EIT Simulation: Conductivity Reconstruction and Optimisation of Electrode Selection

Ivo Mihov

9918179

The University of Manchester

5th January 2020

This experiment was performed in collaboration with *Vasil Avramov* and *Connor Beecham*

Abstract

Electrical Impedance Tomography (EIT) is a non-invasive type of tomography that applies current to a sample to explore its conductivity distribution. Multiple characteristics of the regions inside the sample can be determined from their conductivity [1]. In 2009 experts in the area of EIT constructed a new algorithm for conductivity reconstruction called GREIT [2]. Another team of scientists used the D-Bar method for conductivity reconstruction in 2018 [3]. They improved the reconstructions of discontinuous conductivity distributions using an image segmentation convolutional neural network known as U-Net [3,4]. This proof of concept was done to make advances in the conductivity reconstruction of 2-dimensional materials. Our work combined the two approaches: performing the conductivity reconstructions using the GREIT algorithm and improving the detection of discontinuities using the U-Net image segmentation convolutional neural network. Furthermore, a novel approach to electrode selection was implemented. All introduced methods were tested on simulated data and are presented in the report. The output conductivity maps of the neural network outperformed the raw GREIT reconstructions.

Contents

1 Introduction and Problem Description

A typical Electrical Impedance Tomography (EIT) scenario in 2D consists of equally spaced electrodes placed along the boundary of a sample (see Figure 1). Two of the electrodes are connected to a current source, and two others - to a voltmeter. Note that both direct current and alternating current can be used in EIT. The current source and sink are kept fixed and the voltage difference at two other electrodes is measured. The electrodes measuring the voltage are changed to gain information about the voltage in different areas. After a set of voltage measurements is done, the source and sink are changed to recover more voltage measurements. Different measurement techniques are possible. A commonly used strategy is shown in Figure 1: measuring the voltage difference between all adjacent electrode pairs that have no connection to the current source [5].

The main objective of EIT is the precise reconstruction of the conductivity, given voltage measurements. Thus, a relationship between the voltage and the conductivity must be derived. One can find it using the electric field $\mathbf{E} = -\nabla V - \frac{\partial \mathbf{A}}{\partial t}$, where V is the potential and A is the magnetic vector potential [6]. In Electrostatics $\mathbf{E} = -\nabla V$ and Ohm's law can be expressed through the potential:

$$\mathbf{J} = \sigma \mathbf{E} = -\sigma \nabla V, \tag{1}$$

where $\mathbf{J}$ is the *current density* and σ is the *conductivity* (or *permittivity*[1]) [6]. As seen on Figure 2, the equipotential lines are curves situated around the electrodes. Their shape strongly depends on the conductivity distribution of the sample and the current always crosses them perpendicularly. This is shown by the relation between the current density $\mathbf{J}$ and the potential V (Eq. 1). Eq. 1 also illustrates that more electrons pass through areas with bigger conductivity.

To establish a connection between the potential and the conductivity, we use the conservation of charge equation in two dimensions (see Appendix A for derivation)

$$\frac{\partial \sigma_Q}{\partial t} + \nabla \cdot \mathbf{J} = 0, \tag{2}$$

where $\sigma_Q = \frac{dQ}{dS}$ is the *surface charge density* [6].

One helpful aspect of the physical problem is that the net charge density inside the samples remains constant (see Appendix A). Therefore, the charge density σ_Q is constant and not an explicit function of time. Substituting Eq. 1 into Eq. 2 and setting the first term to zero, we get that globally

$$\nabla \cdot (\sigma \nabla V) = 0. \tag{3}$$

Locally, one can still use

$$\nabla \cdot (\sigma \nabla V) = -\frac{d\sigma_Q}{dt} \tag{4}$$

which is non-zero only at the source and sink.

Since only the derivative of V is present in Eq. 3, V can only be defined up to an additional constant (a derivative of a constant is zero). Thus, a reference is needed to calculate a sensible conductivity distribution. That is why the first task is to simulate the voltages that would be measured were the permittivity uniform.

There are two main problems in EIT, the Forward Problem and the Inverse Problem. In the Forward Problem, the voltage measurements, V, are found from a given conductivity distribution, σ. The opposite, finding the permittivity from voltage measurements is known as

[1]The term permittivity is used throughout this paper as a synonym of conductivity, where the latter carries the meaning described in [1].

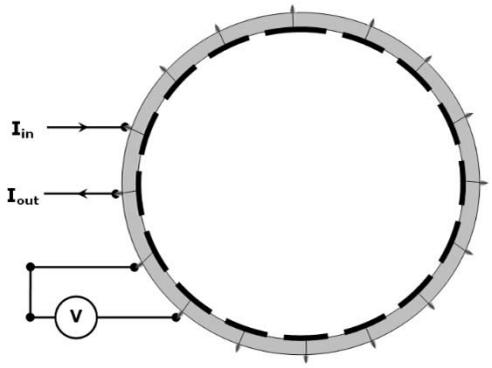


Figure 1: An example of EIT on a circular sample. Two of the electrodes are the current source and sink and other two are connected to a voltmeter to measure voltage difference. Figure taken from [7].

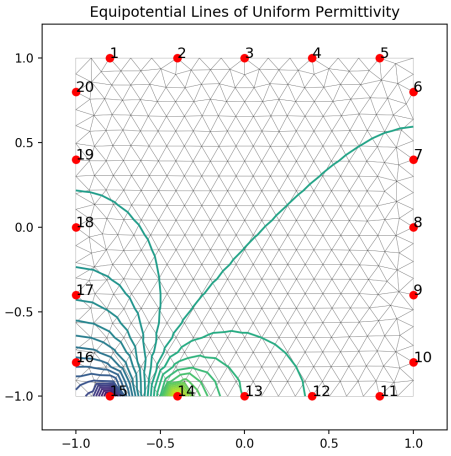


Figure 2: Lines of constant potential are shown on the sample (yellow for high and blue for low potential), calculated using the Finite Element Model. Here electrodes 14 and 15 act as source and sink and the permittivity is uniform.

the Inverse Problem. In the Forward Problem, one calculates what voltage differences would be measured with a given permittivity. Alternatively, one can experimentally measure the voltage differences if working on a real sample and then proceed to the Inverse Problem. One type of boundary conditions that can be used to solve Eq. 3 for the conductivity σ is the Dirichlet boundary condition:

$$V|_{\partial\Omega} = f(x_i), \tag{5}$$

where $f(x_i)$ is a function of the coordinates x_i along the boundary $\partial\Omega$. It assumes that there is a function $f(x_i)$ that describes the potential V along the boundary.

2 Numerical Approximation of the Solution

If the conductivity is known, Eq. 3 can be solved for the potential V using a numerical approximation. One popular method is called the Finite Element Method (FEM) [8]. It assumes that the potential can be split into a number of linear functions, each of which is defined on a limited domain [8]. This is why a discrete meshing is necessary.

2.1 Meshing

The mesh generation technique consists of two parts [9]. First, the nodes are distributed using Delaunay triangulation and then so-called bar forces are used to iteratively make the mesh uniform. [10]

Delaunay's method works by choosing a number of points and finding the cells around them, such that all points inside the cell are closer to this point (in most cases L2 distance is used) than to any other [9]. These polygonal zones are called Voronoi cells. Afterwards, points whose cells share a border are connected with lines. Thus, Delaunay triangulation is complete. Once the nodes are generated, no node lies inside the circumscribed circle of any of the triangles. It

has been proven by Sibson in 1980 that for triangles in a certain meshing to be equiangular, one must start with Delaunay triangulation [11].

The second part of the triangulation exploits bar forces to minimise the differences between the angles of the triangles. The bar forces are vectors of magnitudes equal to the lengths of the sides of the triangles in the same direction, taken with the vertices of the triangles anti-clockwise. This iterative algorithm minimises the net bar force on each node and produces an equiangular meshing, where boundary conditions on fixed points (electrodes and points that define the shape) apply (Figure 2) [12].

2.2 Finite Element Method

Once the mesh is generated, one can use assumptions to model the potential inside the mesh elements. A popular choice for solving elliptical partial differential equations like Eq. 3 is the Finite Element Method (FEM), which assumes that the potential V can be split into many linear functions, each defined only on the domain of one of the elements [8]. With the FEM, the voltage measurements can be simulated based on a permittivity distribution that is either given or assumed to be constant [8]. The form of the assumed boundary condition f from Eq. 5 for triangular elements is shown by the expression

$$f_i = v_1 + v_2 x + v_3 y, \tag{6}$$

where v_1, v_2 and v_3 are the voltages on the three vertices of the triangle and x, y are coordinates on the i-th element, which is the domain of f_i [8]. If we do this for each component of f_i, we get

$$\begin{pmatrix} f_{i,1} \\ f_{i,2} \\ f_{i,3} \end{pmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix},$$

which is valid inside the i-th triangle and where x_1, y_1, x_2, y_2, x_3 and y_3 are the coordinates of the 1st, 2nd and 3rd vertex respectively [8]. This can also be expressed as

$$\mathbf{v} = \mathbf{K}_e^{-1}\mathbf{f}.$$

From the definition of the stiffness matrix one can find that for a single element

$$\mathbf{K}_e = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} = \frac{1}{2\Delta} \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix}, \tag{7}$$

where

$$\begin{aligned} \alpha_1 &= x_2 y_3 - x_3 y_2, \quad \beta_1 = y_2 - y_3, \quad \gamma_1 = x_3 - x_2 \\ \alpha_2 &= x_3 y_1 - x_1 y_3, \quad \beta_2 = y_3 - y_1, \quad \gamma_2 = x_1 - x_3 \\ \alpha_3 &= x_1 y_2 - x_2 y_1, \quad \beta_3 = y_1 - y_2, \quad \gamma_3 = x_2 - x_1 \end{aligned}$$

and Δ is the area of the element:

$$2\Delta = \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix},$$

which can be found in [8]. On the LHS of Eq. 7 we can see the element stiffness matrix that is being sought after [8]. It can be used to assemble the global stiffness matrix, which shows

the effects between all elements. The procedure is simple, stiffness matrices on each element are placed on a large N by N matrix, where N is the number of nodes in the mesh, such that each element of the K_e matrix is placed on the corresponding node of the global stiffness matrix and multiplied by its given permittivity [8, 13]. To clarify, in case we have a triangle with vertices numbered 121, 402 and 322, and a corresponding 3 by 3 stiffness matrix, its elements

$$\begin{bmatrix} (1,1) & (1,2) & (1,3) \\ (2,1) & (2,2) & (2,3) \\ (3,1) & (3,2) & (3,3) \end{bmatrix} \tag{8}$$

go into

$$\begin{bmatrix} (121,121) & (121,402) & (121,322) \\ (402,121) & (402,402) & (402,322) \\ (322,121) & (322,402) & (322,322) \end{bmatrix}, \tag{9}$$

after being multiplied by the corresponding permittivity, such that all terms that go into the same element of $\mathbf{K}_g$ are added to each other [13]. Since all $\mathbf{K}_e$ matrices are symmetric, $\mathbf{K}_g$ is also symmetric [8].

After the global stiffness matrix $\mathbf{K}_g$ is assembled, it is used to find the value of the potential on each node in the mesh from the assumed boundary conditions $\mathbf{f}_{global}$ for the boundary of the entire sample [8]. This is shown by the relation

$$\mathbf{v} = \mathbf{K}_g^{-1}\mathbf{f}_{global}, \tag{10}$$

where $\mathbf{v}$ is the vector containing the values of the potential on each node. To extract any simulated voltage measurements for the given conductivity, one can subtract the potential of the node where the first measuring electrode from the potential of the second one. Therefore, all voltage measurements can be extracted.

2.3 GREIT Method - Conductivity Reconstruction

The GREIT algorithm aims to find a matrix that reconstructs the change in conductivity $\mathbf{x} = \sigma - \sigma_r$ based on a series of measurements [2]. It is called Difference EIT because it can only reconstruct changes in conductivity, rather than absolute values. σ_r is the conductivity which GREIT uses as a reference. The reconstruction algorithm can be summarised using

$$\hat{\mathbf{x}} = \mathbf{R}\mathbf{v}, \tag{11}$$

where $\hat{\mathbf{x}} \in \mathbb{R}^{n_p^2}$ (n_p is the number of pixels for each axis of the square) is the reconstructed map of conductivity changes [2]. $\mathbf{R} \in \mathbb{R}^{n_p^2 \times n_m}$ (n_m is the number of measurements) is the reconstruction matrix and $\mathbf{v}$ is the vector of voltage measurements [2].

To find $\mathbf{R}$, the algorithm minimises the square of the error

$$\begin{aligned} \epsilon^2 =& \sum_k \left\| \tilde{\mathbf{x}}^{(k)} - \mathbf{R}\mathbf{v}^{(k)} \right\|^2_{\mathbf{W}^{(k)}} \\ =& \sum_k \sum_i \left(\left[\tilde{\mathbf{x}}^{(k)}\right]_i^2 \left[\mathbf{w}^{(k)}\right]_i^2 - 2\left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left(\sum_j \mathbf{R}_{ij} \left[\mathbf{v}^{(k)}\right]_j \right) \right. \\ & \left. + \left[\mathbf{w}^{(k)}\right]_i^2 \left(\sum_j \mathbf{R}_{ij} \left[\mathbf{v}^{(k)}\right]_j \right)^2 \right) \end{aligned} \tag{12}$$

where the summation over k is over all training samples [2]. $\mathbf{W}^{(k)}$ is the matrix with the weights, a diagonal matrix composed of the weights on each measurement ($\mathbf{W}^{(k)} = \text{diag}(\mathbf{w}^{(k)})$) [2]. In the above expression $\tilde{\mathbf{x}}$ is the true conductivity change map and thus the L2 norm is the error ϵ^2, which is minimised, as shown in [2]:

$$\mathbf{R} = \text{argmin}(\epsilon^2). \tag{13}$$

The minimum is found by finding where the derivative w.r.t. the matrix $\mathbf{R}$

$$\begin{aligned} -\frac{1}{2}\frac{\partial \epsilon^2}{\partial \mathbf{R}_{ij}} &= \sum_k \left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j - \sum_k \left[\mathbf{w}^{(k)}\right]_i^2 \left(\sum_l \mathbf{R}_{il} \left[\mathbf{v}^{(k)}\right]_l\right) = 0 \\ &= \sum_k \left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j - \sum_l \mathbf{R}_{il} \left(\sum_k \left[\mathbf{v}^{(k)}\right]_l \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j\right) \\ &= \mathbf{A}_{ij} - \sum_l \mathbf{R}_{il}\mathbf{B}_{lij}, \end{aligned} \tag{14}$$

where $\mathbf{A} = \sum_k \left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j$ and $\mathbf{B} = \sum_k \left[\mathbf{v}^{(k)}\right]_l \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j$ [2]. Therefore, one needs to solve the matrix equation

$$\mathbf{A}_{ij} = \sum_l \mathbf{R}_{il}\mathbf{B}_{lij} \tag{15}$$

by rearranging to yield

$$\mathbf{R} = \mathbf{A}\mathbf{B}^{-1} \tag{16}$$

as shown in [2]. It can be shown that GREIT seeks to minimise the generalised Tikhonov regularisation

$$\|\mathbf{v} - \mathbf{J}\hat{\mathbf{x}}\|^2_{\Sigma_n^{-1}} + \|\mathbf{x} - \mathbf{x}^\circ\|^2_{\Sigma_x^{-1}}, \tag{17}$$

where $\mathbf{n}$ is the noise and $\mathbf{\Sigma_n}$ is the noise covariance matrix [2]. Similarly, $\mathbf{\Sigma_x}$ is the conductivity change covariance [2]. With white Gaussian noise which is used for general purpose EIT the noise covariance is zero, so $\mathbf{\Sigma_n}$ is diagonal [2].

3 Improvement of the Recognition of Discontinuous Details

This proof of concept aims to find ways to minimise the number of measurements needed for a high-quality conductivity reconstruction. This problem was tackled in two ways: by an algorithm that finds the next measurement that aims to maximise the information gain and an image segmentation neural network that helps notice discontinuous details in the conductivity map by clearing the noise, increasing the contrast and scaling the conductivity values to match the ones of the individual components.

The idea behind this neural network was to recognise features with distinct permittivity and to cancel the noise. It helped improve regions of discontinuous conductivity, which were not reconstructed well by the GREIT algorithm. It was needed, since GREIT assumed smoothness over some regions. That is why the data the neural network was trained it was composed of individual components (ellipses, triangles and lines of different permittivities). This training set made it suitable for distinguishing discontinuous elements, but more inaccurate in the reconstruction of smooth conductivity distributions. Since it is biased towards finding discontinuities

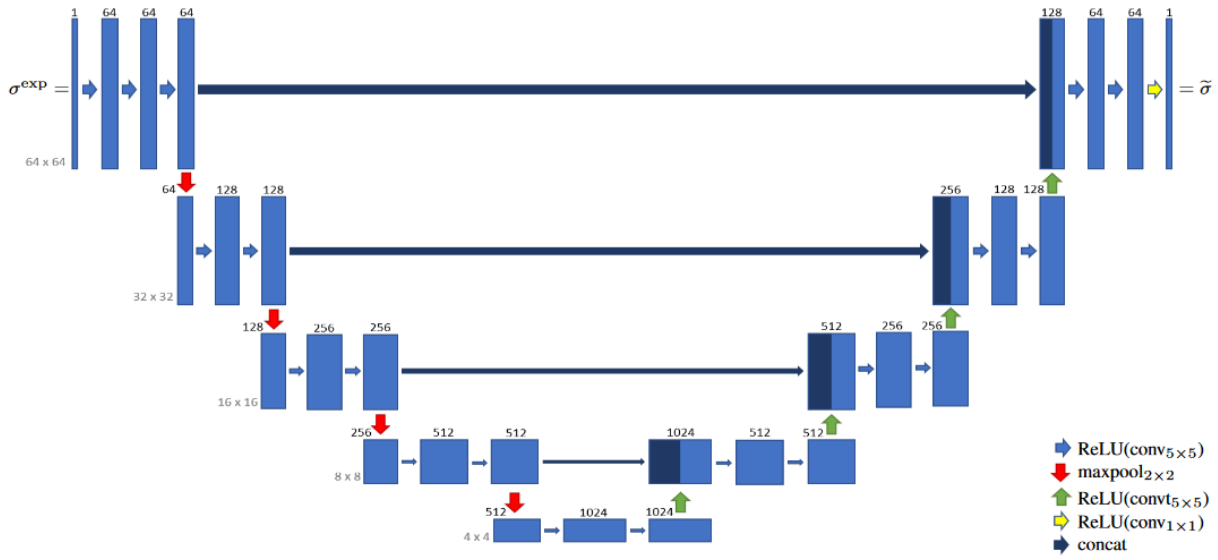


Figure 3: Neural network model from [3] that was used to improve reconstructions from GREIT algorithm [2]. The figure was taken directly from [3] and matches network used in our project exactly.

due to the data it was trained with, it could perform worse on samples with a different type of conductivity distribution.

The output layer of a neural network returns n output values, where n is the number of neurons in the output layer. The one used in our project returns a 64×64 pixel image, so there are 4096 output nodes in total. The squared L2 loss $\pounds$ of the output is found by the expression

$$\pounds = \sum_{i,j} (T_{ij} - R_{ij})^2, \tag{18}$$

where $\mathbf{T}$ is the matrix containing the true image and $\mathbf{R}$ is the matrix containing the reconstructed (output) image [14]. The summation runs over all their elements (or all output nodes). The L2 loss is the cost function, a measure of the quality of reconstruction, i.e. how far it is from the true distribution. Note that the neural network itself uses a half of $\pounds$ for its cost function, but the one from Eq. 18 is the one used throughout this text [14]. The cost function is minimised by the neural network in a process called *backpropagation*. All nodes in the neural network have an activation function, which is applied on the input. It could have any form, but usually discontinuous ones are used [14]. The one used the most in this neural network is the ReLU function

$$f(x) = \max(0, x) \tag{19}$$

where x is the input into the layer. Backpropagation consists of taking the partial derivative of the loss function with respect to each weight in every node of the neural network using chain rule, and propagating it back to the respective node. Then each weight in the network is adjusted in the direction that would decrease the loss [14]. The change of each weight in iteration t is shown by

$$\Delta w_{ji}(t) = -\eta \frac{\partial \pounds(t)}{\partial w_{ji}(t)}, \tag{20}$$

where $w_{ji}(t)$ is the weight of the i-th input for the output of the j-th neuron and $\eta > 0$ is the learning rate [14].

The neural network that was used in this project exploits series of discrete convolutions and max-pooling on the raw image to allow for more variation of the weights and biases of the nodes. The convolutional layers consist of element-wise multiplication with a convolutional filter done on the image beginning with the first square box and then moving by 1 pixel (to the right, then down when the row ends) (Figure 3). All convolutional layers in this neural network have ReLU activation [3,4]. There are 2 convolutional layers before each max-pooling layer. This is repeated 4 times. The max-pooling layer down-samples the input by taking the pixel with the biggest value from each quadrant of the image. After the data is down-sampled to a 4×4 pixel size with a depth of 512, it passes through two convolutional layers and then through a convolutional transpose layer, which up-samples the image. After that it is concatenated with the output from an earlier processing stage (see Figure 3). This way particular characteristics of the original picture can also have an impact on the output to prevent information loss caused by the max-pooling layers. Then two convolutional filters are alternated with one convolutional transpose and a concatenation with an older version, until it reaches the same size as the original image. Finally, it passes through a 1×1 convolutional filter with ReLU activation and the resulting image is returned.

4 Electrode Selection Hierarchy Optimisation

Performing all possible measurements is very hard, time-consuming and engaging. This algorithm was developed to minimise the number of measurements directly without making assumptions about the presence of discontinuities in the permittivity of the sample. This novel algorithm uses knowledge of the reconstruction algorithm and the sample geometry.

The idea behind the algorithm is to look for points/regions that have not been influenced by the measurements. The sensitivity of each pixel is examined by creating fractional perturbations in the formerly performed measurements. The perturbing matrix $\mathbf{P}_n$ for n voltage measurements is given by

$$\mathbf{P}_n = \mathbf{1}_n \mathbf{1}_n^T + p\mathbf{I}_n,$$

where n is the number of performed voltage measurements, $\mathbf{1}_n = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$, such that there are n ones in $\mathbf{1}_n$, and $\mathbf{I}_n$ is the n-dimensional identity matrix. p is a parameter that can be varied ($0 \leq p < 1$) to choose the fractional perturbation. To perturb the measurements, one can construct a matrix $\mathbf{V}_{pert}$ with all perturbed voltage measurements

$$\mathbf{V}_{pert} = \mathbf{P}\,\mathrm{diag}(\mathbf{v}), \tag{21}$$

where $\mathbf{v}$ is the vector of length n with all measured voltages. To get the change of the conductivity with each perturbation, one can use Eq. 11 to get

$$\Delta\sigma_k = \sigma_{pert,k} - \sigma = \mathbf{R}(\mathbf{V}_{pert,k} - \mathbf{v}) \tag{22}$$

for the perturbation of the k-th voltage measurement. Then sum all conductivity reconstructions to find the sensitivity matrix $\mathbf{S}$

$$\mathbf{S} = \sum_{k=1}^{n} |\Delta\sigma_k|. \tag{23}$$

In other words, the unperturbed conductivity is subtracted from each of the maps to find the change that each perturbation had delivered to the conductivity. Finally, the absolute value of all perturbed conductivities was summed to find which region had been affected the least by the previously performed measurements. For the next measurement, the source/sink pair that has the most direct influence on the area of lowest sensitivity was taken. Different options into finding the most direct source-sink pair exist. The method we used is presented in Section 5.4.

5 Implementation

The experiment was split into four parts: preparation, optimisation, data collection for U-net training and measurement number minimisation. This section describes the joint work of two members of our group, Vasil Avramov and Ivo Mihov. In our work we used the NumPy, PyEIT and CuPy packages [12, 15, 16].

5.1 Preparation

Since all data in this project was simulated, all units that we worked with were normalised. The conductivity was normalised such that the background conductivity was set to zero and the absolute zero-conductivity point was set to -1. The side length of the square sample was set to 2 and the centre of the coordinate system was set to be the centre of the sample.

For the generation of the mesh, the 'distmesh.py' module of PyEIT was used as is to generate a triangular mesh with 875 elements [12].

The preparation for the project also included writing new software that was used in addition to the PyEIT package [12]. We wrote a pseudo-random anomaly generator[2], a true conductivity map generator and a module that placed the anomalies into the corresponding regions of the triangular mesh generated using PyEIT (see Appendix B) [12]. The anomaly generator was written for the training of the neural network. It was highly vectorised[3], which allowed further acceleration with the GPU [17]. This enabled the generation of large amounts of training data for the neural network. The anomaly generator created three different types of anomalies: ellipses, triangles and lines. The true conductivity map generator placed the raw anomalies onto a 64×64 pixel map by checking whether the centres of the pixels were inside the anomalous regions. This truth map was used by the U-Net for the calculation of loss. Also, the PyEIT module 'wrapper.py' was modified to allow the existence of elliptic, triangular and linear anomalies on the triangular mesh [12]. This was done using the same technique used for the true data generator. It was checked whether the centres of the triangular elements were inside the anomaly in various manners. The modification allowed solving the Forward Problem with our custom anomalies. Before that it only allowed circular anomalies [12]. Finally, the generation of training data was possible and the conductivity of simulated measurements could be reconstructed.

5.2 Optimisation

Another obstacle in the generation of training data was the time it took for the execution of the code for generation of one sample. Hence, the Finite Element Model (FEM) file 'fem.py' was fully rewritten in vectorised form and the new module was named 'fem_vect.py' ('vect' for 'vectorised'). Originally, it solved the Forward Problem one source-sink pair at a time [12]. With the new,

[2]Anomalies are areas of different conductivities. In our case areas of uniform permittivity were used. They formed discontinuities at the border with the background.

[3]A vectorised code is written only in terms of arrays, matrices and matrix operation [17].

vectorised file, one could solve the Forward Problem for all measurements with all steps[4] at once. Also, the new 'fem_vect.py' file contained no loops, since they were vectorised to make use of the big number of threads in the GPU [17]. To execute the code using the GPU, the CuPy package, which is built onto CUDA, was used [16]. Furthermore, the same procedure was repeated for the 'greit.py' and 'interp2d.py' modules of PyEIT, and they were fully vectorised [2,12]. Finally, the newly written 'fem_vect.py' and parts of 'greit.py' and 'interp2d.py' were accelerated using the GPU with the CuPy package [16].

Furthermore, some matrices that were previously calculated for every sample, were calculated only once in the beginning. All the FEM results were calculated at once for all possible 29070 measurements (with 20 electrodes) and saved on the SSD, along with other details of the calculation (mesh used, measurements done etc.). This way the program was sped up by not having to calculate them and generate the mesh anew for each sample.

5.3 Generation of Simulated Data and Training

The generation of training data was performed using an ordered active electrode list. First, the number of source-sink pairs was set to be 13. Then, for each simulation a random distance between active electrodes was produced, and 13 sources were taken from a list from 0 to 19, ordered by the remainder of their division by 5 (i.e. 0, 5, 10, 15, 1, 6, 11, 16, 2, 7, 12, 17, 3, 8, 13, 18, 4, 9, 14, 19). Then the sinks were found from the source and the electrode distance. This way a better reconstruction quality was ensured by gaining consistent information about the sample.

A limited number of measurements in random order were performed on each sample. The first measuring electrode was a sequence of the numbers between 0 and 19. The second was produced by adding a pseudo-random step to each electrode number (steps range from 1 to 19) and taking the remainder after dividing by 20 (number of electrodes). Thus, an array of measuring electrodes without repeated lines was created. Finally, lines where sources or sinks were present were discarded.

In order to improve the simulation similarity to real EIT data, additive white Gaussian noise with mean zero was added to the voltage readings. The standard deviation of the noise was 3% of the measurement to which it was added. White Gaussian noise has a diagonal covariance matrix, so it is uncorrelated. It was used since no prior knowledge about the real measurement noise was available due to lack of experimental data.

The neural network was trained with 60,000 of these samples and tested with 10,000 samples. The training featured 200,000 iterations, each over a batch of 10 samples from the training data. The number of anomalies on each simulated sample was chosen with a Gaussian probability distribution with a standard deviation of 1, centred around a mean of 1.5. The absolute value of the chosen number was rounded to select the number. The type of anomaly was chosen with a probability of 50% for an ellipse, 10% for a line and 40% for a triangle. Then all other characteristics of the anomalies were pseudo-randomly selected within the appropriate limits (no length could be bigger than the length of the side of the sample and permittivity could not be more than an order from the background permittivity). One exception was the permittivity of lines, which was always 0.001% of the background permittivity to simulate a broken sample.

[4]The distance between the measuring electrodes is called the 'step', e.g. step is 4 if electrodes 4 and 8 are performing measurements, and 19 if electrodes 0 and 19 are measuring (numbering of electrodes starts from zero since Python indices start from zero).

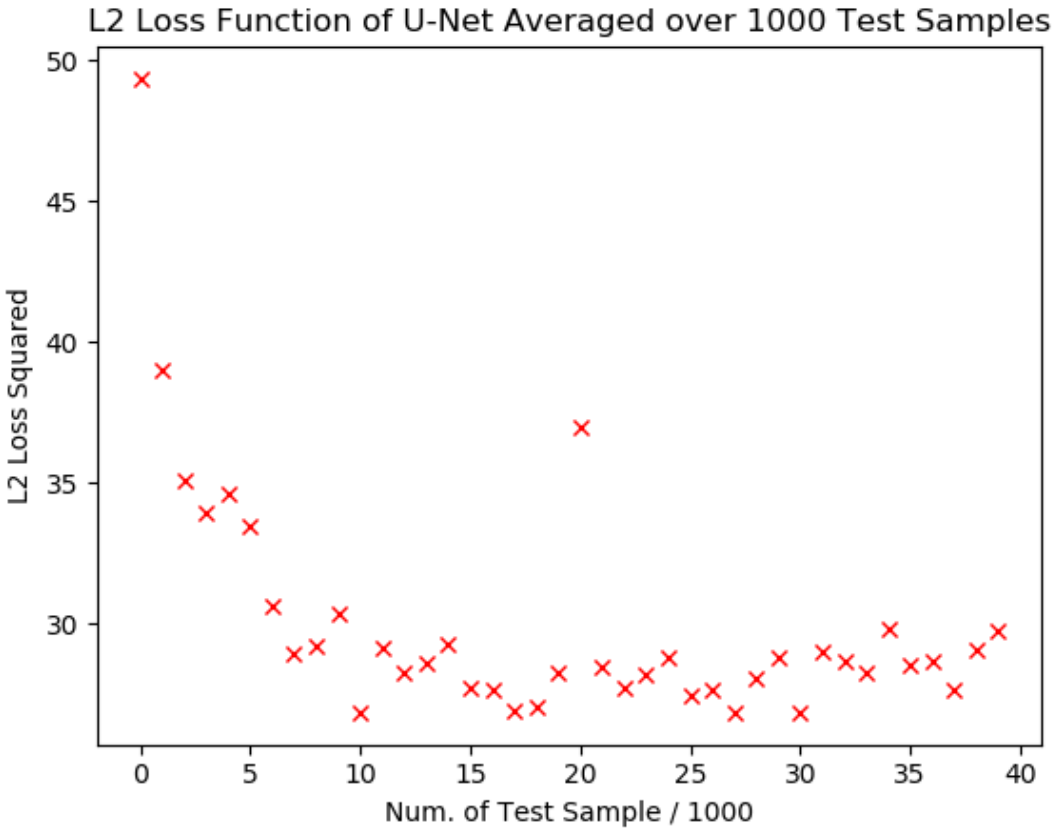


Figure 4: The square of the L2 loss function was monitored throughout the U-Net training by testing on external data. On this figure it is plotted as a function of the sample number divided by 1000, because it was averaged over 1000 samples. The outlier that can be seen in the middle of the graph was caused by a discontinuity that lied on one of the electrodes.

5.4 Electrode Selection Hierarchy Optimisation

The first step was to locate the zones with a small sensitivity to perturbations in the voltages. They were extracted from the array by finding all elements that had a sensitivity smaller that 2 times the sensitivity of the minimal element. The number was chosen arbitrarily. Later, the mean of the resulting distribution was found and the three elements from the distribution that were the closest to the mean were identified.

In the final part of the search for a best source/sink pair, the case of low permittivity variation is examined. The areas of the triangles formed by two electrodes and each the three points were found for all possible pairs of electrodes. Then the areas of the three triangles that share the same electrode pair were summed. This was done for all electrode pairs, resulting in an array with the sum of the three triangle areas for each combination (190 in total since the order does not matter). The pair with the lowest area sum is identified and chosen as the best candidate due to the geometry, i.e. the three points lie close to the line connecting the two electrodes. Thus, information about the conductivity in the area could be found by the amount of current passing through the zone close to the three points.

6 Experimental Results

The results of this proof of concept can be divided into three parts: the speed boost due to the optimisation of the code, the improvement in conductivity reconstruction of samples with discontinuous anomalies and the possible reduction of measurements needed for a conductivity reconstruction.

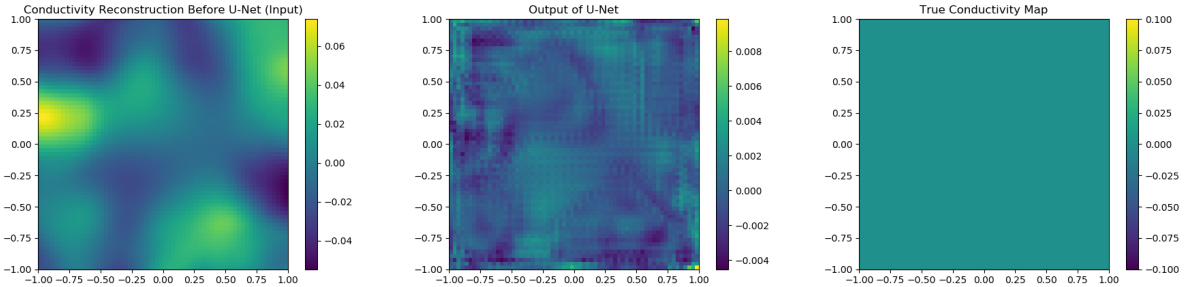
6.1 Code Optimisation Results

All of the optimisation of the code was done and tested on an ASUS FX504GE (Intel Core i5 8300H, RAM 16GB @ 2666 MHz, Nvidia GTX1050Ti 4GB). Other machines could show different results. The initial execution speed of the 'fem.py' module of PyEIT was 1.5 ± 0.1 s, but after the total vectorisation took place, the speed was reduced 37.5 times to 0.04 ± 0.01 s [12]. Less noticeable improvements were seen after the optimisation of the 'greit.py' and the modules that it uses, from which the biggest adjustments were made to 'interp2d.py' [12]. The total running time of 'greit' and all files it uses decreased from 0.8 ± 0.1 s to 0.11 ± 0.01 s. The total time for the simulation of the measurements and the reconstruction of the conductivity of one sample decreased from 4.0 ± 0.3 s to 0.20 ± 0.02 s. Part of this decrease in total reconstruction time was due to the omittance of unnecessary function calls and multiple calculations of equivalent quantities. The results of this part of the project allowed for a better study of the possible solutions of the latter objectives.

6.2 U-Net Results

The U-Net that was taken from the 'Deep D-Bar' paper was trained to decrease the L2 loss of samples that contain discontinuous anomalies [3]. The neural network was trained with 200,000 iterations and every 50 iterations one batch of 10 test samples was tested. The training of the neural network resulted in a decrease of the L2 loss (see Figure 4). The minimum is reached after around 15,000 sample tests, which corresponds to about 75,000 iterations. However, a slight increase can be noticed after the 30,000th sample.

Examples of the U-Net output can be seen on Figures 5-9. The decrease in the loss after the neural network processing was evident in all examined test outputs. Figure 5 contains an instance of the conductivity reconstruction of a sample with uniform conductivity. By comparing the limits on the legends of Figures 5a and 5b, one can notice that the noise in the map has been reduced by approximately one order of magnitude. Figure 6 shows the conductivity maps of a triangle with $\approx 6\%$ deviation from the background before and after the U-Net processing. Figure 6b suggests that the neural network recognised the presence of the anomaly. It can be seen on Figure 6a that the value of the conductivity inside the triangle is of the order of the discrepancies in the reconstruction, caused by the noise on the voltage measurements. However, there was no significant decrease in the squared L2 loss of the conductivity. It decreased from 2.222 to 1.709, which is a 23% decrease. Samples with a bigger deviation from the background are improved by a significantly larger percentage. The conductivity magnitude and the bounds of the elliptical anomaly in the output of the U-Net closely resemble the true conductivity map (see Figure 7). After the U-Net, the sample squared loss decreased by more than 11 times. Figure 8 shows the U-Net interpretation of the GREIT reconstruction of a line anomaly. Although the reconstructed line permittivity was within 0.05 from the true value, the faint triangle on the left of the map was not detected by the U-Net. Also, the slight displacement in the reconstructed position of the line could result in an increase in the loss since the anomaly is about 0.05 units

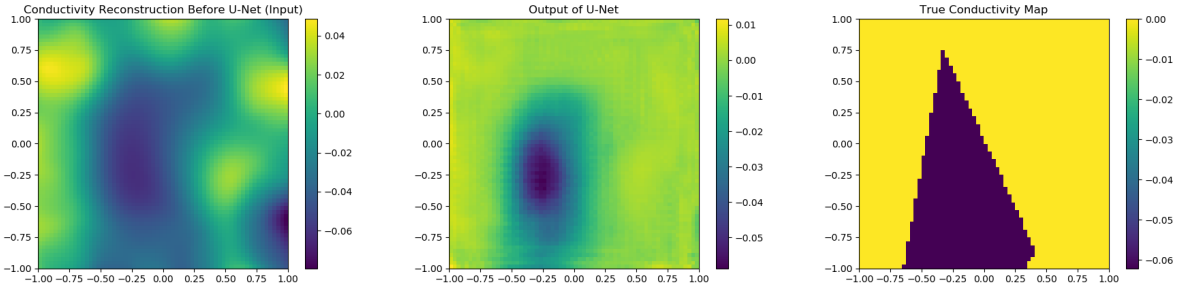


(a) GREIT Reconstruction

(b) Output of neural network.

(c) True permittivity map used to simulate the voltage measurements using the FEM method.

Figure 5: The figures above show the effect of the neural network on test data without any anomalies.



(a) GREIT Reconstruction

(b) Output of neural network.

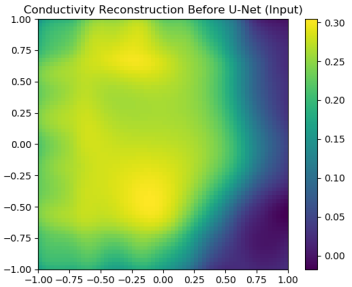
(c) True permittivity map used to simulate the voltage measurements using the FEM method.

Figure 6: The figures above show the effect of the neural network on test data with one triangular anomaly.

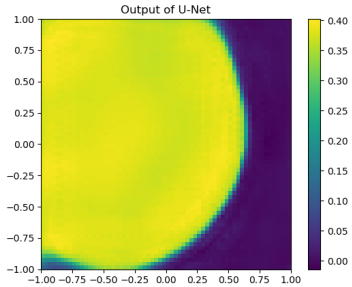
wide. The final image, shown on Figure 9, shows two anomalies and their reconstructions. The loss of the reconstruction decreased from 10.50 before to 2.31 after the U-Net. The triangle in the center of the true conductivity map (Figure 9c) has a deviation of about 2% from the background conductivity and is also noticeable after the U-Net processing , although its area is about 10 times smaller. The results show that this U-Net decreased noise and improved the loss of the shown conductivity reconstructions.

6.3 Electrode Selection Hierarchy Optimisation: Test Results

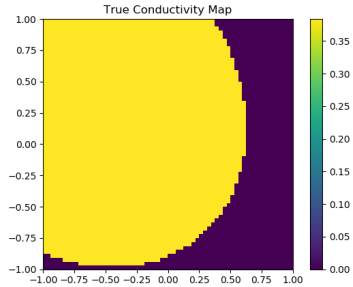
Two tests were performed to evaluate the performance of the custom electrode selection algorithm described in Section 4. Both tests compared the variation of the L2 loss of the conductivity reconstruction with the number of measurements between two measuring strategies. In both cases the two selection techniques were tested on the same permittivity distribution. In each of the two tests, the first strategy was the custom selection technique described in this paper, while the second one was a widely accepted measurement technique. The results of both tests are presented as plots of the squared loss normalised by dividing it by the lowest loss reached by the custom measurement technique from Section 4 versus the number of measurements performed. The values for each test can be found in the text under Figures 10 and 12.



(a) GREIT Reconstruction

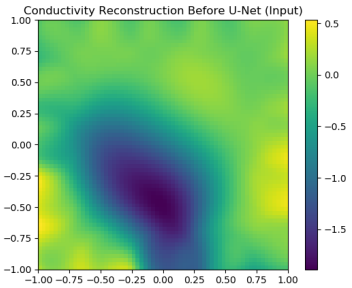


(b) Output of neural network.

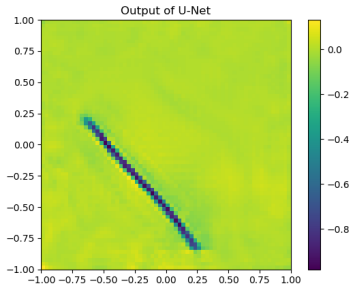


(c) True permittivity map used to simulate the voltage measurements using the FEM method.

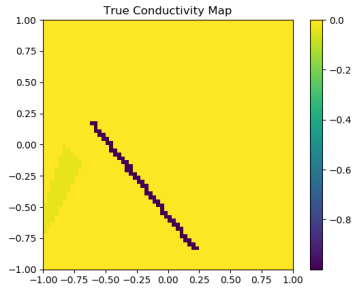
Figure 7: The figures above show the effect of the neural network on test data with one large elliptical anomaly.



(a) GREIT Reconstruction



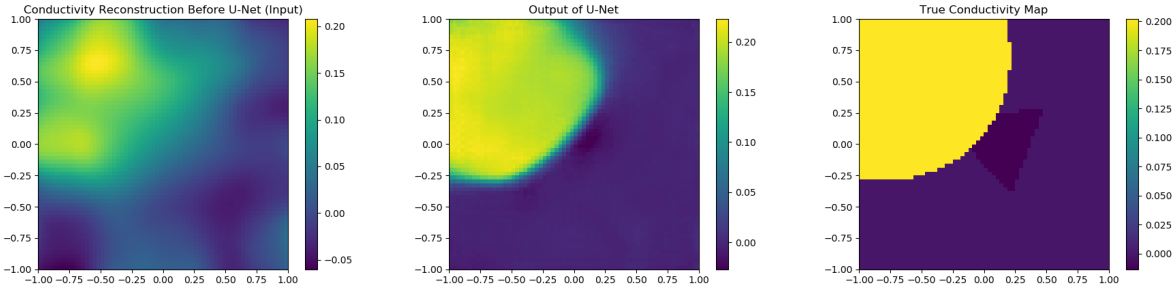
(b) Output of neural network.



(c) True permittivity map used to simulate the voltage measurements using the FEM method.

Figure 8: The figures above show the effect of the neural network on test data with a linear anomaly(simulating a broken sample) and a triangular anomaly with smaller than 10% deviation from the background.

In the first case our measurement technique was compared to a sensing strategy where all measurements with adjacent source and sink and adjacent measuring electrodes are performed (Figure 11). They are done in an order, such that all possible adjacent measurements with a given source/sink pair are performed before it is changed. Figures 10 and 11 suggest that the descent of the conductivity loss generated using the custom algorithm is steeper than the one using the adjacent active electrode scheme. Also, the loss is in general smaller for the same number of measurements selected by the two algorithms. It is noticeable from Figures 10 and 11 that the variation in the loss of some of the samples (red and green dots) was smaller fractionally than in the other ones. By comparing the L2 loss of the custom algorithm after 250 measurements (Figure 10) against the L2 loss of the adjacent measurement technique after 340 measurements (Figure 11), one can notice that the one of the custom measurement technique is consistently smaller. One feature of the standard sensing strategy is the proximity of the source and sink. Because of it, most of the current passes close to the border, which could leave the other zones of the samples less explored [5]. On the contrary, the choice of source and sink pairs by the custom algorithm had no limitations on the choice of source and sink.



(a) GREIT Reconstruction

(b) Output of neural network.

(c) True permittivity map used to simulate the voltage measurements using the FEM method.

Figure 9: The figures above show the effect of the neural network on test data with two overlapping anomalies, one of which has less than 2% deviation from the background.

In the second test the technique was compared to another standard measurement strategy. The difference to the one in the first test was that the active electrodes were always on the opposite sides of the sample, e.g. electrodes 1 and 11 as seen on Figure 1. It can be noticed from Figures 12 and 13 that the final loss of the reconstruction using the commonly used strategy is consistently lower than that of our measurement selection optimiser. The measurement loss of most samples (all but the one represented by the red dots) are decreased to about 90% of the final value for the custom algorithm, while the loss of the 'red' sample is decreased to around 70% of the one reached by the algorithm. In general, one can see from Figure 13 that by the 80-th measurement the proven technique reached the lowest loss value of the custom algorithm. Overall, with this standard measuring practice, a bigger proportion of the current passes through the central region of the sample than with the first standard measurement technique [5].

7 Discussion

The reconstructions of the GREIT algorithm left room for improvements in the loss with the U-Net. There was a decrease in loss in all examples that were seen (Figures 5-9). The loss of the U-Net training reached a minimum value around the 75,000-th iteration and increased after this point, because it began overfitting the training data (see Figure 4). With this training data, the neural network was able to recognise some anomalies with a conductivity that is about 6% smaller than the background (Figure 6b). This is within the boundaries of the noise on the conductivity, which means that with more measurements, this neural network can even distinguish smaller variation. Also, the precision with which the neural network recovered the line on Figure 8 shows that it is very good at recognising elements with deviations of 5 orders from the background conductivity that are of comparable size to the sample. Overall, the neural network performed as expected.

The comparison between the electrode selection hierarchy optimisation and the common sensing strategies shows controversial results. On one hand, the first test (against neighbouring source/sink pairs) shows that in this range of measurements, our custom algorithm performs better (see Figures 10 and 11). On the other hand, the second test suggest exactly the opposite (see Figures 12 and 13). One possibility is that the opposite active electrode technique converges faster, but to a local minimum due to the limitation that it cannot accurately measure

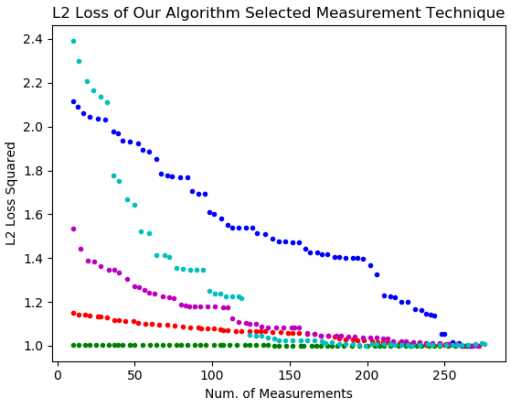


Figure 10: The above plot shows the variation of the normalised squared L2 loss with number of measurements using the measurement technique described in Section 4 on 5 different samples. All losses were normalised by dividing them by the lowest value of the squared loss achieved with the custom measurement technique. The lowest loss value for each sample is as follows: 6.86 for red, 136.98 for blue, 2.103 for green, 126.57 for magenta and 233.50 for cyan.

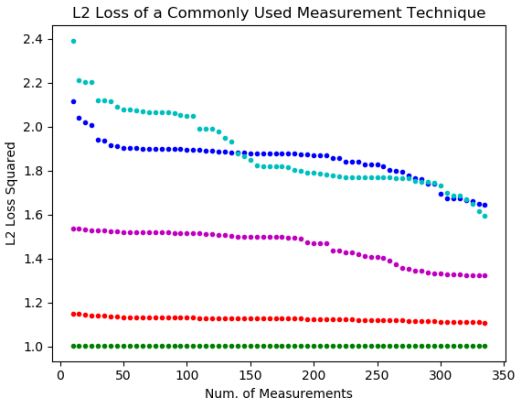


Figure 11: The above plot shows the variation of the normalised squared L2 loss with number of measurements using a standard measuring procedure, performing all measurements with adjacent source and sink and adjacent voltage measuring electrodes [5]. All losses were normalised by dividing them by the lowest value of the loss achieved using our custom measurement technique.

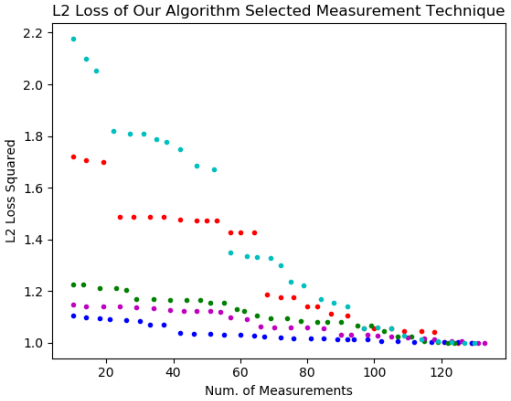


Figure 12: The above plot shows the variation of the normalised squared L2 loss with number of measurements using the measurement technique described in Section 4 for 5 different samples. All losses were normalised by dividing them by the lowest value of the loss achieved using our custom measurement technique. The lowest loss value for each sample is as follows: 21.83 for red, 184.54 for blue, 66.15 for green, 89.23 for magenta and 417.72 for cyan.

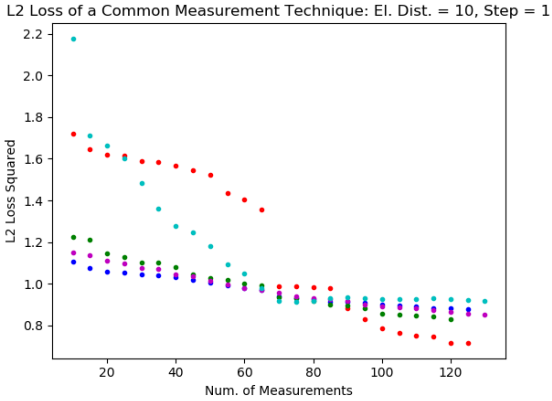


Figure 13: The above plot shows the variation of the normalised squared L2 loss with number of measurements using a standard measuring procedure, performing all measurements with opposite source and sink and adjacent voltage measuring electrodes [5]. Less measurements are possible than with other source/sink pairs due to symmetry. All losses were normalised by dividing them by the lowest value of the loss achieved using our custom measurement technique.

regions close to the boundary. Another option is that the data on which the two algorithms are compared is biased, since there is a bigger probability that there is a central anomaly than in real samples. This would explain also the first test results. In any case, this is an area of further study.

8 Conclusion

The image segmentation neural network showed positive results about future use of such an algorithm in EIT. Its application in the conductivity reconstruction of non-smooth conductivity samples leads to an overall decrease of loss in the examined samples. One possible further improvement is stopping the training earlier to prevent overfitting. Another option is to use data with more types of anomalies and a bigger number of anomalies on each sample to prevent overfitting on the same kind of shapes.

The test results of the electrode selection technique are not conclusive. Further investigation is necessary to find its overall effects on the loss. These trials have to necessarily include a larger number of measurements for which the loss of our algorithm is compared to a widely recognised measurement technique. This would show whether the algorithms are converging to a global, rather than a local minimum. Only then a bigger confidence level of the results could be achieved. The search for an effective way to evaluate the current state of the algorithm is an area of further investigation.

The electrode selection optimisation could also be iteratively trained to find the parameters that would result in a quick descent to the global optimum. This extension of the optimisation is subject to further development.

References

[1] M. Cheney, D. Isaacson, and J. C. Newell, "Electrical impedance tomography," *SIAM review*, vol. 41, no. 1, pp. 85–101, 1999.

[2] A. Adler, J. H. Arnold, R. Bayford, A. Borsic, B. Brown, P. Dixon, T. J. Faes, I. Frerichs, H. Gagnon, Y. Gärber *et al.*, "Greit: a unified approach to 2d linear eit reconstruction of lung images," *Physiological Measurement*, vol. 30, no. 6, p. S35, 2009.

[3] S. J. Hamilton and A. Hauptmann, "Deep d-bar: Real-time electrical impedance tomography imaging with deep neural networks," *IEEE Transactions on Medical Imaging*, vol. 37, no. 10, pp. 2367–2377, Oct 2018.

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[5] W. Mi, W. Qiang, and K. Bishal, "Arts of electrical impedance tomographic sensing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374. [Online]. Available: http://doi.org/10.1098/rsta.2015.0329

[6] D. J. Griffiths, *Introduction to Electrodynamics*, 4th ed. Pearson Education, 2012.

[7] University of Leeds, "Eit sensors - sensor for electrical impedance tomography," 2013, [Online; accessed 29 December 2019]. [Online]. Available: https://www.leeds.ac.uk/olil/tomography/images/DAS/DAS-A.jpg

[8] O. Zienkiewicz, *The Finite Element Method*. London (etc.): McGraw-Hill, 1977.

[9] N. Golias and R. Dutton, "Delaunay triangulation and 3d adaptive mesh generation," *Finite Elements in Analysis and Design*, vol. 25, no. 3, pp. 331 – 341, 1997, adaptive Meshing, Part 2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168874X96000546

[10] M. Vauhkonen, W. R. B. Lionheart, L. M. Heikkinen, P. J. Vauhkonen, and J. P. Kaipio, "A MATLAB package for the EIDORS project to reconstruct two-dimensional EIT images," *Physiological Measurement*, vol. 22, no. 1, pp. 107–111, feb 2001. [Online]. Available: https://doi.org/10.1088%2F0967-3334%2F22%2F1%2F314

[11] R. Sibson, "Locally equiangular triangulations," *The Computer Journal*, vol. 21, no. 3, pp. 243–245, Jan 1978. [Online]. Available: https://doi.org/10.1093/comjnl/21.3.243

[12] B. Liu, B. Yang, C. Xu, J. Xia, M. Dai, Z. Ji, F. You, X. Dong, X. Shi, and F. Fu, "pyeit: A python based framework for electrical impedance tomography," *SoftwareX*, vol. 7, pp. 304–308, 2018.

[13] F. Cuvelier, C. Japhet, and G. Scarella, "An efficient way to perform the assembly of finite element matrices in matlab and octave," *CoRR*, vol. abs/1305.3122, 2013. [Online]. Available: http://arxiv.org/abs/1305.3122

[14] S. Haykin, *Neural networks: a comprehensive foundation*, 2nd ed. Pearson Education, 1999.

[15] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, 2006–, [Online; accessed 02/01/2020]. [Online]. Available: http://www.numpy.org/

[16] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[17] N. P. Rougier, *rougier/from-python-to-numpy: Version 1.1*. Zenodo, Dec. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.225783

[18] R. Wong and Y. Li, "Integral and series representations of the dirac delta function," *Communications on Pure and Applied Analysis*, vol. 7, no. 2, p. 229–247, Dec 2007. [Online]. Available: http://dx.doi.org/10.3934/cpaa.2008.7.229

A Derivation of Local and Global Charge Conservation

Here the Divergence Theorem in two dimensions is derived from Green's Theorem and used to prove the Charge Conservation in 2D. First, an expression for the outward normal is found. One can deduce it from geometry. We need a vector with the same magnitude as the tangent to the boundary $d\mathbf{r} = \begin{pmatrix} dx \\ dy \end{pmatrix}$, but perpendicular to it. This defines the vector up to a sign, which is set so that we get the outward, not the inward normal. The system of equations is as follows:

$$\begin{aligned} &d\mathbf{l} = \begin{pmatrix} l_1 \\ l_2 \end{pmatrix} \\ &l_1^2 + l_2^2 = dx^2 + dy^2 \\ &d\mathbf{r} \cdot d\mathbf{l} = 0. \end{aligned} \tag{24}$$

Now we can find that $d\mathbf{l} = \begin{pmatrix} dy \\ -dx \end{pmatrix}$ and calculate what the change in the charge inside the plate is. Since

$$I = \frac{dQ}{dt} = -\oint \mathbf{J} \cdot d\mathbf{l}, \tag{25}$$

we can now use Green's theorem to express this as an integral over the area [6]. By using Green's theorem, we can express Eq. 25 as

$$\begin{aligned} \oint \mathbf{J} \cdot d\mathbf{l} &= \oint \begin{pmatrix} J_x \\ J_y \end{pmatrix} \cdot \begin{pmatrix} dy \\ -dx \end{pmatrix} \\ &= \int (J_x dy - J_y dx) \\ &= \iint_S \left(\frac{dJ_x}{dx} + \frac{dJ_y}{dy} \right) ds \\ &= \iint_S (\nabla \cdot \mathbf{J})\, ds, \end{aligned} \tag{26}$$

where $ds = dxdy$ is an area element and S is the whole area of the sample [6]. One can also express the charge Q using

$$Q = \iint_S \sigma_Q ds, \tag{27}$$

where σ_Q is the *surface charge density* [6]. Then, by taking the time derivative, we get

$$\frac{dQ}{dt} = \iint_S \frac{d\sigma_Q}{dt} ds. \tag{28}$$

Using this and Eq. 26, we yield

$$\iint_S \frac{d\sigma_Q}{dt} ds = -\iint_S (\nabla \cdot \mathbf{J})\, ds, \tag{29}$$

since any change in charge is charge that came or escaped through the boundary [6]. Therefore, since this must be true for any area S, we get the local charge conservation:

$$\frac{d\sigma_Q}{dt} = -\nabla \cdot \mathbf{J}. \tag{30}$$

Now let us consider if the total charge on the sample is constant, given that the current going into it is the same as the one leaving it. From Eq. 25 and 26 we have

$$\frac{dQ}{dt} = -\iint_S (\nabla \cdot \mathbf{J})\, ds. \tag{31}$$

If the net current (change in charge) is zero, we have

$$\nabla \cdot \mathbf{J} = 0$$

for the total current density on the plate. Therefore, one can show from Eq. 1 that

$$\nabla \cdot (\sigma \nabla V) = 0,$$

where σ is the conductivity.

Thus, the current density at the boundary that is perpendicular to it can be thought of as two Dirac delta functions, one for the source and one for the sink:

$$J_\perp(\mathbf{r}) = J_0(\delta(\mathbf{r_1} - \mathbf{r}) - \delta(\mathbf{r_2} - \mathbf{r})), \tag{32}$$

where $\mathbf{r_1}$ and $\mathbf{r_2}$ are the positions of the source and sink respectively, both along the boundary, $J_\perp(\mathbf{r}) = \mathbf{J}(\mathbf{r}) \cdot \hat{\mathbf{n}}$ is the current density along the inward unit normal to the boundary, and J_0 is the magnitude of the current density that is being passed through the sample. Integrating $J_\perp$ along the boundary, we yield

$$\begin{aligned} \oint J_\perp(x,y)dl &= \int_{-1}^{1} J_\perp(x,1)dx \; + \int_{1}^{-1} J_\perp(1,y)dy \; + \int_{1}^{-1} J_\perp(x,-1)dx \; + \int_{-1}^{1} J_\perp(-1,y)dy \\ &= J_0 - J_0 = 0, \end{aligned} \tag{33}$$

where dl is a line element and the properties of the Dirac delta are used [18]. From the definition of current

$$\Delta J = \frac{dQ}{dt},$$

where Q is the net charge on the sample and J is the net current passing, one can deduce that since $\Delta J = 0$ (Eq. 33 must be true for any boundary dl with such current distribution), Q is constant. Thus, $\sigma_Q = \frac{dQ}{dS}$, where dS is an infinitesimal area element, is also constant.

B Python Code

The three functions below were used in the calculation of the stiffness matrix on each element.[5]

```
import numpy as np
import cupy as cp # you can change all 'cp' with 'np' and it will work on CPU

def calculate_ke(pts, tri):
    '''
    function that calculates the element stiffness matrix on each element
```

[5] 'cp' stands for 'cupy' [16].

```
    takes:

    pts - array that contains the coordinates of all nodes in the mesh - shape
        (n_nodes, 2)
    tri - array with all indices (in pts array) of triangle vertices - shape
        (num_triangles, 3)

    returns:

    ke_array - an array of stiffness matrices for all elements (n_triangles, 3, 3)
    '''

    n_tri, n_vertices = tri.shape
    ke_array = cp.zeros((n_tri, n_vertices, n_vertices))
    coord = pts[tri[:,:]]
    ke_array[:] = triangle_ke(coord)
    return ke_array

def triangle_ke(coord):
   '''
   function that calculates ke

   takes:

   coord - coordinates of each triangle's nodes - shape (n_triangles, 3, 2)

   returns:

    ke_array - an array of stiffness matrices for all elements (n_triangles, 3, 3)
   '''
    s = cp.array(coord[:, [2, 0, 1]] - coord[:, [1, 2, 0]])
    ke_matrix = cp.empty((len(coord), 3, 3))
    area = cp.abs(0.5 * det2x2(s[:, 0], s[:, 1]))
    ke_matrix[:] = cp.einsum('ijk,kli->ijl', s, s.T) / (4. * area[:, None, None])
    return ke_matrix

def det2x2(x1, x2):
   # calculate determinant of many 2D matrices at once
    return x1[:, 0]*x2[:, 1] - x1[:, 1]*x2[:, 0]
```

The function below was used to calculate the global stiffness matrix from all the local stiffness matrices.

```
import numpy as np
from scipy import sparse
import cupy as cp # you can change all 'cp' with 'np' and it will work on CPU
from cupyx.scipy import sparse as sp # also need to change 'sp' to 'sparse'

def assemble_sparse(ke, tri, perm, n_pts, ref=0):
    '''
    function that assembles the global stiffness matrix from all element stiffness
        matrices

    takes:
```

```
    ke - stiffness on each element matrix - array shape (n_triangles, n_vertices,
        n_vertices)
    tri - array with all indices (in pts array) of triangle vertices - shape
        (num_triangles, 3)
    perm - array with permittivity in each element - array shape (num_triangles,)
    n_pts - number of nodes - int
    ref - electrode on which reference value is placed

    returns:

    K - global stiffness matrix - (n_pts, n_pts)
    '''

    n_tri, n_vertices = tri.shape
    row = cp.tile(tri, (1, n_vertices))
    i = cp.array([0, 3, 6, 1, 4, 7, 2, 5, 8])
    row = row[:, i].ravel()
    col = cp.tile(tri, (n_vertices)).reshape((tri.shape[0] * tri.shape[1] *
        n_vertices))
    i = cp.arange(n_tri)
    data = cp.multiply(ke[i], perm[i, None, None]).ravel()
    ind = cp.argsort(row)
    row = row[ind]
    col = col[ind]
    data = data[ind]
    unique, counts = cp.unique(row, return_counts=True)
    index_pointer = cp.zeros(n_pts + 1)
    sum_count = cp.cumsum(counts)
    index_pointer[unique[:]+1] = sum_count[:]

    K = sp.csr_matrix((data, col, index_pointer), shape=(n_pts, n_pts),
        dtype=perm.dtype)

    K = K.toarray()

    if 0 <= ref < n_pts:
        K[ref, :] = 0.
        K[:, ref] = 0.
        K[ref, ref] = 1.

    return K
```

The function below was used to generate all true conductivity distributions from a given anomaly dictionary in the right format (see next code snippet).

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la
import numpy.random as rand
import cupy as cp
def generate_examplary_output(a, npix, anomaly, centre=None):
    '''
    a function that generates true conductivity map to be used in training of U-net
```

```
takes:

a - side of square - float
npix - number of pixels along each axis - int
anomaly - dictionary of anomalies characteristics
centre - centre of coordinate system - array shape (2,)

returns:

true conductivity distribution - array shape (npix, npix)
'''
if centre is None:
    centre = [0, 0]
# array to store permitivity in different square
# assumed that background permitivity is 1 (and therefore difference with uniform
    will be 0)
perm_sq = np.zeros((npix, npix), dtype='f4')
# initialises an array to store the coordinates of centres of squares (pixels)
C_sq = np.empty((npix, npix, 2), dtype='f4')
# initialising the j vector to prepare for tiling
j = np.arange(npix)
# tiling j to itself npix times (makes array shape (npix, npix))
j = np.tile(j, (npix, 1))
# i and j are transposes of each other
i = j
j = j.T
# assigning values to C_sq
C_sq[i, j, :] = np.transpose([a / 2 * ((2 * i + 1) / npix - 1) + centre[0], a / 2
    * ((2 * j + 1) / npix - 1) + centre[1]])
# return an empty array if there are no anomalies generated
if anomaly is None:
  return perm_sq
# putting anomalies on map one by one
for l in range(len(anomaly)):
    if anomaly[l]['name'] == 'ellipse':
    # check what squares have their centres inside the ellipse and set permittivity
        values
        x = anomaly[l]['x']
        y = anomaly[l]['y']
        a_ = anomaly[l]['a']
        b = anomaly[l]['b']
        angle = anomaly[l]['angle']
        # equation for a rotated ellipse in 2d cartesians
        indices = np.sum(np.power([(np.cos(angle)*(C_sq[:, :, 0] - x) -
            np.sin(angle) * (C_sq[:, : , 1] - y))/a_,
                                   (np.sin(angle)*(C_sq[:, :, 0] - x) + np.cos(angle)
                                       * (C_sq[:, : , 1] - y))/b], 2),
                                   axis=0) < 1
        # setting relative permittivity values
        perm_sq[indices] = anomaly[l]['perm'] - 1.

    # check what squares are crossed by the line element and set their permittivity
        values to zero
```

```
elif anomaly[l]['name'] == 'line':
    x = anomaly[l]['x']
    y = anomaly[l]['y']
    theta = anomaly[l]['angle_line']
    length = anomaly[l]['len']
    # coordinates of endpoints of the line
    p_start = np.array([x + (length * np.cos(theta))/2, y + (length *
        np.sin(theta))/2])
    p_end = np.array([x - (length * np.cos(theta))/2, y - (length *
        np.sin(theta))/2])
    # find min and max x and y for any coordinates, so we have lower left and
        upper right corners of rectangle, whose diagonal is our line
    x_min_max = np.sort([x + (length * np.cos(theta))/2, x - (length *
        np.cos(theta))/2])
    y_min_max = np.sort([y + (length * np.sin(theta))/2, y - (length *
        np.sin(theta))/2])
    # checking whether pixel is in that rectangle by setting a limit on x and y
        of its centre
    if abs(y_min_max[0] - y_min_max[1]) < a / (npix/4):
    # the loop increases the allowed distances from the line if line is very
        close to horizontal
        index_1 = (C_sq[:,:,0] > x_min_max[0]) * (C_sq[:,:,0] < x_min_max[1]) *
            (C_sq[:,:,1] > y_min_max[0] - a / (npix * np.sqrt(2))) *
            (C_sq[:,:,1] < y_min_max[1] + a / (npix * np.sqrt(2)))
    elif abs(x_min_max[0] - x_min_max[1]) < a / (npix/4):
    # the loop increases the allowed distances from the line if line is very
        close to vertical
        index_1 = (C_sq[:,:,0] > x_min_max[0] - a / (npix/4)) * (C_sq[:,:,0] <
            x_min_max[1] + a / (npix/4)) * (C_sq[:,:,1] > y_min_max[0]) *
            (C_sq[:,:,1] < y_min_max[1])
    else:
        index_1 = (C_sq[:,:,0] > x_min_max[0]) * (C_sq[:,:,0] < x_min_max[1]) *
            (C_sq[:,:,1] > y_min_max[0]) * (C_sq[:,:,1] < y_min_max[1])

    # checking whether distance from the centre to the line is smaller than the
        diagonal of the square
    indices = (np.absolute(np.cross(p_end - p_start,
                        np.array([p_start[0] - C_sq[:, :, 0], p_start[1] -
                            C_sq[:, :, 1]]).T)
              / la.norm(p_end - p_start))
              < a / (npix * np.sqrt(2)))
    indices = np.transpose(indices)
    # combining the two conditions 1)square in rectangle with line as diagonal
        and 2) distance to line smaller than half diagonal of pixel
    indices = np.multiply(indices, index_1)
    # setting permittivity values (relative)
    perm_sq[indices] = anomaly[l]['perm'] - 1.
elif anomaly[l]['name'] == 'triangle':
#extracting the coordinatec of each of the vertices
    A = anomaly[l]['A']
    B = anomaly[l]['B']
    C = anomaly[l]['C']
#for each point check whether the sum of the areas of the triangles formed by
    it and all combinations of
```

```
        #two of the vertices of the triangle is approximately equal to the area of the
            triangle
            index_tri = triangle_area([A[0], B[0], C_sq[:, :, 0]], [A[1], B[1], C_sq[:,
                :, 1]]) + triangle_area([B[0], C[0], C_sq[:, :, 0]], [B[1], C[1],
                C_sq[:, :, 1]]) + triangle_area([C[0], A[0], C_sq[:, :, 0]], [C[1],
                A[1], C_sq[:, :, 1]]) - triangle_area([A[0], B[0], C[0]], [A[1], B[1],
                C[1]]) < 0.01
        #set permitivity of triangle equal to the permittivity of the anomaly
            perm_sq[index_tri] = anomaly[l]['perm'] - 1
    #check whether any anomalies are too close to each of the edges
    indexx1 = (np.absolute(C_sq[:, :, 0] + a/2) < 0.15)
    indexx2 = (np.absolute(C_sq[:, :, 0] - a/2) < 0.15)
    indexy1 = (np.absolute(C_sq[:, :, 1] + a/2) < 0.15)
    indexy2 = (np.absolute(C_sq[:, :, 1] - a/2) < 0.15)
    #combine all edge conditions
    index = (indexx1 + indexx2 + indexy1 + indexy2)
    #check for permitivities close to 0
    index_p = perm_sq < -0.9
    index = np.multiply(index_p, index)
    index = index > 0
    #set such conditions equal to the background to ensure existing solution to
        forward problem
    perm_sq[index] = 0.
    '''
    #optional plot conductivity distribution as a check
    plt.imshow(perm_sq, cmap=plt.cm.viridis, origin='lower', extent=[-1, 1, -1, 1])
    plt.colorbar()
    #plt.show()
    '''
    return perm_sq
```

The function below was used to generate a random anomaly and create a special format dictionary.

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la
import numpy.random as rand
import cupy as cp
def generate_anoms(a, b):
    '''
    a function that generates an array of a random number of dictionaries for the
        different anomalies;

    also randomises the type and characteristics of the different anomalies

    takes:

    a - length of side on x axis - float
    b - length of side on y axis - float

    returns:

    anoms - array of dictionaries with information about anomalies - array shape
```

```
    (number of anomalies); type(anoms[0]): object

'''
n_anom = int(np.round_(np.absolute(np.random.normal(1.5, 1.))))
# stop the function if n_anom == 0
if n_anom == 0:
    return None
# initialises an empty array of objects(dictionaries) with length n_anom
anoms = np.empty((n_anom,), dtype='O')
# chooses types of anomalies, (ellipse ( == 0) with probability 0.5, line ( == 1 )
    with probability 0.1 and triangle ( == 2) with probability 0.4)
names = rand.choice([0, 1, 2], n_anom, p=[0.5, 0.1, 0.4])
# randomises characteristics of each anomaly and assigns dictionary to the anoms
    array
for i in range(n_anom):
    if names[i] == 0:
        # x, y from -a to a or -b to b
        x = a * rand.random() - a/2
        y = b * rand.random() - b/2
        # a, b from 0 to a and 0 to b
        a_ = a * rand.random()
        b_ = b * rand.random()
        # angle from 0 to PI
        ang = np.pi * rand.random()
        #modulus to ensure positive values 0.25 times a random number + number from
            [0,1] based of power law of 1.2 + 0.5 base offset to prevent
            conductivities too low
        #main goal is to have a bias to higher conductivities since this is what we
            expect in real data
        perm = np.absolute(0.25 * rand.randn() + rand.power(1.2) + 0.5)
        # add anomaly to array
        anoms[i] = {'name': 'ellipse', 'x': x, 'y': y, 'a': a_, 'b': b_ ,'angle':
            ang, 'perm': perm}
    if names[i] == 1:
        # x, y from -a to a or -b to b
        x = a * rand.random() - a/2
        y = b * rand.random() - b/2
        # diagonal of the given line
        l = 0.5 * np.sqrt(a ** 2 + b ** 2)
        # total length of line anomaly ( with pdf proportional to 1.4) + constant
            length value
        #prevents existence of lines too short and overfitting of CNN
        length = l * (np.random.power(1.4) + 0.14)
        # angle from 0 to PI
        ang = np.pi * rand.random()
        # add anomaly to array
        anoms[i] = {'name': 'line', 'len': length, 'x': x, 'y': y, 'angle_line':
            ang, 'perm': 0.00001}
    if names[i] == 2:
        #two individual choices of triangles (one with random angles that are not
            too small with probability 70% and
        #one with angles 120/60 deg to capture the symmetry of the graphene sample)
        tri_type = rand.choice([0, 1], p = [0.7, 0.3])
        if tri_type == 0:
```

```
        #initialise cosines to be one rad.
        cosa = np.ones(3)
        while (np.absolute(cosa) - 0.5).all() > 0.01 and (np.absolute(cosa) -
            np.sqrt(3)/2).all() > 0.01:
            #initialise coordinates of two of the points
            Ax = a * rand.random() - a/2
            Ay = b * rand.random() - b/2
            Bx = a * rand.random() - a/2
            By = b * rand.random() - b/2
            #randomise which root to take since two such triangles will be
                possible for a set of 2 points
            sign = rand.choice([-1, 1])
            #calculate length of line connecting A/B
            l = np.sqrt(np.power(Ax - Bx, 2) + np.power(Ay - By, 2))
            #randomise length of second side of triangle (constant factor added
                to ensure existence of this specific triangle)
            l2 = np.sqrt(3) * l / 2 + rand.random()
            #calculate differences of y's of A/B
            diff = np.absolute(Ay - By)
            #initialise angle of 60 deg
            ang = 1/3 * np.pi
            #calculate angle needed to determine y coord of C
            beta = np.pi - np.arcsin(diff/l) - ang
            #Calculate coordinates of C[x, y]
            Cy = Ay - np.sin(beta)*l2
            Cx = Ax - np.sqrt(np.power(l2, 2) - np.power(Ay - Cy, 2))
            #iterate over created triangles to ensure desired result given lack
                of general formulation
            a_sq = np.power(Ax - Bx, 2) + np.power(Ay - By, 2)
            b_sq = np.power(Bx - Cx, 2) + np.power(By - Cy, 2)
            c_sq = np.power(Cx - Ax, 2) + np.power(Cy - Ay, 2)
            N1 = a_sq + b_sq - c_sq
            D1 = 2 * np.sqrt(a_sq) * np.sqrt(b_sq)
            N2 = b_sq + c_sq - a_sq
            D2 = 2 * np.sqrt(b_sq) * np.sqrt(c_sq)
            N3 = c_sq + a_sq - b_sq
            D3 = 2 * np.sqrt(c_sq) * np.sqrt(a_sq)
            cosa = ([np.true_divide(N1, D1), np.true_divide(N2, D2),
                np.true_divide(N3, D3)])

    elif tri_type == 1:
        #initialise cosine in order to enter loop
        cosa = 1
        #interate over created triangles to ensure no silver triangles
        while np.absolute(cosa) > np.sqrt(2)/2:
            #initialise coordinates randomly
            Ax = a * rand.random() - a/2
            Ay = b * rand.random() - b/2
            Bx = a * rand.random() - a/2
            By = b * rand.random() - b/2
            Cx = a * rand.random() - a/2
            Cy = b * rand.random() - b/2
            #calculate length of each side
            a_sq = np.power(Ax - Bx, 2) + np.power(Ay - By, 2)
```

```
                b_sq = np.power(Bx - Cx, 2) + np.power(By - Cy, 2)
                c_sq = np.power(Cx - Ax, 2) + np.power(Cy - Ay, 2)
                #calculate the maximum angle in triangle using cosine rule
                N1 = a_sq + b_sq - c_sq
                D1 = 2 * np.sqrt(a_sq) * np.sqrt(b_sq)
                N2 = b_sq + c_sq - a_sq
                D2 = 2 * np.sqrt(b_sq) * np.sqrt(c_sq)
                N3 = c_sq + a_sq - b_sq
                D3 = 2 * np.sqrt(c_sq) * np.sqrt(a_sq)
                cosa = np.amax([np.true_divide(N1, D1), np.true_divide(N2, D2),
                    np.true_divide(N3, D3)])
            #initialise perimitivity of triangle (constant value + pdf proportional to
                power 1.2 in the range [0,1] and another random constant factor)
            perm = np.absolute(0.25 * rand.randn() + rand.power(1.2) + 0.5)
            #return anomaly to the array of anomalies
            anoms[i] = {'name': 'triangle', 'A': [Ax, Ay], 'B': [Bx, By], 'C': [Cx,
                Cy], 'perm': perm}
    #return the whole dictionary
    return anoms
```

The function below was called in the function that generates a true conductivity map (see above).

```
def triangle_area(x, y):
    '''
    function that finds area given 2d coordinates of all vertices of triangle

    takes:

    x - array storing the x-coordinates of all vertices [3, 1] float
    y - array storing the y-coordinates of all vertices [3, 1] float

    returns:
    area of the triangle
    '''
    return 0.5 * np.absolute(x[0] * (y[1] - y[2]) + x[1] * (y[2] - y[0]) + x[2] *
        (y[0] - y[1]))
```

The function below was our implementation of the electrode selection algorithm. The 'influence_mat' was referred to as the sensitivity matrix and the 'findMinimumTriArea' function finds the area of the triangles formed by each electrode pair and point of low sensitivity in vectorised form.

```
import numpy as np
import greit_rec_training_set as train
import cupy as cp
import h5py as h5
import matplotlib.pyplot as plt
def findNextPair(fileJac, ne, anomaly=None, el_dist=None, meas=None, a=2., npix=64,
    pert=0.05):
   # find indices in the already calculated const. permitivity Jacobian (CPJ)
   if meas is None:
      if el_dist is None:
         el_dist = np.random.randint(1, 20)
      A = cp.arange(ne)
```

```
        B = (cp.arange(ne) + el_dist) % ne
        ex_mat = cp.stack((A, B), axis=1)
        #test to see how different excitation matrices will affect the sensitivity in
            different areas
        volt_mat, ex_mat, ind_new = voltMeterwStep(ne, ex_mat, step_arr=10)
        measurements = cp.concatenate((ex_mat, volt_mat), axis=1)
    else:
        measurements = meas
    rec, h_mat, volts, volt_const, tr, num = simulateMeasurements(fileJac,
        anomaly=anomaly, measurements=measurements)
    v_pert = np.empty(shape=(len(volts), len(volts)))
    perturbing_mat = np.ones((len(volts), len(volts))) + pert * np.identity(len(volts))
    v_pert[:] = np.dot(perturbing_mat, np.diag(volts))
    influence_mat = -np.dot(h_mat, v_pert).reshape(npix, npix, len(volts)) - rec[:, :,
        None]
    influence_mat = np.absolute(influence_mat)
    influence_mat = np.sum(influence_mat, axis=2)

    mask = circleMask(npix, a)
    influence_mat[~mask] = np.amax(influence_mat)
    zones = np.less(influence_mat, 2 * np.amin(influence_mat))
    zone_index = np.where(zones)

    zone_index = np.concatenate((zone_index[1][:, None], zone_index[0][:, None]),
        axis=1)
    zone_coords = (zone_index * a / npix + a / (2 * npix)) - a / 2
    avg_coords = np.mean(zone_coords, axis=0)
    closest = np.argsort(np.linalg.norm(zone_coords - avg_coords[None], axis=1))
    closest_coords = zone_coords[closest[:3]]
    return findMinimumTriArea(closest_coords), rec, tr, num


def findMinimumTriArea(closest_coords):
    el_coords = train.fix_electrodes(edgeX=0.2, edgeY=0.2, a=2, b=2, ppl=20)[1]
    ex_mat_all = train.generateExMat(ne=20)
    coord_mat = el_coords[ex_mat_all]

    tri_area_sum = np.sum(0.5 * np.absolute(coord_mat[:, None, 0, 0] * (coord_mat[:,
        None, 1, 1] - closest_coords[None, :, 1]) - coord_mat[:, None, 0, 1] *
        (coord_mat[:, None, 1, 0] - closest_coords[None, :, 0]) + coord_mat[:, None, 1,
        0]*closest_coords[None, :, 1] - coord_mat[:, None, 1, 1] * closest_coords[None,
        :, 0] ), axis=1)
    min_area_ind = np.argsort(tri_area_sum)
    ex_mat_pred = ex_mat_all[min_area_ind[:10]]
    #print('Next src/sink recommendation (from 1 to n_el starting from upper left
        corner clockwise):', ex_mat_pred + 1)
    return ex_mat_pred
```