. The net decreases computation time by 4 orders of magnitude at the expense of 13% accuracy. The biggest loss is found for low permeability samples due to the information loss of tight pore-necks during convolution. The lower permeability samples are the least important in application and training with an alternative error metric is suggested. Improvements that should still be investigated include hard-wiring symmetry recognition into the neural net, further parameter optimization and adaptations to the custom loss function that biases the training process.

Contents

1 Introduction

The porosity of a material is an important property that impacts many fields of research. The fields of influence and application include for example food degradation [1], medicine tablet disintegrability [2], soil structure characterization [3] and geological admittance of CO_2 storage [4]. This report focuses on the latter, where the porosity interconnectivity via narrow pore throats determines the ease at which fluids can be injected into the subsurface. The driving capabilities in 3D pore-scale modeling of reservoir rocks have been techniques like micron-scale and submicron-scale X-ray microcomputed tomography(CT) [5] and the statistical reconstruction from two-dimensional images [6].

The 3D scan or 3D reconstruction of the porous material is converted to a binary image in which each voxel contains a Boolean value that represents it being either a permeable void or a non-permeable material. These pore-scale images allow for estimations of the permeability, sometimes called fluid admittance, which will be investigated in this report. As this report focusses on establishing a proof-of-concept, these geometry images are generated randomly through the use of White Gaussian Noise (WGN) and will be used both for training and validation. This process is described in more detail in section 2.

The goal of this report is now to extract the permeability from these 3D binary pore-scale images using a Convolutional Neural Network(CNN). The calculation of the permeability is computationally expensive and the number of voxels in a single image grows cubically with respect to the resolution of a tomography scan. The computational algorithms that determine the permeability vary widely in the complexity of geometries that they can handle; for example, some of these models capture the topological connectivity [7], others use the lattice Boltzmann method [8]. This paper uses the so called 'two-step method' [9] as a reference, in which the local permeability of each voxel is first calculated, followed by a pressure-field calculation that gives rise to a value for the total permeability. This calculation is explained in depth in section 3.

The hypothesis is such that a CNN is able to recognize the relations between pore-scale morphology and it's permeability. This includes relations between the permeability and the porosity, pore-neck constraints, rotational invariance and symmetry within the media. Machine-learning concepts behind the design decisions of the CNN are explained in section 4 after which the final network is evaluated in section 5 with respect to speed, computational efficiency and accuracy. This section also contains the limitations uncovered and gives suggestions for improvements and further research.

2 Geometry generation

To train a convolutional neural network, first data has to be generated. This is done through the so called 'white noise method' [10], in which first a 3D cube of voxels is generated. Each voxel in the structure is given a random value between 0 and 1 according to a random flat distribution.

The resulting cube is filtered by a 3D averaging filter in a $3x3x3$ kernel according to

$$G(x,y,z) = g_1 R_{x,y,z} + g_2 R_{x+1,y+1,z+1} + g_3 R_{x,y+1,z+1} + ... + g_{27} R_{x-1,y-1,z-1} \tag{1}$$

where R represents the original value that was given to the voxel and g_1 to g_{27} represent weights that follow the 3D-Gaussian distribution (2).

$$g(x,y,z) = \frac{1}{(2\pi\sigma^2)^{3/2}} e^{-\frac{x^2+y^2+z^2}{2\sigma^2}}. \tag{2}$$

In this equation, the weight g decreases for the voxels that are further away from the central voxel of the kernel $R_{x,y,z}$. This voxel is chosen as zero-point and the values x,y,z represent the distance from this point. The standard deviation σ is a value that determines the rate of descent of the

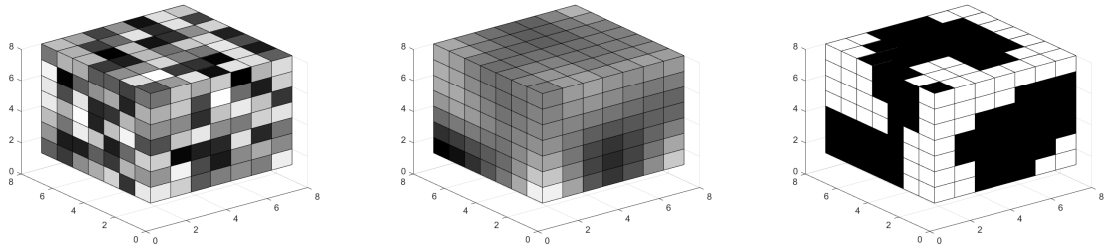
weights when moving away from the central voxel. The kernel moves over all voxels in the image such that the values are smoothed with respect to their 26 nearest neighbours.

The values of the voxels are now clustered by this averaging filter after which a cut-off filter f binarizes the result through

$$f(x) = \begin{cases} 1 & \text{if } x \geq t \\ 0 & \text{if } x < t \end{cases}.$$

Values that lie above a certain threshold are set to 1, acting as porous space. Those under the threshold are set to 0, acting impermeable during computation.

Snapshots of a geometry sample during this process are shown in Fig. 1. While these images show 2D-grid projections in 3D, it should be noted that all the voxels and filters are in fact 3-dimensional.



(a) *Generated noise* ***(b)*** *3D Gaussian filtered* ***(c)*** *Binarization after filter*

Figure 1: *Visualisation of the three steps of geometry generation.*

A benefit of using this method is the ability to rapidly generate voxel structures of any size that differ greatly in shape. Varying the standard deviation of the Gaussian filter (Fig. 1b) allows one to change the cluster size of voxels and thus change the pore-space to best represent the material that is to be imitated. Varying the cut-off value of the second filter (Fig. 1c) allows for a variance in porosities that are generated. A high cut-off value gives rise to a very low porosity or 'void fraction' and vice versa.

For this paper the dimensions 32x32x32 are chosen for the structure as it strikes a balance between the structure's complexity and available computational power at the time. To best represent rocky materials, a minimum pore diameter of roughly 5 voxels and a porosity range of 10-35% should be achieved. To realise this a standard deviation σ of 1.5 was used for weights of the Gaussian filter in (2) and a cut-off range of $0.511 \leq t \leq 0.534$ was used for the cut-off filter. An algorithm is used to exclude samples that do not have at least 1 set of opposing faces connected through a continuous open pore-space as these samples could hinder the training process of the neural net. With these parameters 10^5 samples were generated and their porosity distribution is shown in Fig. 2.

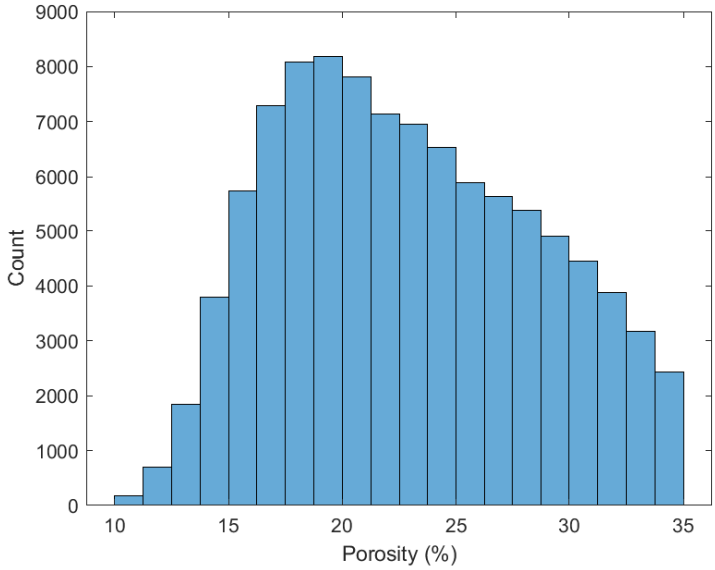


Figure 2: *Histogram that shows the porosity of the 10^5 samples that were generated.*

Something that should be addressed is the small number of samples with porosity between 10-15%. This can be attributed to the stochastic nature of the generation process; with a sample being accepted only when there is a connection between two of the cube's opposing faces, this decreases the probability of a low-porosity sample being accepted. Nevertheless, some of the low-porosity samples are accepted and will impact the performance of the CNN model that is created in section 4. To give an example of what the final porous structure looks like, the permeable pore-space of a random sample is shown in Fig. 3.

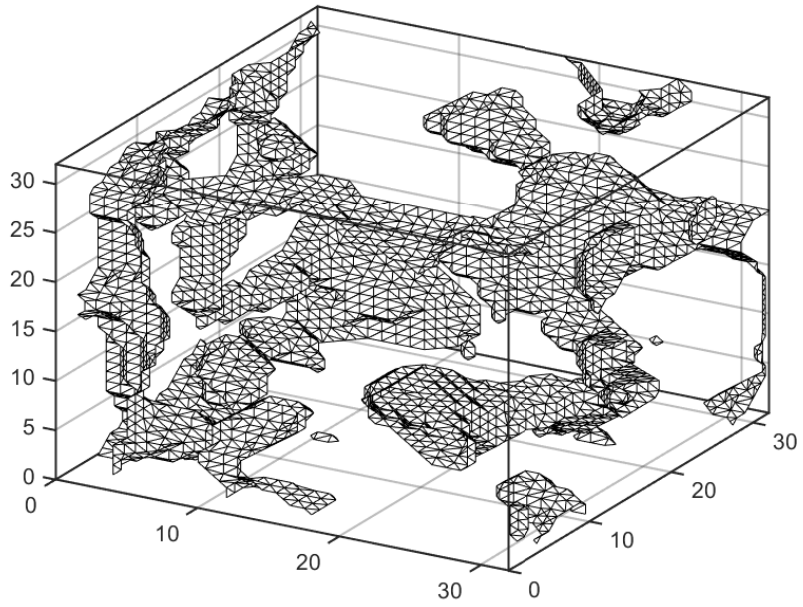


Figure 3: *Schematic view of the open voxels in a 32x32x32 cube.*

3 Two-step permeability calculation

As was stated in the introduction, the method used for calculating the permeability of the geometry samples is a two-step method that uses local voxel-connectivity and a pressure gradient [9].

3.1 Local connectivity

The goal of the first step of this method is to determine a local conductivity value for every individual voxel. For modeling single-phase steady-state incompressible flows in a porous media, the Navier-Stokes equation reduces to the steady-state Stokes equation:

$$\mu \nabla^2 u = \nabla P \tag{3}$$

with the viscosity μ, the velocity u and pressure P. Mass conservation reduces to the incompressibility equation

$$\nabla \cdot u = 0. \tag{4}$$

We rewrite the local velocity as a Darcy flow velocity proportional to the local pressure with a local conductivity scalar κ and inversely proportional to viscosity μ following

$$u = -\frac{\kappa}{\mu} \nabla P. \tag{5}$$

Combining (3) and (5) and using a 3D version of the lubrication approximation gives a constraint for the local conductivity scalar values

$$-\nabla^2 \kappa = 1. \tag{6}$$

This constraint constitutes the first step and allows for calculation of local 3D voxel connectivity. Application of this constraint to individual voxels is done through the discretization steps:

$$\nabla^2_{x,y,z} \kappa = \frac{\partial^2 \kappa}{\partial x^2} + \frac{\partial^2 \kappa}{\partial y^2} + \frac{\partial^2 \kappa}{\partial z^2},$$

$$\frac{\partial^2 \kappa}{\partial x_i^2} = \frac{\kappa_{i+1} - 2\kappa_i + \kappa_{i-1}}{h^2} + \mathcal{O}(h^2),$$

$$\nabla^2_{x,y,z} \kappa = \frac{\kappa_{x+1,y,z} - 2\kappa_{x,y,z} + \kappa_{x-1,y,z}}{h^2} + \frac{\kappa_{x,y+1,z} - 2\kappa_{x,y,z} + \kappa_{x,y-1,z}}{h^2} + \frac{\kappa_{x,y,z+1} - 2\kappa_{x,y,z} + \kappa_{x,y,z-1}}{h^2} + \mathcal{O}(h^2) \tag{7}$$

where for every voxel the 6 nearest neighbours are taken into account and diagonal flows are disregarded. The variable h is a measure of distance, in this case 1 voxel. This gives for (6)

$$(-\nabla^2 \kappa)_{x,y,z} = \frac{1}{h^2}(6\kappa_{x,y,z} - \kappa_{x+1,y,z} - \kappa_{x-1,y,z} - \kappa_{x,y+1,z} - \kappa_{x,y-1,z} - \kappa_{x,y,z+1} - \kappa_{x,y,z-1}) + \mathcal{O}(h^4) = 1 \tag{8}$$

and finally

$$\kappa_{x,y,z} = \frac{1}{6}(h^2 + \kappa_{x+1,y,z} + \kappa_{x-1,y,z} + \kappa_{x,y+1,z} + \kappa_{x,y-1,z} + \kappa_{x,y,z+1} + \kappa_{x,y,z-1}) + \mathcal{O}(h^4) \tag{9}$$

as an expression for local connectivity. This final equation can now be translated to a set of NxN linear equations in which N represents the number of porous voxels in the structure. A sparse linear solver should be used to solve the standard equation

$$A\kappa = h^2 \tag{10}$$

where A is the NxN connectivity matrix that has a value of 6 on all the diagonals and -1 for the off-diagonal elements that represent a connection to a neighbouring voxel. Voxels at an edge of the cube should have their diagonal element reduced by 1 for every neighbour that is past the boundary of the sample. The vector κ represents the local connectivity. When κ is expressed in units h^2 equation (10) reduces to

$$A\kappa = 1. \tag{11}$$

As an example, the matrix and vectors are filled in below for at least 8 voxels.

$$\begin{bmatrix} 3 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & \dots & 0 \\ -1 & 4 & 0 & -1 & 0 & -1 & -1 & 0 & \dots & 0 \\ 0 & 0 & 4 & -1 & -1 & -1 & -1 & 0 & \dots & 0 \\ 0 & -1 & -1 & 5 & -1 & -1 & -1 & 0 & \dots & 0 \\ -1 & 0 & -1 & -1 & 4 & -1 & 0 & 0 & \dots & 0 \\ -1 & -1 & -1 & -1 & -1 & 6 & -1 & 0 & \dots & 0 \\ 0 & -1 & -1 & -1 & 0 & -1 & 4 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 6 \end{bmatrix} \begin{bmatrix} \kappa_1 \\ \kappa_2 \\ \kappa_3 \\ \kappa_4 \\ \kappa_5 \\ \kappa_6 \\ \kappa_7 \\ \kappa_8 \\ \vdots \\ \kappa_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

From the first diagonal element it can be seen that this voxel is in one of the corners of the cube, as it has a diagonal value reduced by 3. Permeable voxels 2 and 3 are on the edges of the cube because they have diagonal element 4. Voxel 1 and 2 are connected since they have $A_{12} = A_{21} = -1$. Regardless of whether a voxel is in the interior or at the boundary of the geometry, the row and column of it's cell add up to 0. Matrix A looks densely populated considering only a $8x8$ set of connections is written out. In reality this is a very sparse matrix, since every row or column can have at most 7 entries while the length and width of the matrix are generally 4 orders of magnitude larger. Most of the voxels are in the interior of the structure and most of the diagonal elements are therefore 6. Lower values are depicted here due to the size constraint.

Before this calculation, a layer of porous voxels with a thickness of 1 is added to two opposing faces of the cube to create a $32x32x34$ structure. This is done to change k values to include boundary conditions that will be used for the pressure calculation in section 3.2. These two layers are added on three occasions, once per direction $[x, y, z]$, and are removed after the pressure field calculation.

The results of this calculation for a random sample are shown in Fig. 4 where it becomes clear that the connectivity value can be seen as a measure of distance to the nearest impermeable voxels. Note that this image is also shown in 2.5D and that the open space perpendicular to the projection contributes heavily to the connectivity value.

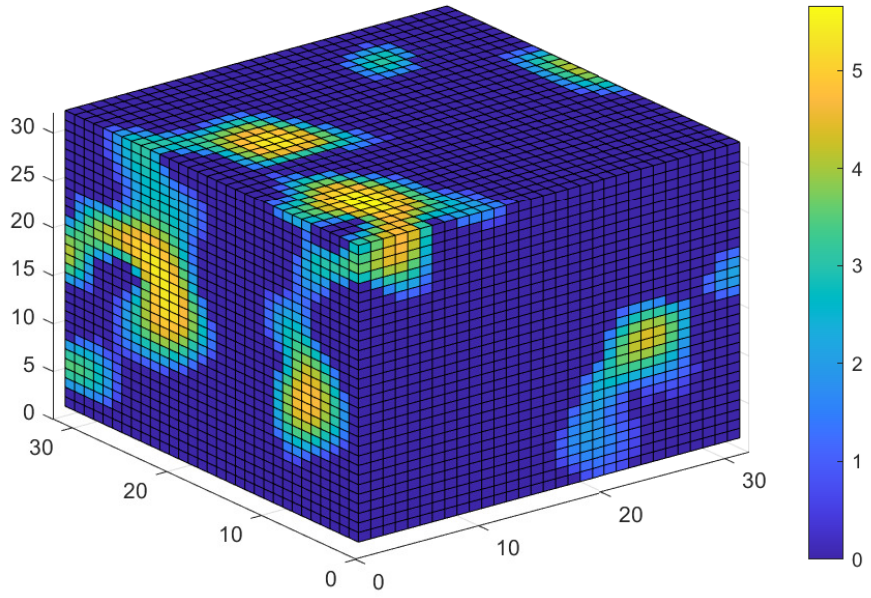


Figure 4: *2.5D projection of the κ-values of a random sample.*

3.2 Pressure grid

The second step of this method involves imposing a flow-condition. Inserting the velocity field of (5) into the incompressibility equation (4) yields

$$\nabla \cdot (\kappa \nabla P) = 0 \tag{12}$$

with the assumption that the viscosity μ is constant for Newtonian incompressible and isothermal fluids. Discretizing this equation equates to demanding a balance between in- and outflow

$$\sum_j -k_{i,j} \Delta P_{i,j} = 0 \tag{13}$$

where ΔP is the difference in pressure between two voxels and k the transmissibility between voxels. This transmissibility $k_{i,j}$ is different from the connectivity κ from the previous step and the two should not be confused. The k can be calculated as a series conductance

$$k_{i,j} = \begin{cases} \dfrac{1}{\frac{1}{\kappa_i} + \frac{1}{\kappa_j}} & \text{if i and j are neighbours} \\ 0 & \text{otherwise} \end{cases} . \tag{14}$$

A single voxel now carries 6 local connectivity values, one for each face. These transmissibility values determine the pressure field through the geometry. The calculation for the pressure field is done through another set of linear equations similar to (11), namely

$$k_{eff} P = 0 \tag{15}$$

where k_{eff} is a matrix containing the transmissibility values determined in (14) and P a vector containing the local pressure of each voxel. Because this equation does not enable any calculation by itself, boundary conditions need to be imposed. These boundary conditions introduce a flow

between two opposing cube faces such that a pressure gradient emerges. The amplitude of the flow can be chosen arbitrarily since it will not affect the final value of the structure's permeability as this value will be normalized by the same pressure gradient. The boundary conditions are set at the 2 additional layers that were added to the structure in the previous step where the $32x32x32$ samples was expanded to $32x32x34$. The calculation of (15) including boundary conditions is shown below

$$\begin{bmatrix} k_{1,-1} & k_{11} & 0 & k_{13} & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & k_{22} & k_{23} & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & k_{31} & k_{32} & k_{33} & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{44} & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{55} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & k_{N-1,N-1} & k_{N-1,N} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & & k_{N,N-1} & k_{NN} & k_{N,N+1} \end{bmatrix} \begin{bmatrix} 33 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ \vdots \\ P_{N-1} \\ P_N \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $k_{13} = k_{31}$ is the transmissibility between voxels 1 and 3 and N is the number of voxels. There is no connection between voxels 1 and 2 which shows in the matrix as $k_{12} = k_{21} = 0$. The enumeration of the voxels is not representative of their position in the pore-space nor is it important as long as connectivity is correctly described through (14). Just like in the connectivity matrix, all the transmissibility values in a row add up to 0. The diagonal value, for example k_{11}, is defined as $k_{ii} = \sum -k_{ij}$ where the connection to the added layer $k_{1,-1}$ is also included.

In this example only one voxel per added layer is visible, k_{-1} and k_{N+1} are displayed to show their effects in the equation. The first and last column of the matrix have k-values $k_{1,-1}$ and $k_{N,N+1}$ that represent the connection between the out-most layers of the $32x32x34$ structure and the inner core. These voxels have set pressures of 33 and 0 respectively as is visible in the P vector. Only a single voxel per side is displayed here, but in practice this means that 32^2 columns are added to the left and right of the matrix. The pressure is expanded by 32^2 times the value 33 at the top, and 32^2 times 0 on the bottom. These voxels now introduce a net flux from the voxel with a pressure of 33 to the voxel with 0 pressure. After the calculation is done and P_1 to P_N are found, the added columns and values can be discarded to make way for a new set of boundary conditions that are applied to a different plane of the cube.

Another way this calculation can be done is by moving the added layers directly to the right of the equation, creating a boundary vector. The pressure vector P contains only the unknown pressures and the boundary vector contains only the transmissibility values from the added layers to the core, multiplied with a scalar. The 3 calculations with boundary conditions along the cube faces x,y and z then only differ in the boundary vector b. For the calculation with a pressure gradient in the x direction, the layers are added on top of the two faces in the yz plane and the k-values of the voxels in these planes are contained within b. This gives the set of linear equations

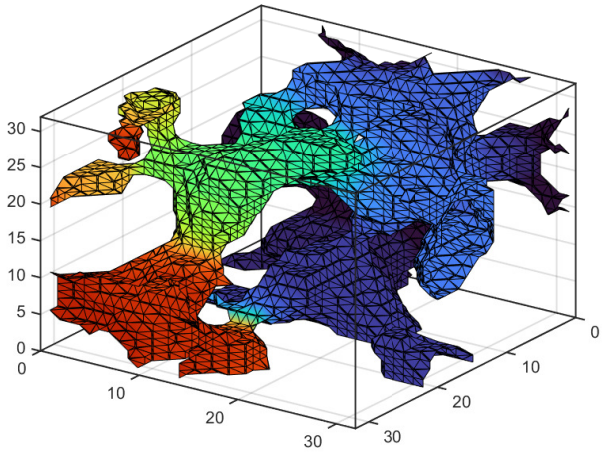
$$k_{eff} P = b. \tag{16}$$

An example calculation for this new equation is shown below, where the boundary pressure is set to 33 to introduce a gradient of 1. Only border voxel k_{-1} is present in the vector b because k_{N+1}

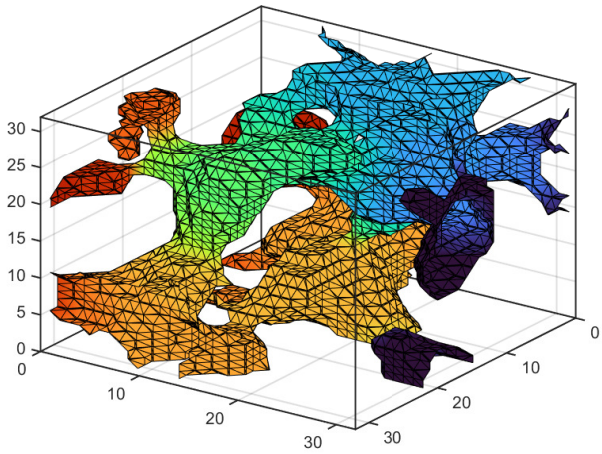
was multiplied with a pressure of 0.

$$\begin{bmatrix} k_{11} & 0 & k_{13} & 0 & 0 & \dots & 0 & 0 \\ 0 & k_{22} & k_{23} & 0 & 0 & \dots & 0 & 0 \\ k_{31} & k_{32} & k_{33} & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & k_{44} & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{55} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & & \\ 0 & 0 & 0 & 0 & 0 & & k_{N-1,N-1} & k_{N-1,N} \\ 0 & 0 & 0 & 0 & 0 & & k_{N,N-1} & k_{NN} \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ \vdots \\ P_{N-1} \\ P_N \end{bmatrix} = \begin{bmatrix} -33 \cdot k_{1,-1} \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

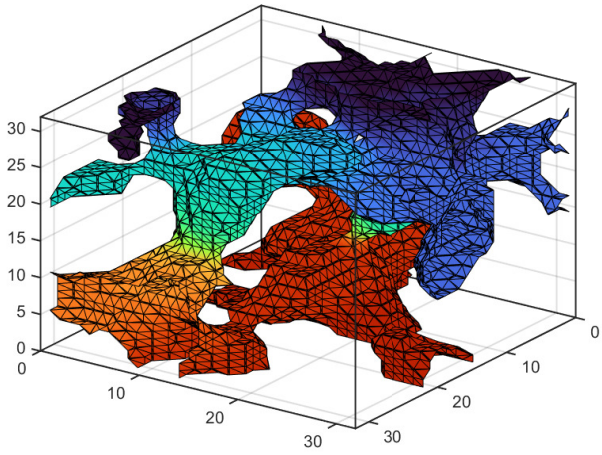
In order to find the 3 diagonal elements of the permeability tensor, there should be 3 calculations, each with a different vector for boundary conditions b_{x_i} with $x_i \in [x, y, z]$. As was explained previously, to determine the permeability in the x-direction the voxels in the two faces of the cube in the yz plane should have values to allow directional flow while the rest of the linear equations retain a balanced in- and outflow. This process should be repeated for the xy and xz planes to get 3 different pressure fields for a single geometry. Just like the connectivity matrix, this set of equations is solved with a sparse matrix solver. To illustrate the results, Fig. 5 contains a visualisation of these pressure fields for a random sample taken from the set. While the geometry is the same, each image contains a pressure grid with a different flow direction.



(a) *Pressure along X*



(b) *Pressure along Y*



(c) *Pressure along Z*

Figure 5: *Visualisation of the pressure fields calculated for a sample.*

With these boundary conditions, (13) should changed to include a velocity since there is now a net flow through the grid. One equation for each direction is then found as

$$\sum_{x_i}(b_{x_i} - P_{x_i}) = u_{x_i} \tag{17}$$

where $x_i \in [x, y, z]$ is the direction of the pressure gradient. The directional replacement permeability for the full structure can be found using the Darcy equation (5) and rewriting it as

$$k = \vec{u}\frac{\mu}{\Delta P}. \tag{18}$$

With the geometries being isotropic, the 3 diagonal entries of the permeability tensor can be averaged to a single value. A histogram of the 10^5 permeabilities that were calculated with respect to the porosities of Fig. 2 is shown in Fig.6.

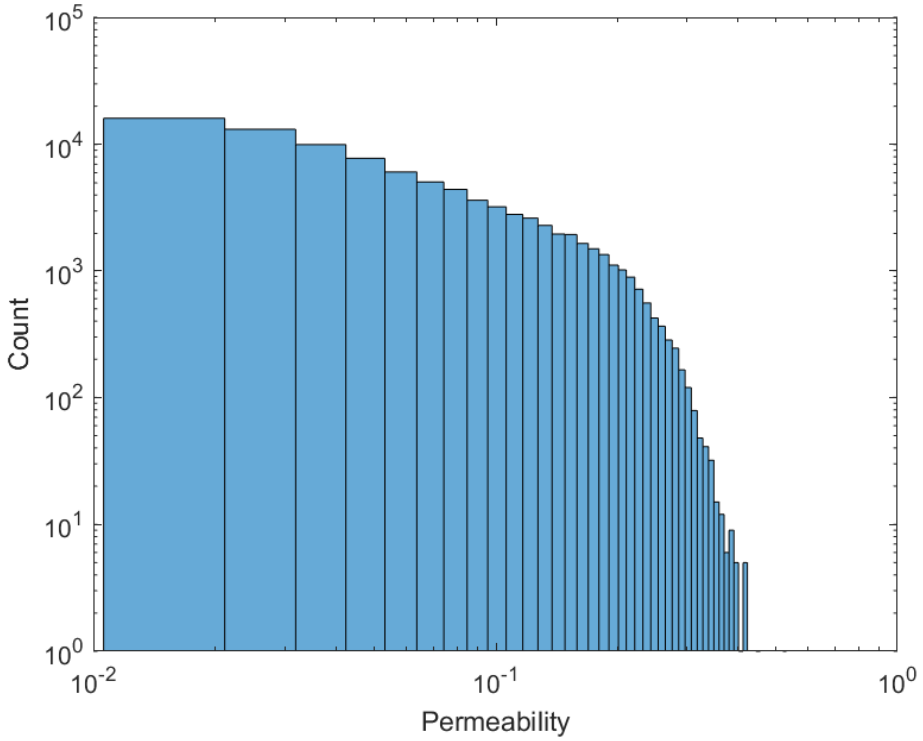


Figure 6: *Histogram that shows a log-log plot of the permeability of the 10^5 samples that were generated.*

From the comparison of the two figures, a relation is not immediately evident. While theoretically the permeability increases with the porosity, the manner in which this happens is unpredictable as it relies heavily on the interconnectivity of the porosity in the structure. The relation between permeability and porosity is also shown in Fig. 7 where the density of datapoints increases from red to blue.

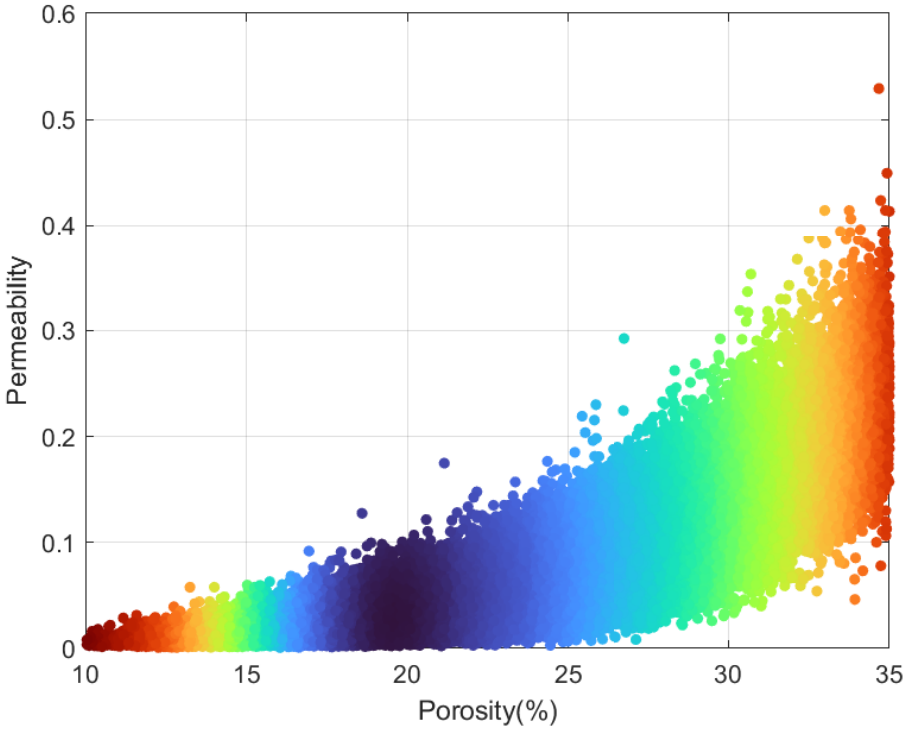


Figure 7: *Density scatter plot of the permeability against the porosity of the 10^5 samples that were generated.*

3.3 Symmetry

Since the permeability is determined as the average of results obtained from pressure gradients applied in three orthogonal directions, all symmetry operations of a cube applied to the sample will lead to the same permeability. These symmetry operations can be used to generate additional inputs for the CNN to train on, in the hopes of having it learn these symmetric properties. Because there is a chance of over-fitting the net by flooding it with symmetries and not having enough unique geometries, only the 24 rotational operations are added to the set. The 24 reflective operations of the cube are disregarded at this stage but will be discussed further in section 5. Nevertheless, the original data-set of 10^5 samples is increased to 2.4×10^6 samples with negligible computational effort.

4 Convolutional Neural Network

Because the geometries contain 32^3 voxels leading to as many inputs in the neural network, a traditional densely connected neural network cannot be implemented. Such a network would require too many weights and would lead to inefficient, redundant computation. A convolutional neural net can be used instead to work with local features of the geometry that can be processed identically. A schematic overview of the convolutional layer can be seen in Fig. 8.

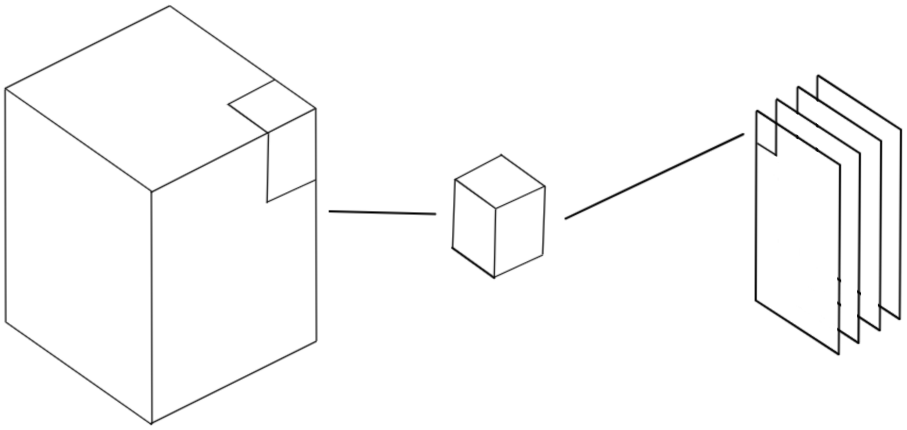


Figure 8: *Schematic overview of feature mapping by a CNN. Every feature map on the right has a filter that convolves the geometry to a set of values.*

The convolutional layer moves a set of 3D-kernels across the original geometry and convolves the $NxNxN$ voxels to a smaller set of values. The step-size of the kernel movement should be set to less than the kernel's size to introduce overlap between the output of the kernel and include connectivity between the voxels [11]. The goal is to retain these connectivity properties as much as possible while reducing the number of values that express this information. For the purpose of this report a kernel size of $4x4x4$ is chosen with a kernel stride of 2. The layers can be added sequentially to reduce the amount of neurons to a size that is then manageable for the densely connected layers that 'top off' the neural net. All these layers, including the convolutional ones use regression through gradient descent in order to find the permeability value.

4.1 Activation and Loss functions

The activation function used in this paper is the so called rectified linear unit (ReLU) which passes the value through a neuron following

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

This activation function has become standard in training deep neural networks because of quick gradient propagation, sparse activation and efficient computation [12]. The 'ReLU' has a downside however, which is the 'dead ReLU' problem. Because the weight of a neuron is updated through gradient descent, a neuron with a gradient difference of 0 will get trapped and never get updated nor activated. This decreases the range of possibilities that the net can converge to and causes it to get stuck in a local minimum or find another sub-optimal solution. To combat this problem a LeakyReLU activation function is also investigated in which weights are updated through

$$\mathrm{LeakyReLU}(\alpha, x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

where α is a value typically ranging from 0 to 1 that drags a ReLU node from it's dead zone.

The first loss function that is used in this paper is the mean absolute percentage error (MAPE). Though the name contains the word 'absolute', this function gives a relative error in the form of

$$\text{MAPE} = \frac{1}{N}\sum^{N}\left|\frac{\text{Truth} - \text{Predicted}}{\text{Truth}}\right| \cdot 100\%. \tag{19}$$

This error metric does not account for the fact that a given percentage error for a very permeable sample counts 'more heavily' than the same percentage error for a low permeability sample. In section 5 it will become clear that it might be beneficial to work with an error metric that accounts for this.

4.2 Technicalities

The total set-up including both software and hardware that was used to simulate the neural net and achieve the results is shown in Table 1.

Software	
Matlab	2020b
NVIDIA Cuda Toolkit	Version 11.2
cuDNN	Version 8.1.1
Python	Version 3.8.6
Keras Tensorflow	Nightly build 2.5.0-dev
Hardware	
CPU	Intel Core i7 10700k
RAM	32GB @3200MHz
GPU	Nvidia RTX 3070*

Table 1: *Software and Hardware used during simulation. *184 Tensor Cores and 20.31 TFLOPS.*

Geometry generation, dataset expansion through symmetry and permeability calculations are all done with Matlab and the GPU Coder package that generates CUDA code. The neural net was simulated on Windows through a Conda kernel. All calculations including those of the network were done with GPGPU acceleration to allow for parallel processing, this should be taken into account when judging time-based results.

5 Results and Discussion

The CNN that is generated will be evaluated with respect to the deviation between prediction and the true permeability, and also with respect to the time it took to get to this result. The set of samples is shuffled to remove any patterns and split into a training set of 2.3×10^6 samples and a validation set of 10^5 samples. The training set is used to train the network and it's error will be addressed as 'loss' in the results. The purpose of the validation set is to evaluate the network with respect to inputs that it has not seen before, the error of this set will be addressed as 'validation loss'. Both losses are expressed in MAPE (19).

5.1 Error

The amount of combinations that can be made with the parameters of a CNN is vastly larger than the amount that can realistically be explored. To optimize the parameters manual regression is used, in which parameter changes with positive results are kept and those with negative effects

are reversed. A total of more than 50 configurations is evaluated and the most notable parameter changes are condensed into Table 2.

Conv. Layers	Filters	Padding	Dense Layers	Activation	Validation Loss
3	16	False	3	ReLU	**32.0**
3	32	False	3	ReLU	**28.4**
2	32	False	3	ReLU	**31.0**
3	32	False	4	ReLU	**30.1**
3	32	True	3	ReLU	**17.6**
3	32	True	3	LeakyReLU(0.05)	**13.3**

Table 2: *CNN Results for the most effective parameter changes.*

The progression of the losses of the final configuration is shown in Fig. 9 in which it becomes clear that over-fitting starts to occur. The loss for the training data decreases while the validation loss remains relatively constant, meaning that the net is developing a bias towards the training data but is not improving on new inputs. The final net contains 264,842 trainable weights which is less than

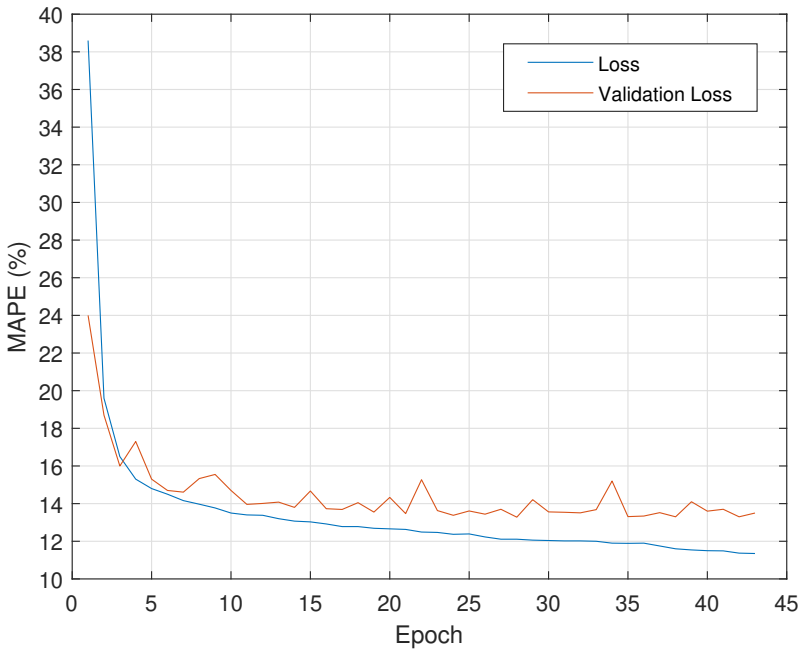


Figure 9: *Mean absolute percentage error for each epoch of both the loss and validation loss.*

10 times the amount of inputs that the net has. While it is still possible to make small improvements on validation loss through the amount of nodes in the dense layer, this is seen as a diminishing return on computational investment. A smaller network is less prone to over-fitting, is more easily deployed and has a faster performance, making it ill-advised to sacrifice these properties. However, this goes both ways; decreasing the amount of nodes too far causes the net to miss out on important features in determining the permeability. A validation loss of 13.3 is therefore deemed acceptable in the scope of this report being a proof of concept. Epoch 27 is chosen as the best performing parameter configuration and the prediction results for the validation set are shown in Fig. 10.

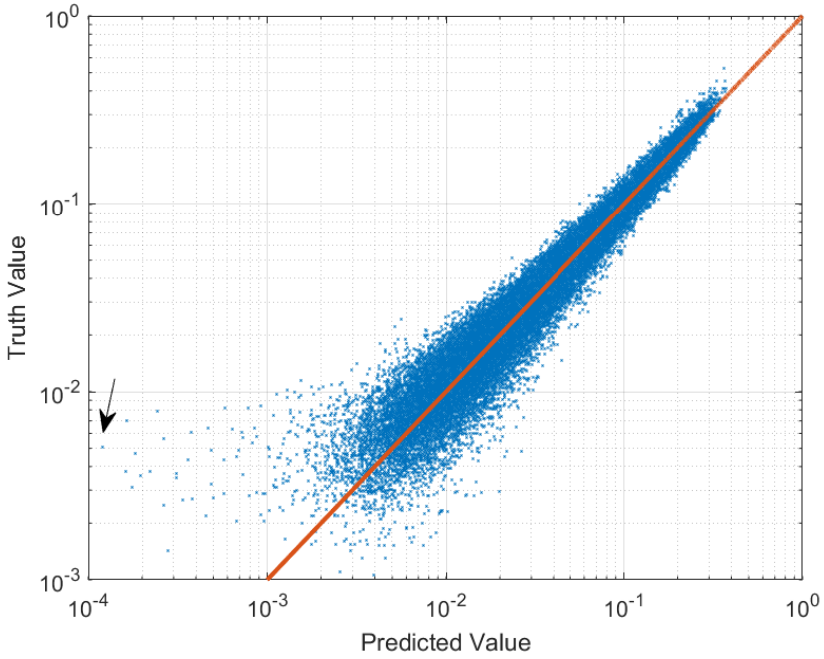


Figure 10: *A plot showing the performance of the last CNN configuration of Table 2. The red line represents a 0% error and the arrow indicates the biggest outlier.*

What immediately becomes clear here is the relation between performance degradation and the permeability of the sample. A hypothesis is that small permeability samples are constrained by a small pore-neck that connects the pressure boundaries. This property generally does not play well with a convolutional net, as this one connection voxel is only one out of the 64 values that the kernel averages over.

5.2 Outliers

In order to improve the CNN it is worthwhile to look at the outliers that have the biggest prediction error. The arrow in Fig. 10 shows the worst prediction that was made for the 10^5 validation samples and it's pressure field is shown in Fig. 11.

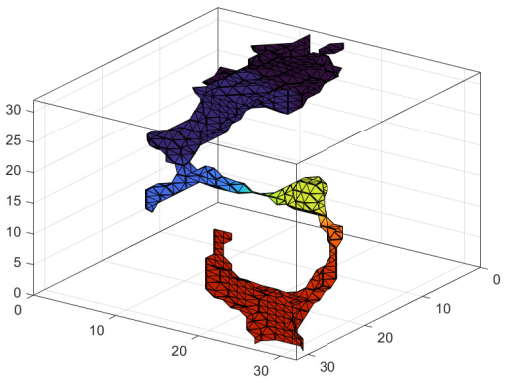


Figure 11: *The geometry and pressure field of the sample with the biggest validation error.*

The figure only shows the branch by which the two pressure boundaries are connected, all other voxel branches are left out of the plot. As was hypothesized, this sample has it's boundaries connected by a branch that narrows down to just a single voxel. Porosity now loses it's role in permeability estimation entirely and the network has to guess the order of magnitude at this scope. In practice, this error is not detrimental but it does leave room for improvement.

5.3 Improvements and alternative loss function

A series of improvements can still be made on the CNN to ensure a lower validation error. First, the dataset contained no impermeable or directionally-impermeable samples. While the dataset was constrained to rocks that have between 10% and 35% porosity, training samples with a porosity lower than this 10% could help the CNN estimate samples at the lower bound better.

Then, the results are limited by computational power as well. Due to the stochastic nature of regression, running more epochs will eventually find better weights and less validation error for the same parameters. There is also no reason to believe that the final parameters that were used are optimal, they were simply settled for due to time constraint. A supercomputer with a proper parameter estimation algorithm is likely to improve on the 13.3% error that was reached.

Rotational symmetry recognition is also something that was never configured in the CNN. The data-set was created with this recognition in mind and it was assumed that the network would pick up on this and evolve to a state where it takes the property into account, but this was never investigated. A more elegant solution in the future would be to 'hard-wire' this property into the network. At this point one can guess that the net understands rotational independence by the results, but there is no guarantee that it does. If it is certain that the CNN understands symmetry, the data-set can safely double by including the reflective permutations of every image. The full symmetry group of a cube, all 48 permutations, can be fed to the network with less danger of over-fitting.

Finally, the results can be changed by disregarding the error of low-permeability samples. It has become clear that samples with a low permeability cause the greatest error, while they also require the least precision in practice. For example, when investigating the permeability of a rocky material for CO_2 storage, an absolute error is more important than a relative one at this scale [4]. An attempt can be made to improve performance by disregarding low permeability samples through a high pass filter(HPF). This filter can be implemented through the $tanh(x)$ function. The response of the hyperbolic tangent is shown in Fig. 12 for the range of the permeabilities in the data-set.

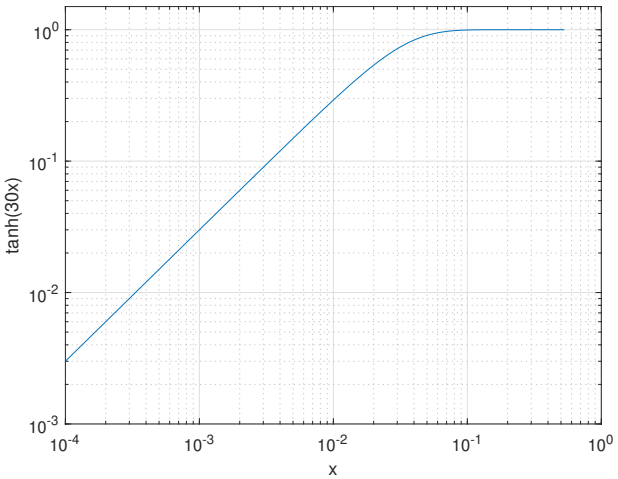


Figure 12: *Response of the tanh function within the bounds of the data-set.*

The samples with a relatively high permeability retain their original error while the error of those with a low permeability is reduced. This is an attempt to keep the error relative in the important range and making it more absolute or 'numeric' in the range of little importance. Instead of adjusting the results with this importance bias, the net can use this bias to achieve better results. The original MAPE error function of (19) can be adapted to include the bias as

$$\text{Custom Error} = \frac{1}{N}\sum^{N}\left|\frac{\text{Truth} - \text{Predicted}}{\text{Truth}}\right| \cdot \text{Tanh}(30 \cdot \text{Truth}) \cdot 100\%. \tag{20}$$

Implementation of this custom error function gives rise to the average error plot and individual error plot in Fig. 13 and 14 respectively.

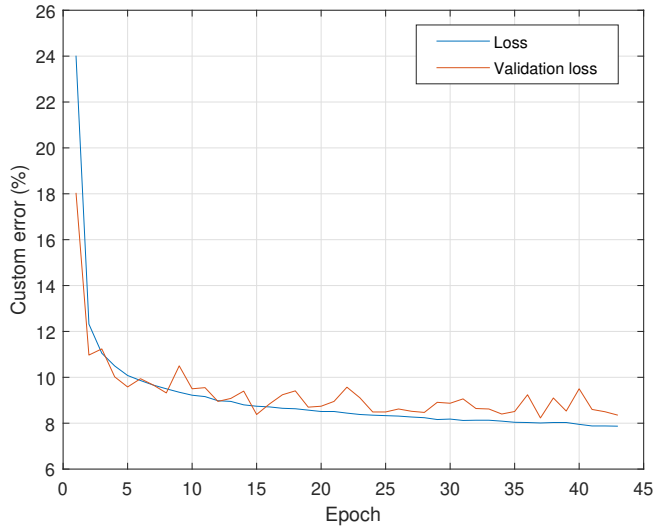


Figure 13: *Custom error for each epoch of both the loss and validation loss.*

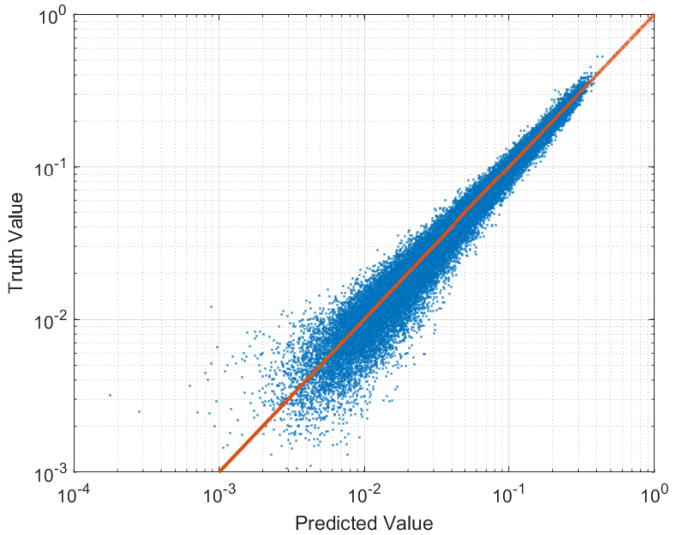


Figure 14: *A plot showing the performance of the last CNN configuration of Table 2 with custom error function. The red line represents a 0% error.*

These figures are deceiving. Comparing Figures 9 and 13 could lead one to believe that implementation of the custom error function leads to a 5% decrease in validation loss. This is not the case because they are not using the same error metric. Comparing Figures 10 and 14 gives the impression that the spread is approximately the same for both low and high permeability samples. This is also not correct though, since only the furthest outliers of each segment are visible and the density of samples on the line is not. Table 3 gives a better depiction of the differences between the nets by evaluating the results with both error functions.

Training loss function	MAPE error	Custom error
MAPE	13.3	11.5
Custom loss function	13.4	8.35

Table 3: *Evaluation of the neural net with different loss functions.*

From here it becomes clear that the new loss function did not improve on validation losses, but biased the network to focus more on high-permeability samples during training. This is a promising result, as it proves that the CNN allows for manipulation towards a bias.

5.4 Computation time

Another aspect that the performance of the CNN is judged by is the time it takes to estimate the permeability of a sample. An important advantage that the CNN has is that the computational time is independent of the sample. Simulating a pressure grid across a sample is highly dependent on the amount of voxels and the connectivity between these voxels, while the evaluation of a net is a constant time-step. For this reason, the porosity-time relation is plotted in Fig. 15.

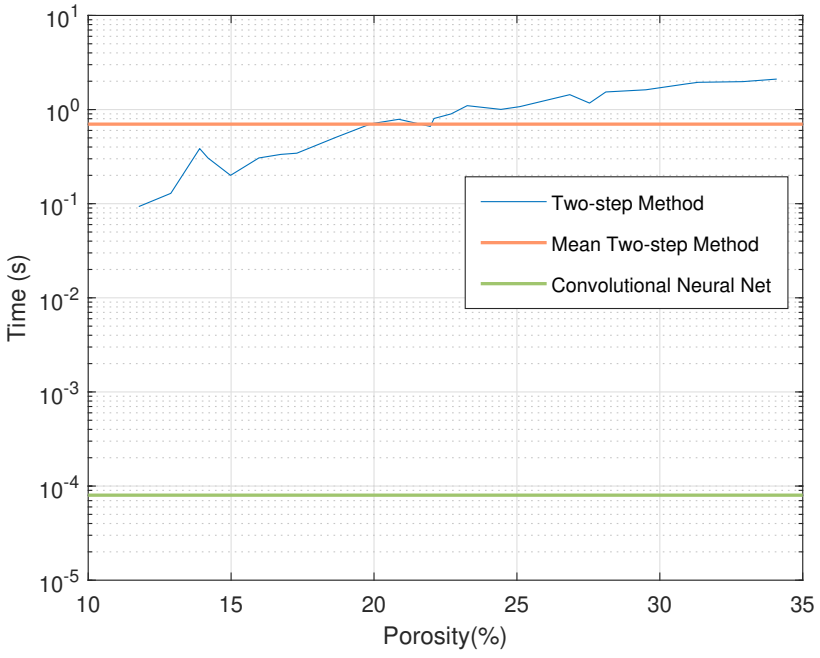


Figure 15: *The time performance of the CNN with respect to the two-step method and the mean of the two-step method. The values depict the time elapsed for a single permeability calculation.*

Because an optimization algorithm removes all the voxels that do not contribute to the permeability value in the two-step method, the blue line gains some stochastic properties. For this reason the

mean elapsed time of the two-step method is of more importance, and should be compared with the CNN instead. On average, using the two-step method requires 0.74 seconds to compute the permeability while the CNN uses only 8×10^{-5} seconds. This means that with an average error of 13% a speedup of almost 10,000x is reached.

5.5 Future Research

The CNN proposed in this paper leaves room for accuracy improvements. There are other hurdles that need investigation before implementation though. No concise research is done on scalability and the increased complexity of structures that comes with a larger scale. An image of dimensions 1024x1024x1024 has 32^3 as many voxels as the geometries used in this paper. The question remains whether a CNN can work with this many, often redundant, inputs. As the amount of inputs grows, a larger kernel or more sequential convolutional layers are required for the image to be processable. This begs the question whether small pore-necks or 'thin' connections that often define the permeability remain observable by the network. If this is the case and the accuracy remains when scaling up, permeability estimation could be done in real-time. In application, a CNN attached directly on scanner-level through an FPGA [13] would then be able to estimate the permeability faster than the inputs can be generated.

6 Conclusion

A CNN is created that can successfully estimate the permeability of binary pore-scale images. An error of 13.3% is considered acceptable and a 10^4-fold speedup is achieved on the same hardware. The error increases for low-permeability samples which can be attributed to small pore-necks not being properly recognized by a set of convolutional layers. Over-fitting became a re-occurring problem which hints at problems in the data-set or parameters of the CNN. Room for improvement remains with possibility for more stochastic descent and parameter optimization.

An alternative custom loss function including the hyperbolic tangent is proposed and it is proven that the created neural net can be biased to shift or possibly decrease the error. What should be investigated next is whether a CNN can scale to larger images, as this is required for application. Symmetric properties should also be hardwired into the network as this can lead to a drastic increase in performance and allows for an increase of the data-set. Because the network's performance can at most be as good as the permeability calculation it is trying to imitate, the data-set for networks generated with this purpose should be created with care. The permeabilities in this set should be calculated with only accuracy in mind and the set should contain cube permutations as well as impermeable samples to improve performance.

7 References

[1] Richard J Brown. *Porosity Establishing the Relationship between Drying Parameters and Dried Food Quality*. 2016. ISBN 9783319230443.

[2] L.S. Efimova and S.A. Minina. The porosity of tablets. 6(7):105, 1973.

[3] Alejandro Romero-Ruiz, Niklas Linde, Thomas Keller, and Dani Or. A Review of Geophysical Methods for Soil Structure Characterization. *Reviews of Geophysics*, 56(4):672–697, 2018. ISSN 19449208. doi: 10.1029/2018RG000611.

[4] Kun Sang Lee, Jinhyung Cho, and Ji Ho Lee. *CO 2 Storage Coupled with Enhanced Oil Recovery*. 2020. ISBN 9783030419004.

[5] L. Leu, S. Berg, F. Enzmann, R. T. Armstrong, and M. Kersten. Fast X-ray Micro-Tomography of Multiphase Flow in Berea Sandstone: A Sensitivity Study on Image Processing. *Transport in Porous Media*, 105(2):451–469, 2014. ISSN 01693913. doi: 10.1007/s11242-014-0378-4.

[6] Hiroshi Okabe and Martin J. Blunt. Prediction of permeability for porous media reconstructed using multiple-point statistics. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, 70(6):10, 2004. ISSN 1063651X. doi: 10.1103/PhysRevE.70.066135.

[7] Amir Raoof and S. Majid Hassanizadeh. A new method for generating pore-network models of porous media. *Transport in Porous Media*, 81(3):391–407, 2010. ISSN 01693913. doi: 10.1007/s11242-009-9412-3.

[8] Zhenyu Liu and Huiying Wu. Pore-scale modeling of immiscible two-phase flow in complex porous media. *Applied Thermal Engineering*, 93:1394–1402, 2016. ISSN 13594311. doi: 10.1016/j.applthermaleng.2015.08.099. URL `http://dx.doi.org/10.1016/j.applthermaleng.2015.08.099`.

[9] Umang Agarwal, Faruk Omer Alpak, and J. M.Vianney A. Koelman. Permeability from 3D Porous Media Images: a Fast Two-Step Approach. *Transport in Porous Media*, 124(3):1017–1033, 2018. ISSN 15731634. doi: 10.1007/s11242-018-1108-0. URL `https://doi.org/10.1007/s11242-018-1108-0`.

[10] Y L De Jong. Permeabilities from binary porescale images A machine learning proof-of-principle in 2D. 2021.

[11] V Koelman. From Logistic Regression to Deep Learning. pages 1–53, 2019.

[12] Qishang Cheng, Hong Liang Li, Qingbo Wu, Lei Ma, and King Ngi Ngan. Parametric Deformable Exponential Linear Units for deep neural networks. *Neural Networks*, 125:281–289, 2020. ISSN 18792782. doi: 10.1016/j.neunet.2020.02.012. URL `https://doi.org/10.1016/j.neunet.2020.02.012`.

[13] Sayuti Rahman, Marwan Ramli, Fitri Arnia, Rusdha Muharar, and Arnes Sembiring. Performance Analysis of mAlexnet by Training Option and Activation Function Tuning on parking images. *accepted for publication on Journal Of Physics Conference Seies*, 2020. doi: 10.1088/1757-899X/1087/1/012084.

A Appendix

A.1 Github

All of the code that was used in this document is stored in a Github repository `https://github.com/swpenninga/permeabilityCNN`. It includes the generation of geometries, permeability calculation and network training. A workflow proposal is also given along with some explanation on the technical side. Some of the algorithms that were briefly mentioned in the report are also explained further in the code.

A.2 Code used for CNN training - Python

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense, Conv3D
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import MinMaxScaler
import time
import gc
import seaborn as sns
import math

physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
print('keras: %s' % keras.__version__)


model = keras.models.Sequential()
layers = tf.keras.layers

model.add(layers.Conv3D(filters=32, kernel_size=(4,4,4), strides=2,padding="same",
    activation=tf.keras.layers.LeakyReLU(alpha=0.05), use_bias=False, input_shape
    =(32,32,32,1),data_format='channels_last'))
model.add(layers.Conv3D(filters=32, kernel_size=(4,4,4), strides=2,padding="same",
    activation=tf.keras.layers.LeakyReLU(alpha=0.05), use_bias=False))
model.add(layers.Conv3D(filters=32, kernel_size=(4,4,4), strides=2,padding="same",
    activation=tf.keras.layers.LeakyReLU(alpha=0.05), use_bias=False))
model.add(layers.Flatten())

model.add(layers.Dense(64,tf.keras.layers.LeakyReLU(alpha=0.05),use_bias = False))
model.add(layers.Dense(10,tf.keras.layers.LeakyReLU(alpha=0.05),use_bias = False))
model.add(layers.Dense(1,tf.keras.layers.LeakyReLU(alpha=0.05),use_bias = False))

model.summary()

def custom_loss(y_true,y_predicted):
    error = 100*tf.math.abs((y_true-y_predicted)/y_true)*tf.math.tanh(30*y_true)
    return error

model.compile(optimizer=Adam(learning_rate=0.001,amsgrad=True),loss=custom_loss)

ep=1
batch=20

for j in range(0,43):
    print('Set of Epoch '+str(1+ep*j)+'-'+str(ep*(j+1)))
    print('')

    for i in range(0,8):
        labels = np.load('data/labels'+str(i)+'.npy')
        samples = np.load('data/samples'+str(i)+'.npy')
        if i==7:
```

```
            model.fit(x=samples[0:200000],y=labels[0:200000],validation_split=0.001,batch_size=batch,epochs=ep,shuffle=True,verbose=1)
        else:
            model.fit(x=samples,y=labels,validation_split=0.001,batch_size=batch,epochs=ep,shuffle=True,verbose=1)
        gc.collect()
        time.sleep(10)

    samples = np.load('data/samples7.npy')
    labels = np.load('data/labels7.npy')
    score=model.evaluate(samples[200001:300000],labels[200001:300000], verbose=1)
    gc.collect()
    time.sleep(10)
```

A.3 Geometry generation - Matlab

```
%%clear
clear;
tic

%% initialize values
sigma = 2.5;                    %filtering strength
vertice = 32;                   %vertice length of cube
level = 0.517;                  %cut-off level that determines porosity
boundaries = [10,35];           %min and max porosity
n = 100000;                     %amount of porous images
i = 0;                          %position of stored data
iteration = 0;                  %counting iterations per successful geometry

%% generation
while 1
    image = rand(vertice,vertice,vertice); %generate random noise in a 3D matrix
    filtered = imgaussfilt3(image,sigma);  %apply a gaussian averaging filter
    BW = imbinarize(filtered,level);       %cut off and replace with binary values

    %Determine if material is porous in z-direction
    locations = find(BW(:,:,1));
    BW2 = imfill(~BW,locations);
    BW3 =  BW - ~BW2;  %The change after z-direction filling algorithm

    %Determine if material is porous in x-direction
    locations2 = find(BW(:,1,:));
    locations2 = floor((locations2-1)/vertice)*vertice^2 + 1 + mod(locations2-1,vertice);
    BW22 = imfill(~BW,locations2);
    BW32 =  BW - ~BW22; %The change after x-direction filling algorithm

    %Determine if material is porous in y-direction
    locations3 = find(BW(1,:,:));
    locations3 = floor((locations3-1)/vertice)*vertice^2 + 1 + vertice*mod(locations3-1,vertice);
    BW23 = imfill(~BW,locations3);
    BW33 =  BW - ~BW23; %The change after y-direction filling algorithm

    iteration = iteration + 1;

    %determine percentage of ones / porosity
    percentageOfOnes = sum(BW(:) == 1) / numel(BW) * 100;

    %boundary conditions in the if statement: porous in all 3 directions,
    %and the proper porosity value.
    if (sum(BW3(:,:,vertice),'all') ~= 0 && sum(BW32(:,vertice,:),'all') ~= 0 && sum(BW33(vertice,:,:),'all') ~= 0 && percentageOfOnes > boundaries(1) &&
    percentageOfOnes < boundaries(2))
        i = i + 1;
        dataset(:,:,:,i) = BW;              %store data
```

```
        level = unifrnd(0.506,0.517);       %refresh the cut-off boundary for a
    different porisity
        iterations(1,i) = iteration;
        percentages(i,1) = percentageOfOnes;
        iteration = 0;
        if i == n
            break                           %quit the program when the dataset is
    full
        end

    end

end
toc

save('dataset.mat', 'dataset','percentages','-v7.3');

%% plotting
% Uncomment these lines for plots.

% figure(1)
% colormap gray
% slice(double(BW),1:vertice,1:vertice,1:vertice)
%
% figure(2)
% colormap gray
% slice(double(filtered),1:vertice,1:vertice,1:vertice)
%
% figure(3)
% colormap gray
% slice(double(image),1:vertice,1:vertice,1:vertice)
%
% figure(4)
% histogram(percentages,30)
```

A.4 Permeability calculation - Matlab

```
%DISCLAIMER: This code contains functions that do not have a
%CUDA-equivalent. In this state, the code cannot be ran on a GPU using for
%example the GPU coder, but it does use the parallel processing toolbox of
%matlab for multicore CPU processing.

clear;
format long g

%% initialize values
load('dataset.mat');                    %Load 32x32x32xN 3D matrices and porosity
    values from 'generate_data.m'
vertice = length(dataset(:,:,:,1));     %Find the length of the cube's vertices
n = length(dataset(1,1,1,:));           %Find the amount of cubes in the set
depthnumber = vertice^3 - vertice^2;    %depthnumber and verticesq are the same for the
     entire set, only calculate once
verticesq = vertice^2;
pressure = [vertice+1,0];               %solutions independent of pressure, this gives
    a gradient of 1 anyhow.

perm = zeros(8*n,1);                    %predefining the permeability vector.

tic

%% Boundary conditions and finding k
parfor v=1:n                            %Parfor enables hyperthreading (use the
    parallel processing toolbox).
    for xyz=1:3                         %3 loops for calculations in 3 directions
        geometry = dataset(:,:,:,v);

        %Here the locations of the x,y, and z planes are found. They are
```

```
        %automatically mapped to the 1-1024 plane so the second floor()
        %function maps them back to their actual position.
        if xyz==1
            locations = find(geometry(:,1,:));
            locations = floor((locations-1)/vertice)*vertice^2 + 1 + mod(locations-1,
    vertice);
            locations2 = find(geometry(:,vertice,:));
            locations2 = floor((locations2-1)/vertice)*vertice^2 + vertice*(vertice
    -1) + 1 + mod((locations2-1),vertice);
        elseif xyz==2
            locations = find(geometry(1,:,:));
            locations = floor((locations-1)/vertice)*vertice^2 + 1 + vertice*mod((
    locations-1),vertice);
            locations2 = find(geometry(vertice,:,:));
            locations2 = floor((locations2-1)/vertice)*vertice^2 + vertice + vertice*
    mod((locations2-1),vertice);
        elseif xyz==3
            locations = find(geometry(:,:,1));
            locations2 = find(geometry(:,:,vertice))+depthnumber;
        end
        %Fillforward and fillbackward together determine the unreachable or
        %unused voxels of the structure, these are removed to decrease
        %computation time as they do not contribute.
        fillforw = imfill(~geometry,locations);
        fillback = imfill(~geometry,locations2);
        unreachable = ~fillforw | ~fillback;
        geometry=  geometry - unreachable;


        %position finds the indices of open spaces (1's) and shrinks the
        %size by 65%-90% to decrease computation time.
        position = find(geometry);
        sparsematrix = zeros(length(position),length(position));   %predefining the
    matrix that contains linear equations of 2-step method.
        keffmatrix = zeros(length(position),length(position));     %predefining the
    matrix that contains linked permeabilities from voxel to voxel.
        neighbourtype = zeros(length(position),6);                 %for each voxels
    this matrix will contain it's 3D neighbours.


        % edge-planes
        yplanetop = [];
        yplanebot = [];
        xplanetop = [];
        xplanebot = [];
        zplanetop = [];
        zplanebot = [];

        %This loop runs for every voxel that is open.
        for j=1:length(position)
            %Matlab matrix counting from find() goes as follows;
            %(1,1,1) -> (1,2,1) -> (2,1,1) -> (2,2,1) -> (1,1,2)
            %initialize these values once so they will not be computed multiple
            %times
            mody = mod(position(j),vertice);
            modx = mod(position(j),verticesq);
            %set the diagonal that will be reduced if at edges
            diagonal = 6;
            %y-axis check in positive direction
            if mody ~= 0 %modulo determines if at y-bottom
                if ismember(position(j)+1,position)==1 %is there something under j
                    sparsematrix(j,find(position==position(j)+1)) = -1;
                    neighbourtype(j,1) = j+1;
                end
            else
                diagonal = diagonal -1; %if j is at y-bottom diagonal is decreased
                yplanebot(end+1) = j;
            end
```

```
            %y-axis check in negative direction
            if mody ~= 1 %modulo determines if at y-top
                if ismember(position(j)-1,position)==1  %is there something above j
                    sparsematrix(j,find(position==position(j)-1)) = -1;
                    neighbourtype(j,2) = j-1;
                end
            else
                diagonal = diagonal -1; %if j is at y-top diagonal is decreased
                yplanetop(end+1) = j;
            end
            %x-axis check in positive direction
            if modx <= (vertice^2 - vertice) && modx ~= 0 %modulo determines if at x
    top
                if ismember(position(j)+vertice,position)==1  %is there something to
    the right of j
                    sparsematrix(j,find(position==position(j)+vertice)) = -1;
                    neighbourtype(j,3) = find(position==position(j)+vertice);
                end
            else
                diagonal = diagonal -1; %if j is at x-edge diagonal is decreased
                xplanetop(end+1) = j;
            end
            %x-axis check in negative direction
            if modx > vertice || modx == 0 %modulo determines if at x bottom
                if ismember(position(j)-vertice,position)==1 %is there something to
    the left of j
                    sparsematrix(j,find(position==position(j)-vertice)) = -1;
                    neighbourtype(j,4) = find(position==position(j)-vertice);
                end
            else
                diagonal = diagonal -1; %if j is at x-edge diagonal is decreased
                xplanebot(end+1)=j;
            end
            %z-axis check in positive direction
            if position(j) <= depthnumber %if not at the max depth
                if ismember(position(j)+verticesq,position)==1 %is there something
    behind j
                    sparsematrix(j,find(position==position(j)+verticesq)) = -1;
                    neighbourtype(j,5) = find(position==position(j)+verticesq);
                end
            else
                diagonal = diagonal -1; %if j is at max depth diagonal is decreased
                zplanebot(end+1) =j;
            end
            %z-axis check in negative direction
            if position(j) > vertice^2 %if not at the front
                if ismember(position(j)-verticesq,position)==1 %is there something in
     front of j
                    sparsematrix(j,find(position==position(j)-verticesq)) = -1;
                    neighbourtype(j,6) = find(position==position(j)-verticesq);
                end
            else
                diagonal = diagonal -1; %if j is at front diagonal is decreased
                zplanetop(end+1)=j;
            end
            sparsematrix(j,j) = diagonal;
        end
        %making the matrix sparse after filling decreases computation time.
        sparsematrix = sparse(sparsematrix);
        %calculating the k value per voxel
        k = sparsematrix\ones(length(position),1);
        %generating a spaceholder matrix that will store series k values.
        keff = zeros(length(k),6);

        %% Second step with pressure gradient
        %for every k the neighbouring k's have their series k calculated
        %for the k_effective matrix.
        for j=1:length(k)
```

```
            valuek = k(j);
            neighbours = find(sparsematrix(j,:)==-1);
            if neighbours ~= 0
                for h=1:length(neighbours)
                    keff(j,h) = 2*k(neighbours(h))*valuek / (k(neighbours(h))+valuek)
;
                    keffmatrix(j,neighbours(h)) = -keff(j,h);
                end
                keffmatrix(j,j) = sum(keff(j,:));
            else %this should never be called but is a safety measure nonetheless.
                sparsematrix(j,j)=0;
            end
        end
        %Here the boundary conditions are applied, depending on whether we
        %work on x,y or z the diagonal elements of the matrix are
        %increased at the border voxels.
        %one boundary is set to pressure(1), the other to pressure(2) and
        %no-flow conditions are removed.
        if xyz==1
            if isempty(xplanebot)==0 && isempty(xplanetop)==0
                keffmatrixX=keffmatrix;
                boundary=zeros(length(position),1);
                for q=1:length(xplanetop)
                    if neighbourtype(xplanetop(q),4)~= 0
                        keffmatrixX(xplanetop(q),xplanetop(q)) = keffmatrixX(
xplanetop(q),xplanetop(q)) - keffmatrixX(xplanetop(q),neighbourtype(xplanetop(q)
,4));
                        boundary(xplanetop(q),1) = pressure(1)*-keffmatrixX(xplanetop
(q),neighbourtype(xplanetop(q),4));
                    end
                end
                for q=1:length(xplanebot)
                    if neighbourtype(xplanebot(q),3)~= 0
                        keffmatrixX(xplanebot(q),xplanebot(q)) = keffmatrixX(
xplanebot(q),xplanebot(q)) - keffmatrixX(xplanebot(q),neighbourtype(xplanebot(q)
,3));
                        boundary(xplanebot(q),1) = pressure(2)*-keffmatrixX(xplanebot
(q),neighbourtype(xplanebot(q),3));
                    end
                end
                keffmatrixX = sparse(keffmatrixX);
                px = keffmatrixX\boundary;
                velocity = zeros(length(xplanetop),1);
                for q=1:length(xplanetop)
                    if neighbourtype(xplanetop(q),4)~=0
                        velocity(q,1) = (pressure(1)-px(xplanetop(q)))*-keffmatrixX(
xplanetop(q),neighbourtype(xplanetop(q),4));
                    end
                end
                permeabilityX = sum(velocity)/((pressure(1)-pressure(2))*vertice);

            else
                permeabilityX =0;
            end
        end
        if xyz==2
            if isempty(yplanebot)==0 && isempty(yplanetop)==0
                keffmatrixY=keffmatrix;
                boundary=zeros(length(position),1);
                for q=1:length(yplanetop)
                    if neighbourtype(yplanetop(q),1)~= 0
                        keffmatrixY(yplanetop(q),yplanetop(q)) = keffmatrixY(
yplanetop(q),yplanetop(q)) - keffmatrixY(yplanetop(q),neighbourtype(yplanetop(q)
,1));
                        boundary(yplanetop(q),1) = pressure(1)*-keffmatrixY(yplanetop
(q),neighbourtype(yplanetop(q),1));
                    end

```

```
                end
                for q=1:length(yplanebot)
                    if neighbourtype(yplanebot(q),2)~= 0
                        keffmatrixY(yplanebot(q),yplanebot(q)) = keffmatrixY(
    yplanebot(q),yplanebot(q)) - keffmatrixY(yplanebot(q),neighbourtype(yplanebot(q)
    ,2));
                        boundary(yplanebot(q),1) = pressure(2)*-keffmatrixY(yplanebot
    (q),neighbourtype(yplanebot(q),2));
                    end
                end
                keffmatrixY = sparse(keffmatrixY);
                py = keffmatrixY\boundary;
                velocity = zeros(length(yplanetop),1);
                for q=1:length(yplanetop)
                    if neighbourtype(yplanetop(q),1)~=0
                        velocity(q,1) = (pressure(1)-py(yplanetop(q)))*-keffmatrixY(
    yplanetop(q),neighbourtype(yplanetop(q),1));
                    end
                end
                permeabilityY = sum(velocity)/((pressure(1)-pressure(2))*vertice);

            else
                permeabilityY =0;
            end
        end
        if xyz==3
            if isempty(zplanebot)==0 && isempty(zplanetop)==0
                keffmatrixZ=keffmatrix;
                boundary=zeros(length(position),1);
                for q=1:length(zplanetop)
                    if neighbourtype(zplanetop(q),5)~= 0
                        keffmatrixZ(zplanetop(q),zplanetop(q)) = keffmatrixZ(
    zplanetop(q),zplanetop(q)) - keffmatrixZ(zplanetop(q),neighbourtype(zplanetop(q)
    ,5));
                        boundary(zplanetop(q),1) = pressure(1)*-keffmatrixZ(zplanetop
    (q),neighbourtype(zplanetop(q),5));
                    end

                end
                for q=1:length(zplanebot)
                    if neighbourtype(zplanebot(q),6)~= 0
                        keffmatrixZ(zplanebot(q),zplanebot(q)) = keffmatrixZ(
    zplanebot(q),zplanebot(q)) - keffmatrixZ(zplanebot(q),neighbourtype(zplanebot(q)
    ,6));
                        boundary(zplanebot(q),1) = pressure(2)*-keffmatrixZ(zplanebot
    (q),neighbourtype(zplanebot(q),6));
                    end
                end
                keffmatrixZ = sparse(keffmatrixZ);
                pz = keffmatrixZ\boundary;
                velocity = zeros(length(zplanetop),1);
                for q=1:length(zplanetop)
                    if neighbourtype(zplanetop(q),5)~=0
                        velocity(q,1) = (pressure(1)-pz(zplanetop(q)))*-keffmatrixZ(
    zplanetop(q),neighbourtype(zplanetop(q),5));
                    end
                end
                permeabilityZ = sum(velocity)/((pressure(1)-pressure(2))*vertice);

            else
                permeabilityZ = 0;
            end
        end
    end
    perm(v,1) = (permeabilityX + permeabilityY + permeabilityZ)/3;
end
toc

```

```
%if the dataset is too large, use '-v7.3'
save('data\porosities.mat','percentages')
save('data\permeabilities.mat','perm')


%% Plotting
%uncomment to plot, though the plots only work for the last orientation
%that was calculated as this has the proper geometry.

% figure(1)
% fig = double(geometry);
% fig(position(:)) = px(:);
% slice(fig,1:vertice,1:vertice,1:vertice);
%
% figure(2)
% fig = double(geometry);
% fig(position(:)) = py(:);
% slice(fig,1:vertice,1:vertice,1:vertice);

% figure(3)
% fig = double(geometry);
% fig(position(:)) = pz(:);
% slice(fig,1:vertice,1:vertice,1:vertice);

% figure(4)
% slice(double(geometry),1:vertice,1:vertice,1:vertice);

%% Questions?
% s.w.penninga@student.tue.nl
```