EIT Simulations: An Optimised Electrode Selection Algorithm and Sensitivity Test of a New Forward Solver

Ivo Mihov

9918179

The University of Manchester

11th May 2020

This experiment was performed in collaboration with *Vasil Avramov*.

Abstract

In this project Electrical Impedance Tomography (EIT) conductivity reconstructions of 2-dimensional simulated samples were performed. A new forward solver was implemented in Python and presented. It is an upgrade of an old forward solver that is described in [1]. Furthermore, a novel Electrode Selection Algorithm was devised and the reconstructions of series of recommended measurements were compared against the ones of common measurement techniques. Due to the limited applications of EIT, the potential for non-medical uses was analysed. The possibility to use the sensitivity of graphene samples to find their specific characteristics was analysed using the newly built forward solver. It was found that reconstructions made with the novel Electrode Selection Algorithm outperformed two commonly used measurement techniques on sets of 1500 samples.

Contents

1 Introduction

Electrical Impedance Tomography (EIT) gained popularity in the end of the 20th century as a prospective replacement of other imaging techniques such as CT scans and MRIs [2]. One reason was its harmlessness due to the small currents and the lack of ionising radiation [2]. However, the resolution of the reconstruction algorithms is limited by the non-linearity of the Inverse Problem in EIT. Numerical models are used due to the complexity of an analytical solution [3]. This hindered its development, although some uses have been recognised, e.g. for prostate scans [4].

In EIT, electrodes are attached to the surface/boundary of the analysed object [3]. During a single measurement, current flows between two of the electrodes, as other two measure the potential difference [3]. The process is repeated while changing the used electrodes, until enough information for the conductivity reconstruction is collected. In this project a square 2D sample with 5 electrodes on each side (20 in total) is simulated. The simulations are used to test a newly built Electrode Selection Algorithm (ESA) and to analyse the potential of EIT in non-medical applications, e.g. detection of graphene crystal formation.

2 Formulation of EIT Problem

The aim of EIT is to reconstruct the conductivity distribution inside a sample, based on the surface voltage measurements. Conductivity is defined as the inverse of resistivity

$$\sigma = \frac{1}{\rho}, \tag{1}$$

and the resistivity of a conductor with uniform cross section (e.g. a wire) can be expressed by

$$\rho = R\frac{A}{l},$$

where R is the overall resistance of the conductor, A is its cross section and l is its length [5]. In other words, resistivity is the continuum extension of the resistance [5]. Ohm's law for discrete resistors is expressed as

$$I = \frac{V}{R},$$

where I is the current, V is the voltage and this formula applies over a single resistor [5]. Using Eq. 1, one can show that the continuous representation of the Ohm's law is given by

$$\mathbf{J} = \frac{\mathbf{E}}{\rho} = \sigma\mathbf{E}, \tag{2}$$

where $\mathbf{J}$ is the current density in the sample, $\mathbf{E}$ is the electric field in the sample and σ is the conductivity distribution, which is in general a rank-2 tensor [5]. However, the samples in this study are all isotropic, thus σ is treated as a scalar function.

Maxwell's extension of the Ampere's law can be expressed as

$$\nabla \times \mathbf{B} = \mu_0 \left(\mathbf{J} + \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}\right), \tag{3}$$

where $\mathbf{B}$ is the magnetic field vector, μ_0 is the magnetic permeability of free space, $\mathbf{J}$ is the current density, ϵ_0 is the electric permittivity of free space and $\frac{\partial \mathbf{E}}{\partial t}$ is the time derivative of the electric field vector [5]. If the divergence of the entire equation is taken and the result is rearranged, we yield

$$\nabla \cdot \mathbf{J} = \frac{1}{\mu_0}\left(\nabla \cdot (\nabla \times \mathbf{B}) - \mu_0 \epsilon_0 \frac{\partial (\nabla \cdot \mathbf{E})}{\partial t}\right), \tag{4}$$

since the spatial derivatives are independent of time (inertial frame). By using that the divergence of the curl of a vector field is zero and using Gauss' law in 2 dimensions

$$\nabla \cdot \mathbf{E} = \frac{\sigma_Q}{\varepsilon_0} \text{ [5]}, \tag{5}$$

where σ_Q is the surface charge density, we cancel the first term on the right and substitute, to yield

$$\frac{\partial \sigma_Q}{\partial t} + \nabla \cdot \mathbf{J} = 0. \tag{6}$$

Using Eq. 2, we can substitute $\mathbf{J}$ and show that

$$\frac{\partial \sigma_Q}{\partial t} + \nabla \cdot (\sigma \nabla u) = 0, \tag{7}$$

where u is the voltage potential field. During an EIT measurement, the current that enters the sample through the current source is equal to the one that leaves through the sink, so the charge density is constant everywhere, except at the source and sink (only if sample is static) [3]. Globally, one can use

$$\nabla \cdot (\sigma \nabla u) = 0, \tag{8}$$

since the total charge on the sample is constant [3].

By solving this equation for a given conductivity distribution, one can simulate the voltages that would be measured on a sample with this conductivity distribution, σ. This is known as solving the forward problem. Besides σ, one needs boundary conditions to solve Eq. 8. The complete electrode model takes into account the contact impedances between the electrodes and the sample by using boundary conditions of the form

$$u + z_n \sigma \frac{\partial u}{\partial \mathbf{n}} = V_n \text{ for } x \in e_n \tag{9}$$

where z_n is the contact impedance between the n-th electrode and the material, $\mathbf{n}$ is the normal to the boundary, e_n is the n-th electrode and V_n is the voltage measured on e_n or simulated by the forward solver [3]. The current density along all points on the boundary that are not part of the electrodes satisfies

$$\mathbf{J} = -\sigma \nabla u = 0 \text{ for } x \in \partial\Omega \text{ and } x \notin e_n \text{ [3]}.$$

Also, one can show that

$$\int_{e_n} \sigma \nabla u \cdot \mathbf{n} ds = \int_{e_n} \sigma \frac{\partial u}{\partial \mathbf{n}} ds = \mathbf{I_n},$$

where the integral is over an active electrode e_n and $\mathbf{I_n}$ is the current applied through that electrode [6]. In the case where e_n is not an active electrode, $\mathbf{I_n}$ is zero.

The final part of the forward problem is finding the Jacobian which gives a measure of how much the results from one measurement affect the conductivity in the sample. It can be found by applying the Galerkin method to Eq. 8 again [7] and using Green's theorem

$$\int_{\Omega} \sigma \nabla u \cdot \nabla u \, \mathrm{d}s = \int_{\partial\Omega} u \sigma \frac{\partial u}{\partial \mathbf{n}} \, \mathrm{d}l, \tag{10}$$

where the inner product of Eq. 8 with u was taken this time [8]. From Eq. 10 and using Eq. 9, we can show that

$$\begin{aligned}\int_{\Omega} \sigma |\nabla u|^2 \,\mathrm{d}s &= \int_{\partial\Omega} u\sigma \frac{\partial u}{\partial \mathbf{n}} \,\mathrm{d}l \\ &= \sum_m \int_{e_m} \left(V_m - z_m \sigma \frac{\partial u}{\partial \mathbf{n}} \right) \sigma \frac{\partial u}{\partial \mathbf{n}} \,\mathrm{d}l,\end{aligned} \tag{11}$$

where the sum is over all electrodes [8]. This can be rearranged to get

$$\int_{\Omega} \sigma |\nabla u|^2 \,\mathrm{d}s + \sum_m \int_{e_l} z_m \left(\sigma \frac{\partial u}{\partial \mathbf{n}} \right)^2 \mathrm{d}l = \sum_m V_m I_m, \tag{12}$$

which shows that some of the power delivered by the current is turned into heat at the contact between the electrode and the sample, and the rest is transferred to the sample [8]. Making small perturbations in the conductivity, potential and voltage and taking only the first order, we yield

$$\begin{aligned}&\int_{\Omega} \delta\sigma |\nabla u|^2 \,\mathrm{d}s + 2\int_{\Omega} \sigma \nabla u \cdot \nabla \delta u \,\mathrm{d}s \\ &+ 2\sum_m \int_{e_m} z_m \left(\sigma \frac{\partial u}{\partial \mathbf{n}} \right) \delta \left(\sigma \frac{\partial u}{\partial \mathbf{n}} \right) \mathrm{d}l = \sum_m I_m \delta V_m \quad [8].\end{aligned} \tag{13}$$

By taking the total derivative of Eq. 9,

$$\delta \left(\sigma \frac{\partial u}{\partial \mathbf{n}} \right) = \frac{1}{z_m} \left(\delta V_m - \delta u \right), \tag{14}$$

the following expression is obtained

$$\begin{aligned}&\int_{\Omega} \delta\sigma |\nabla u|^2 \,\mathrm{d}s + 2\int_{\partial\Omega} \delta u \sigma \frac{\partial u}{\partial \mathbf{n}} \,\mathrm{d}l - 2\sum_m z_m \int_{e_m} \frac{\delta u}{z_m} \sigma \frac{\partial u}{\partial \mathbf{n}} \,\mathrm{d}l \\ &+2\sum_m \delta V_m \int_{e_m} \sigma \frac{\partial u}{\partial \mathbf{n}} \,\mathrm{d}l = \sum_m I_m \delta V_m \quad [8].\end{aligned} \tag{15}$$

Here we can see that the second and third term on the left-hand side cancel, since the integral over the boundary is non-zero only along the electrodes. Rearranging, we get

$$\sum_m I_m \delta V_m = -\int_{\Omega} \delta\sigma |\nabla u|^2 \,\mathrm{d}s. \tag{16}$$

If we consider the different electrodes with current $I_m = \delta_{im}$, where i runs over the electrodes, the left-hand side only has a value at the current source and sink [8]. There are two potential distributions that are obtained when two different current driving pairs are used, u_i and u_j. By substituting for the sum and the difference of the two potentials into Eq. 16 and then subtracting, as shown in [8], one obtains

$$J_{ij} = \frac{\partial V_{ij}}{\partial \sigma} = -\int_{\Omega} \nabla u_i \cdot \nabla u_j \,\mathrm{d}s, \tag{17}$$

which is the desired result for the Jacobian of the measurement [8].

Once the Jacobian is calculated, the inverse problem uses it to solve Eq. 8 for the conductivity, σ, with the obtained voltage measurements.

3 Numerical Method

The main objective in the forward problem is the calculation of the potential that would be measured for a certain conductivity distribution by using Eq. 8. In experimental EIT, the forward problem is solved by nature and the voltages are simply measured. However, the simulation of the voltages is not straightforward. This is due to the second-order partial differential equation, Eq. 8. It is highly computational to solve it analytically. As a result, one standard approach is to use the finite element method (FEM) for a numerical solution. To apply it, one first needs to discretise the space into a mesh. In 2D, the simplest shape that can be used is the triangle, but other polygons are also possible.

3.1 Meshing

The meshing algorithm that is used to discretise the space in this project relies on Delaunay triangulation [9]. This technique aims to connect a given set of points to form triangles such that no point is inside a circumscribed circle about any triangle [10]. This is achieved by using the Voronoi diagram [10]. It is constructed by forming a polygon around each point, such that each polygon contains all points that are closer to the point inside it than to any other point [10]. Once this is done, all points whose polygons share a border are connected and the Delaunay triangulation is complete [10].

When the triangles are generated, the meshing technique seeks to normalise the lengths of all edges. This is accomplished by defining so-called bar forces [11]. These are vectors whose magnitudes are proportional to the lengths of the bars (edges) that point along the respective edges [11]. The algorithm aims to reach a certain magnitude for all bar forces in the meshing. The target magnitude is smaller the closer the edge is to an electrode. Whenever the magnitude is larger than desired, the algorithm reduces the length of the bar by moving the node that defines it opposite to the direction of the bar force [11]. This is performed iteratively until a stopping criterion is reached or until the algorithm converges.

3.2 Finite Element Method

The FEM assumes that the potential can be discretised and its assumed form inside an element is given by

$$u_{FEM} = \sum_{i=1}^{N} f_i \phi_i,$$

where ϕ_i is the shape function on the i-th node, N is the number of nodes (equal to 3 for triangles) and f_i are the coefficients of the i-th shape function [12]. The shape functions are identical to the barycentric coordinates in the case of triangles -

$$\phi_i = \begin{cases} 1 & \text{on node } i \\ 0 & \text{on the other nodes.} \end{cases} \tag{18}$$

Since our aim is to obtain

$$u_{FEM} \approx u,$$

we can set

$$u - \sum_{i=1}^{N} f_i \phi_i = 0$$

and integrate

$$(\nabla \cdot \sigma \nabla (u - u_{FEM})) = 0$$

over all surface and then apply the Galerkin method [6, 7]:

$$\begin{aligned}
&\int_\Omega \phi_i \left(\nabla \cdot \sigma \nabla \left(u - u_{FEM}\right)\right) \, \mathrm{d}s = \\
&\int_\Omega \phi_i \left(\nabla \cdot \sigma \nabla u_{FEM}\right) \, \mathrm{d}s = 0,
\end{aligned} \tag{19}$$

where Ω indicates that the integration is over the surface where the ϕ_i is defined [6]. We can use the Green's theorem on Eq. 19 to yield

$$\begin{aligned}
&\int_\Omega \phi_i \left(\nabla \cdot \sigma \nabla u_{FEM}\right) \, \mathrm{d}s = \\
&\int_\Omega \nabla \cdot \left(\phi_i \sigma \nabla u_{FEM}\right) \mathrm{d}s - \int_\Omega \nabla \phi_i \cdot \left(\sigma \nabla u_{FEM}\right) \mathrm{d}s = \\
&\int_{\partial\Omega} \phi_i \sigma \frac{\partial u_{FEM}}{\partial \mathbf{n}} \mathrm{d}l - \int_\Omega \nabla \phi_i \cdot \left(\sigma \nabla u_{FEM}\right) \mathrm{d}s = 0.
\end{aligned} \tag{20}$$

Finally, the gradient of u_{FEM} along the outward normal $\mathbf{n}$ on the left-hand side of Eq. 20 is substituted from Eq. 9 [6]. By choosing a reference voltage so that

$$\sum_n V_n = 0,$$

where the sum is over all electrodes, we yield a linear system of equations that can be expressed by

$$\mathbf{A}\mathbf{f} = \mathbf{b}, \tag{21}$$

where $\mathbf{A}$ is the admittance matrix, $\mathbf{f}$ is the vector of potentials on each node, as found by the FEM and $\mathbf{b}$ is a vector of the net current at each point:

$$\mathbf{b} = \begin{pmatrix} 0 \\ \mathbf{I} \end{pmatrix}$$

where I_i is the net current at the i-th electrode (i runs over all electrodes) [6,8]. The admittance matrix can be divided further into 4 parts:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{12}^{\mathrm{T}} & \mathbf{A}_{22} \end{pmatrix} \tag{22}$$

where the submatrices are defined as

$$\begin{aligned}
[\mathbf{A}_{11}]_{ij} &= [\mathbf{A}_{11,\ \mathrm{cond}}]_{ij} + [\mathbf{A}_{11,\ \mathrm{el}}]_{ij} \\
&= \int_\Omega \sigma \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}s + \sum_{n=1}^{N_e} \frac{1}{z_n} \int_{e_n} \phi_i \phi_j \, \mathrm{d}l \quad \text{where } i, j \in [1, N_n];
\end{aligned} \tag{23}$$

$$[\mathbf{A}_{12}]_{ij} = -\sum_t^{N_{t,i}} \frac{1}{z_j} \int_{e_j} \phi_{i,t} \, \mathrm{d}l \quad \text{where } i \in [1, N_n] \text{ and } j \in [1, N_e]; \tag{24}$$

$$[\mathbf{A}_{22}]_{ii} = \frac{\delta_{ni}}{z_n} \int_{e_n} \mathrm{d}l \quad \text{where } i \in [1, N_e], \tag{25}$$

where δ_{ni} is the Kronecker Delta (1 if i corresponds to an electrode and 0 otherwise) [8]. N_e and N_n represent the number of electrodes and the number of nodes respectively. All integrals over dl are over the length of an electrode [8]. The sum over n runs over all N_e electrodes, and e_n indicates that the integration is over the n-th electrode. An important aspect of Eq. 24 is that the sum over t runs over all triangles that share node i. If one imagines the case where two triangles share a node and have edges that lie on the same electrode. Then, the two shape functions of the node should be integrated separately along the edge of the triangle that they are defined in. This way the sum of the two integrals corresponds to the element ij in $\mathbf{A}_{12}$ [13]. Furthermore, the elements in the submatrices must satisfy the following conditions:

$$\begin{aligned}
&[\mathbf{A}_{11,\text{ cond}}]_{ii} > 0, [\mathbf{A}_{11,\text{ el}}]_{ii} > 0, \text{ where } i \in [1, N_n];\\
&[\mathbf{A}_{22}]_{ii} > 0 \text{ where } i \in [1, N_e];\\
&[\mathbf{A}_{11,\text{ el}}]_{ij} \geq 0 \text{ where } i, j \in [1, N_n];\\
&[\mathbf{A}_{12}]_{ij} \leq 0 \text{ where } i \in [1, N_n], j \in [1, N_e] \;\; [13].
\end{aligned} \tag{26}$$

After the admittance matrix is assembled, the nodal values of the potential can be extracted by using Eq. 21:

$$\mathbf{f} = \mathbf{A}^{-1}\mathbf{b}. \tag{27}$$

Once the potential on the nodes is found, the Jacobian is calculated. It can be obtained by substituting the approximate FEM potential u_{FEM} into Eq. 17. This way the FEM Jacobian is shown to be

$$J_k = \int_{\text{element } k} \nabla u_i \cdot \nabla u_j \mathrm{d}s \tag{28}$$

for two different excited electrodes i and j [14].

3.3 GREIT Algorithm

The inverse problem in EIT represents an obstacle for most algorithms, since it is neither smooth nor linear. Contrary to X-rays, current does not travel in straight lines. As a result, algorithms such as backprojection, which are commonly used in CT scans, are ineffective in EIT [15]. Alternatively, other algorithms such as the Gauss-Newton solver and the GREIT algorithm can be used [15]. These are known as difference EIT algorithms, because they only find a deviation from a reference conductivity that they cannot measure [15]. To find the magnitude of the conductivity, a reference conductivity distribution is used [15]. Then the difference between the measured conductivity and the reference $\mathbf{x} = \sigma - \sigma_r$ can be found using GREIT [15].

Effectively, GREIT is a first-order approximation of Tikhonov regularisation [1,15]. Its aim is to find a matrix $\mathbf{R} \in \mathbb{R}^{N_{el} \times N_m}$ (where N_{el} is the number of elements in meshing, and N_m is the number of performed voltage measurements) which satisfies the expression

$$\hat{\mathbf{x}} = \mathbf{R}\mathbf{V}, \tag{29}$$

where $\hat{\mathbf{x}} \in \mathbb{R}^{N_{el}}$ is the reconstructed vector of conductivity deviations and $\mathbf{V}$ is the vector of measured voltages [1,15].

The idea behind GREIT is to find $\mathbf{R}$, which minimises the L2 error

$$
\begin{aligned}
\epsilon^2 &= \sum_k \left\| \tilde{\mathbf{x}}^{(k)} - \mathbf{R}\mathbf{v}^{(k)} \right\|^2_{\mathbf{W}^{(k)}} \\
&= \sum_k \sum_i \left(\left[\tilde{\mathbf{x}}^{(k)}\right]_i^2 \left[\mathbf{w}^{(k)}\right]_i^2 - 2\left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left(\sum_j \mathbf{R}_{ij} \left[\mathbf{v}^{(k)}\right]_j \right) \right. \\
&\quad \left. + \left[\mathbf{w}^{(k)}\right]_i^2 \left(\sum_j \mathbf{R}_{ij} \left[\mathbf{v}^{(k)}\right]_j \right)^2 \right),
\end{aligned} \tag{30}
$$

where $\tilde{\mathbf{x}} \in \mathrm{I\!R}^{N_{el}}$ is the true conductivity distribution, $\mathbf{W} \in \mathrm{I\!R}^{N_{el}}$ is the vector of weights on each element, $\left(\mathbf{W}^{(k)} = \mathrm{diag}\left(\mathbf{w}^{(k)}\right)\right)$, and k runs over the training samples [1, 15]. Then $R = \mathrm{argmin}(\epsilon^2)$. By taking the derivative of Eq. 30 w.r.t. $\mathbf{R}_{ij}$ and setting it to zero, one finds that

$$
A_{ij} = \sum_l R_{il} B_{lij}, \tag{31}
$$

where $A_{ij} = \sum_k \left[\tilde{\mathbf{x}}^{(k)}\right]_i \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j$ and $B_{lij} = \sum_k \left[\mathbf{v}^{(k)}\right]_l \left[\mathbf{w}^{(k)}\right]_i^2 \left[\mathbf{v}^{(k)}\right]_j$. Thus, one can rearrange and find that

$$
\mathbf{R} = \mathbf{A}\mathbf{B}^{-1}, \tag{32}
$$

as claimed in [15].

The Jacobian $\mathbf{J}$ derived using the forward solver with the reference conductivity is then used to obtain $\mathbf{R}$ [15]. This is achieved by using

$$
\mathbf{R} = \mathbf{W}^{\mathrm{T}}\mathbf{J}^{\mathrm{T}} \left(\mathbf{J}\mathbf{J}^{\mathrm{T}} + \lambda \left(\mathbf{J}\mathbf{J}^{\mathrm{T}} \circ \mathbf{I} \right)^p \right), \tag{33}
$$

where $\mathbf{W}$ is the weighting matrix that depends on the size of the target anomaly, $\mathbf{I}$ is the identity matrix, λ is the regularisation control coefficient and p is adjusted according to the noise present in the measurement [11, 15]. The $\circ$ indicates the Hadamard (element-wise) product between the two matrices.

4 Electrode Selection Algorithm (ESA)

Scientific experiments are limited by a variety of factors. Such limitations could include the apparatus being used, the needed time to perform the tests, the size of the examined material, the accessibility of the materials etc. In EIT, measurements are limited by the time it takes to switch the current driving electrodes and measure the voltage (takes longer for higher precision). Thus, an algorithm that uses the available data to predict the next current driving electrode pair that maximises information gain would make experimental conductivity reconstruction more time-efficient. One possible implementation for such an algorithm is outlined in this section.

The proposed algorithm aims to locate the zones where a direct measurement would provide the largest amount of new information. It is based on the following main assumptions:

1. Each new voltage measurement provides additional data, although total loss could increase;

2. The probability for the GREIT algorithm to miss a deviation in the conductivity is higher close to an already located deviation (anomaly), where the gradient of the conductivity is larger.

One argument in support of the second assumption is the weighting matrix used in GREIT (see Eq. 33), which depends on the size of the target anomalies in the conductivity [15].

The suggested Electrode Selection Algorithm (ESA) is suitable for any sample shape, because it does not rely on it. Also, it is extensible to 3D with minor changes. The main objective of the algorithm is to locate the zones of interest in the sample. A direct measurement of the zones of interest has a high probability of finding a deviation from the current reconstruction. This is achieved by combining two approaches:

1. Measure close to boundary of found anomalies, according to Assumption 2;
2. Measure the zones affected by the previous measurements the least.

Two maps correspond to the approach 1 and 2 respectively: the deviation map and the influence map. Their generation is shown in Fig. 1.

The deviation map is the Hadamard product of the gradient map $\mathbf{G}$ and the element-wise logarithm of the conductivity map $\mathbf{L}$. The 2D gradient is defined by

$$G_{ij} \approx \sqrt{\left(\frac{\sigma_{i+1,j} - \sigma_{i-1,j}}{2}\right)^2 + \left(\frac{\sigma_{i,j+1} - \sigma_{i,j-1}}{2}\right)^2},$$

where i and j run over the elements of σ along the x and y axis respectively. The second order contributions are ignored. The logarithmic conductivity map is denoted by

$$L_{ij} = \log(\sigma_{ij}).$$

Thus, the deviation map $\mathbf{D}$ is given by the definition

$$\mathbf{D} = \mathbf{G} \circ \mathbf{L}^{\circ q},$$

where $\circ$ stands for the element-wise product and the superscripted $\circ q$ - for element-wise power q.

The influence map is found by performing fractional perturbations in the previously measured voltages [1]. Each measurement is perturbed separately while leaving all the other intact. Since the voltages are stored in a vector V, the perturbations result in m vectors of length m with only 1 perturbed voltage in each (m is the number of measurements). A perturbing matrix P is used to perturb all measurements at once

$$\mathbf{P}_m = \mathbf{1}_m \mathbf{1}_m^T + p\mathbf{I}_m = \begin{pmatrix} 1+p & 1 & \dots & 1 \\ 1 & 1+p & \ddots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1+p \end{pmatrix}, \tag{34}$$

where $\mathbf{1}_m$ is a vector of m ones and $\mathbf{I}_m$ is the m by m identity matrix, while p is a parameter between 0 and 1 that controls the size of the perturbations.

$$\mathbf{V}_{pert} = \text{diag}(\mathbf{V})\mathbf{P_m},$$

where $\mathbf{V}_{pert}$ is the matrix of perturbed voltages, where the k-th column is a vector of voltages $\mathbf{V}_{pert,k}$ with the k-th one perturbed ($k \in [1, m]$). Then the GREIT algorithm is used to reconstruct the conductivity with the perturbed voltages and find the difference with the current reconstruction

$$\boldsymbol{\Delta}\sigma_k = \mathbf{R}(\mathbf{V}_{pert,k} - \mathbf{V}).$$

Finally, the absolute values of all conductivity deviations $\boldsymbol{\Delta}\sigma_k$ are summed to obtain the influence map

$$\mathbf{S} = \sum_k |\boldsymbol{\Delta}\sigma_k|.$$

The deviation map and the influence map are combined to get the total map $\mathbf{T}$, which shows the zones that are favourable to measure:

$$\mathbf{T} = \mathbf{D} \circ \mathbf{S}^{\circ r}, \tag{35}$$

where the first $\circ$ indicates a Hadamard product, and the second - element-wise power r. Finally,

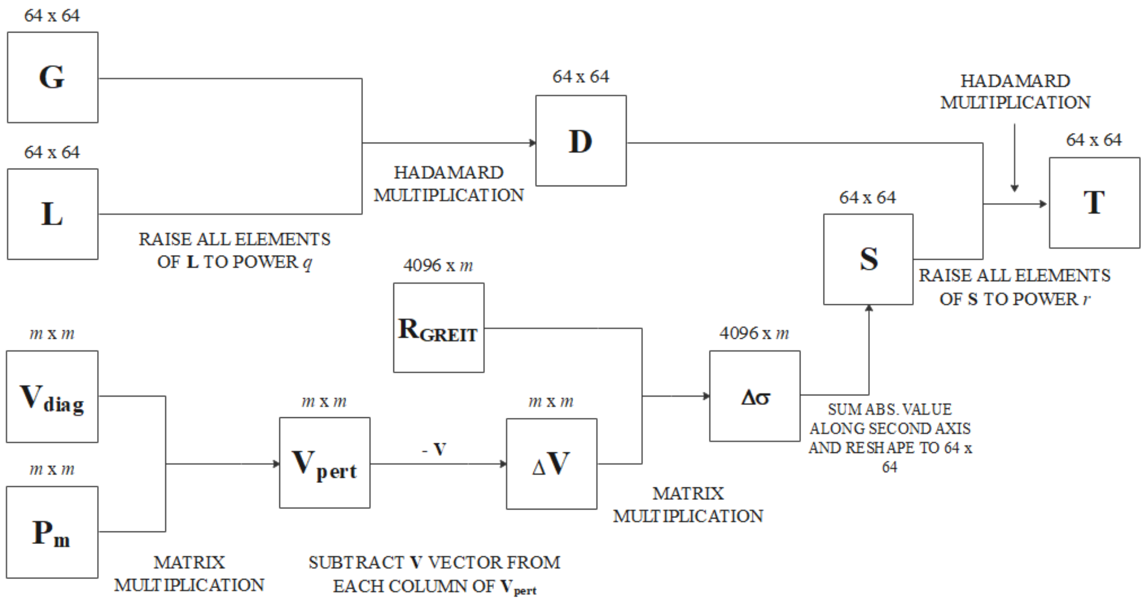


Figure 1: The above schematic shows and describes the steps in the generation of the total prediction map $\mathbf{T}$. Each square represents a matrix and its shape can be found above the square. The matrices are denoted using the same symbols that are used throughout Section 4. m stands for the number of performed measurements on the sample.

the line segments between all possible pairs of electrodes are found. The total map values of all pixels crossed by a line segment are summed to give the score of the corresponding electrode pair. The electrode pair with the highest score is then picked.

The parameters p and q are found by using Bayesian Optimisation. It is used for the optimisation of black-box functions that are expensive to evaluate (e.g. finding the location of an oil deposit) [16]. It fits Gaussian processes to the already evaluated points of the function and updates its acquisition function at each iteration [16]. The acquisition function depends on the value and standard deviation of the fitted Gaussian process at each point and shows where the next evaluation should be (at the maximum of the acquisition function) [16].

5 Implementation and Testing

This report describes the joint work of Vasil Avramov and Ivo Mihov, the two members of our team. First, a new forward solver that uses the complete electrode model was written in

Python and tested against the one made in [1]. The forward solver was used together with the meshing algorithm and GREIT implementation in the PyEIT package [11]. Secondly, a universal electrode selection algorithm was developed, optimised and tested. Finally, the sensitivity of the reconstruction algorithm was analysed by evaluating its ability to detect Gaussian anomalies of various spread and size in the conductivity of the sample.

5.1 Forward Solver

The newly written Finite Element Solver (described in Section 3.2) implements a solution of Eq. 8 under the boundary condition shown in Eq. 9 (see code in Appendix). It is the first FEM model based on the complete electrode model that was written in Python. The four parts of the admittance matrix were first calculated separately and then assembled (see Eq. 22). The code in this implementation of the CEM was fully vectorised and was adapted to run on the graphics card (see Appendix). The GPU boost was handled by the `CuPy` package [17]. All parts of the code except for the integrating matrices are trivially extensible to 3D. This algorithm is the upgraded version of the old forward solver written in the Semester 1 project [1]. The new solver considers the width of the electrodes, instead of treating them as ones of infinitesimal size. The effects of the contact impedance layer between the electrodes and the material are also included in the model. Finally, a comparison between the conductivity reconstructions made by the newly written CEM forward solver and by the old forward solver (described in [1]) was performed.

5.2 ESA Implementation and Parameter Optimisation

While there are some time-consuming obstacles in real-life testing, the speed of a simulation only depends on the available computing power. Therefore, the proposed algorithm (see Section 4) was optimised and tested on simulated data with added white uncorrelated Gaussian noise on the voltages. The standard deviation of the noise on each voltage was 3% of its value. The optimisation of ESA's parameters was performed using Bayesian optimisation.

The Bayesian optimisation was done using the `scikit-optimize` package. One call of the function consisted of testing the algorithm efficiency on 1000 samples with random conductivity distributions. The distributions include both 2D Gaussian anomalies of random amplitude, standard deviation and mean, and discontinuous anomalies (of constant conductivity). The discontinuous ones have elliptical/triangular shapes, or represent lines with 0.01% of the reference conductivity (simulating cracks in the sample). Such a variety of the samples on which the algorithm was optimised aimed to train it universally. 150 measurements were performed on each sample and the Bayesian optimisation algorithm was allowed a maximum of 100 calls of the objective function (each reconstructing 1000 samples).

The algorithm calculated the L2 loss of the reconstruction every 10 voltage measurements. L2 loss is defined by

$$L = \sum_{p=1}^{64} \sum_{q=1}^{64} (\sigma_{pq} - \sigma_{true,pq})^2,$$

where the sum is over all pixels in the images, $p, q \in [1, 64]$, and σ_{true} is the true conductivity image. Therefore, an array of losses was produced for each sample. The point-wise gradient of the L2 loss was obtained and divided by the number of measurements conducted:

$$f_i^{(k)} = \frac{1}{n_i^{(k)}} \frac{L_{i+1}^{(k)} - L_i^{(k)}}{n_{i+1}^{(k)} - n_i^{(k)}}, \tag{36}$$

where $f_i^{(k)}$ is the i-th element of the objective function vector of the k-th sample, $n_i^{(k)}$ is the number of conducted measurements until loss $L_i^{(k)}$ is recorded. The parameter i runs from 1 to $\frac{m}{10}$, the number of measurements divided by 10 and k - from 1 to 1000. Then the elements of the objective function vector are summed to get the objective function for the k-th sample. Finally, the mean of the objective functions for all samples is found and minimised using Bayesian Optimisation:

$$\mathcal{F} = \frac{1}{1000} \sum_{k=1}^{1000} \sum_{i=1}^{m/10} f_i^{(k)}. \tag{37}$$

Lastly, ESA was compared against a conventional electrode technique - performing all measurements with opposite current source and sink and adjacent voltage measuring electrodes. Both electrode choice hierarchies were allowed a total of 150 measurements and were tested on 1500 samples identical to the ones used in the parameter optimisation.

5.3 Sensitivity Tests

The sensitivity of the reconstruction algorithm was tested by simulating Gaussian distributed conductivity distributions. All tested samples involve only one Gaussian anomaly each. Gaussians of twenty different amplitudes were tested, ranging from 0 to two times the reference conductivity with interval of $\frac{1}{10}$ of the reference. Also, ten standard deviations were tested for each amplitude. They ranged from $\frac{1}{20}$ to half the side length of the square samples. A hundred different locations for the mean of the Gaussian were sampled. This was done for each possible amplitude and standard deviation, which resulted in a total of 20,000 test samples. This aimed to provide a measure of the sensitivity without bias from the location of the continuous anomaly.

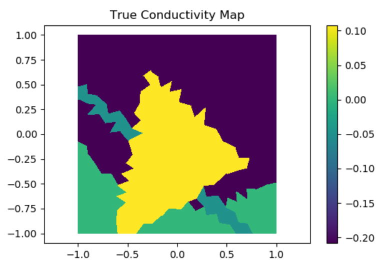
The results of the sensitivity tests were evaluated using two criteria. First, the L2 losses of the reconstructions were compared. Second, the location of the mean was predicted and the Euclidean distance from the true value was calculated. The location of the mean was estimated from the conductivity reconstruction by a weighted mean. It relied on the fact that the conductivity was maximum at the mean of the Gaussian. Thus, the mean prediction accuracy and the loss were plotted against amplitude for a range of standard deviations.

6 Experimental Results

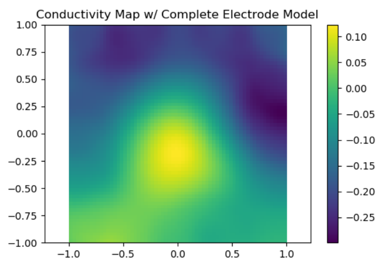
The results in this project can be summarised by three points: comparison between the newly made forward solver and the old one [1], parameter optimisation and evaluation of the Electrode Selection Algorithm and the sensitivity test that aims to provide insight on possible practical uses of EIT.

6.1 Forward Solver Test

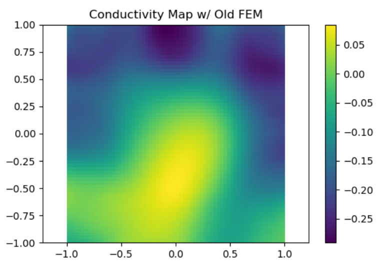
The forward solver was tested against its predecessor, which is described in [1]. The results of the test show that the mean L2 loss of the two algorithms, averaged over 10,000 samples, agrees to within $3 \pm 1\%$. The L2 loss of the reconstructions with the old FEM was 284 ± 1, while the one with the new one was 276 ± 1. Furthermore, the mean execution time of the new FEM algorithm is less than 2 times longer than the old, 0.12 ± 0.01 s for the new algorithm against 0.07 ± 0.01 s for the old one.



(a) True conductivity map.

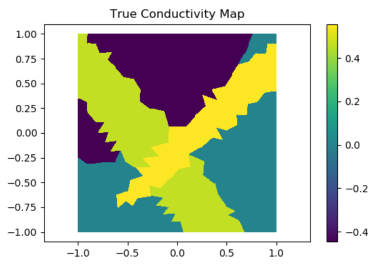


(b) Conductivity reconstruction using new forward solver.

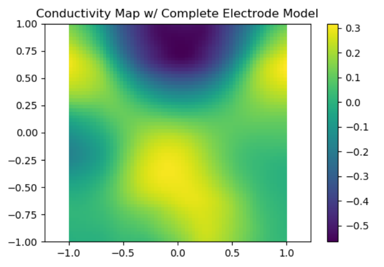


(c) Conductivity reconstruction using old forward solver.

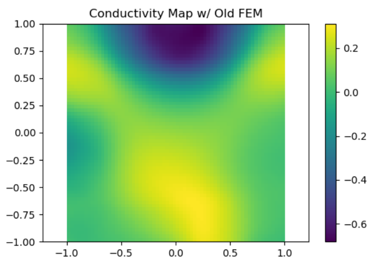
Figure 2: Reconstruction comparison of three anomalies with 10% to 20% deviation from background. NOTE: The zero conductivity value is at -1, whereas the reference conductivity value is at 0.



(a) True conductivity map.

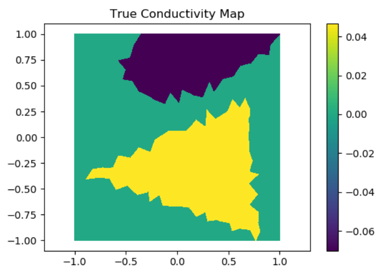


(b) Conductivity reconstruction using new forward solver.

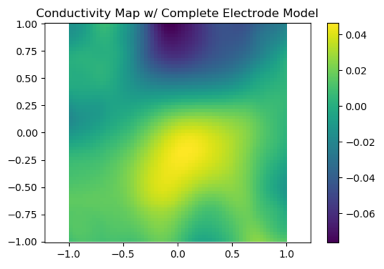


(c) Conductivity reconstruction using old forward solver.

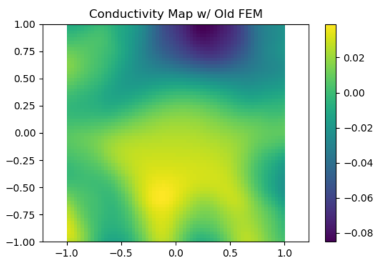
Figure 3: Reconstruction comparison of anomalies crossing each other. NOTE: The zero conductivity value is at -1, whereas the reference conductivity value is at 0.



(a) True conductivity map.



(b) Conductivity reconstruction using new forward solver.

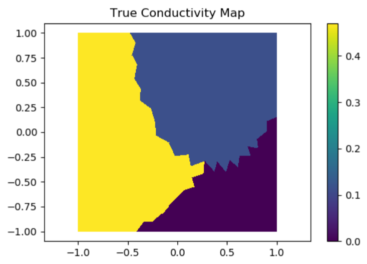


(c) Conductivity reconstruction using old forward solver.

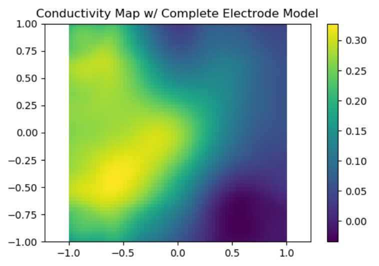
Figure 4: Reconstruction comparison of anomalies that are of comparable deviation as the noise - 4% to 6%. NOTE: The zero conductivity value is at -1, whereas the reference conductivity value is at 0.

The reconstructions of four samples, made with the two algorithms can be seen in Figs. 2 - 5. Note that the conductivity values on the maps are all relative. The zero conductivity value is at -1, whereas the reference conductivity value is at 0.

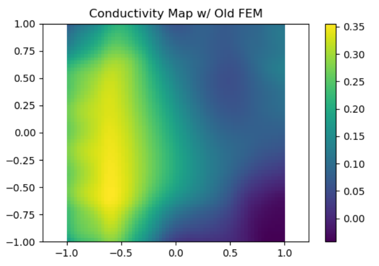
In Fig. 2 the triangular anomaly deviated by only 10% from the reference conductivity. However, strong contrast can be seen with the large elliptical anomaly at the back. It is visible from the two reconstructions that both algorithms detected the triangular and elliptical shape, but failed to detect the third linear anomaly that has 5% smaller conductivity than the background.



(a) True conductivity map.



(b) Conductivity reconstruction using new forward solver.



(c) Conductivity reconstruction using old forward solver.

Figure 5: Reconstruction comparison of anomalies that are screening the central part from the measurements. NOTE: The zero conductivity value is at -1, whereas the reference conductivity value is at 0.

The reconstructions of the crossing anomalies can be seen in Fig. 3. The reconstruction of the anomalies by the new solver differs from the true value by about 10% of the reference. However, the conductivity map reconstructed using the old solver disagrees by up to 20% in some areas.

Two anomalies that have a relatively small conductivity deviation ($\approx 4 - 6\%$ of reference value) from the reference are shown in Fig. 4. The two anomalies are noticeable in the conductivity maps recreated by both algorithms.

In the reconstruction maps, shown in Fig. 5, the middle of the sample is locked between areas of lower conductivity. The yellow area on the left of the image has 40% greater conductivity that the one in the bottom right. While both algorithms distinguished the left and right part of the sample, it can be seen that the locked area in the middle of the reconstruction by the old FEM disagrees with the true value by around 25% of the reference conductivity.

6.2 ESA Parameter Optimisation

The Bayesian Optimisation was run on the objective function (see Section 5.2) to optimise the three parameters from Section 4 - p, q and r. The bounds set on the perturbation p (see Eq. 34) were between 0 and 1. The bounds on the element-wise power of the log-reconstruction q were between -1 to 10 and the bounds on the power of the influence matrix r were between -10 and 1. The parameter values that the optimisation found were

$$p = 0.5;$$
$$q = 10;$$
$$r = -10.$$

Thus, these values were used for the subsequent test.

6.3 ESA Test Results

Two short tests on the quality of the maps reconstructed with the new selection algorithm were performed. Results of the tests can be seen on Figs. 6 - 9.

The first two, Figs. 6 and 7, show the comparison of the loss evolution between the newly written ESA and a common measuring technique - measuring with opposite source and sink and adjacent voltage measuring electrodes [18]. It is visible from the plots that the losses of the ESA reconstructions are dropping sharply at 10 to 30 measurements, until they settle at 60-80

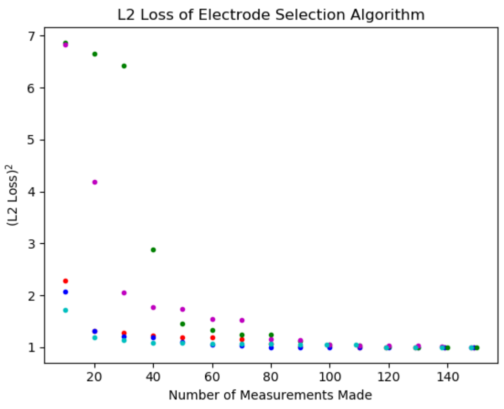


Figure 6: This plot shows the evolution of L2 loss of the reconstruction with the number of measurements for 5 different samples. The measurements were performed by using the electrode pairs recommended by the newly developed ESA (see Section 4). All loss values were standardised by dividing them by the lowest value of the squared loss reached by the measurements with the ESA. The lowest reached squared loss value for each sample is: 1358 for red, 1687 for blue, 9.943 for green, 21.05 for magenta and 4616 for cyan.

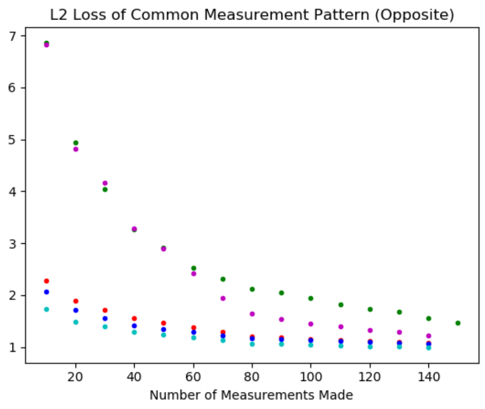


Figure 7: This plot shows the evolution of L2 loss of the reconstruction with the number of measurements for 5 different samples. The measurements were performed by using a standard measuring protocol: performing all measurements with opposite source and sink and adjacent voltage electrodes [18]. All losses were standardised by dividing them by the lowest value of the squared loss reached by the measurements with the ESA.

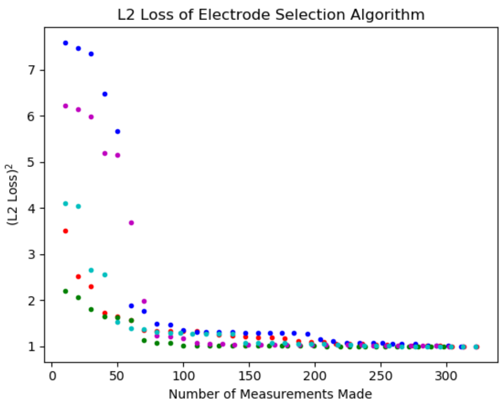


Figure 8: This plot shows the evolution of L2 loss of the reconstruction with the number of measurements for 5 different samples. The measurements were performed by using the electrode pairs recommended by the newly developed ESA (see Section 4). All loss values were standardised by dividing them by the lowest value of the squared loss reached by the measurements with the ESA. The lowest reached squared loss value for each sample is: 0.020 for red, 89.45 for blue, 4566 for green, 74.92 for magenta and 476.24 for cyan.

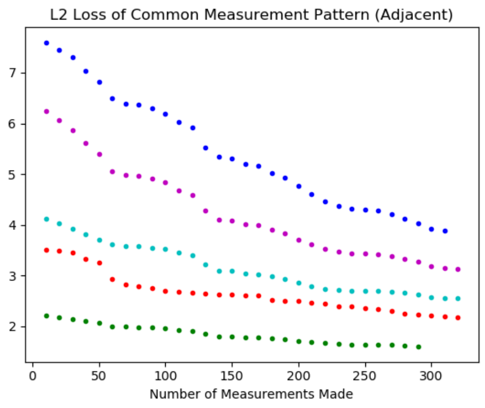


Figure 9: This plot shows the evolution of L2 loss of the reconstruction with the number of measurements for 5 different samples. The measurements were performed by using a standard measuring protocol: performing all measurements with adjacent source and sink and adjacent voltage electrodes [18]. All losses were standardised by dividing them by the lowest value of the squared loss reached by the measurements with the ESA.

measurements (Fig. 6). In contrast, the loss of the opposite pair algorithm (Fig. 7) appears to be in exponential decay. From the two graphs, it can be noticed that the final values of the two algorithms are comparable. However, it can be noticed that the arrival at the minimum occurs with less measurements with the ESA than with the opposite pair technique.

The second pair of figures, Figs. 8 and 9, show the reconstructions of our algorithm compared with the ones made following a measuring protocol with adjacent current driving electrodes and adjacent measuring electrodes. It is evident from the two plots that the ESA, shown on Fig. 8, converges more quickly than the common protocol (Fig. 9). Furthermore, the final loss of the ESA reconstructions of 4 out of the 5 samples (all but the green sample) was at least two times smaller than the respective loss for the common protocol. The steepest descent for the loss of the ESA reconstructions was at around 40 - 80 measurements, while the descent of the common adjacent pattern was relatively steady.

Number of Measurements	10	20	30	40	50	60
Electrode Selection Algorithm	4.418	2.813	2.126	1.654	1.451	1.321
Opposite Electrode Protocol	4.418	3.304	2.862	2.361	2.117	1.874

Number of Measurements	70	80	90	100	110	120
Electrode Selection Algorithm	1.224	1.156	1.104	1.063	1.027	1.000
Opposite Electrode Protocol	1.610	1.378	1.310	1.253	1.205	1.168

Table 1: This two tables show the evolution of the mean squared L2 loss at different number of measurements for two different measurement selection techniques. All losses are normalised with respect to the smallest loss reached by the ESA and averaged over 1500 tested samples with different conductivity distributions.

Furthermore, a more thorough comparison between the losses of two measuring techniques was done - our electrode selection algorithm and the opposite driving electrode protocol [18]. 1500 samples were reconstructed using both algorithms. Meanwhile, the losses were recorded, then normalised with respect to the smallest loss reached by the Electrode Selection Algorithm and averaged over 1500 samples. The results are shown in Table 1. It can be noticed that the normalised loss of the ESA decreases significantly during the first 60 measurements. However, the standard protocol steadily compensates for the initial progress made by the ESA and at the 120-th measurement the two selection algorithms reach comparable normalised loss values. The squared loss value of the opposite pair technique is $17 \pm 1\%$ larger than that of the ESA.

6.4 Sensitivity Analysis

The sensitivity test included reconstructions of 20,000 simulated samples with one 2D Gaussian anomaly each. The Gaussian anomalies had different amplitudes and standard deviations. The mean errors in the mean estimation and the squared losses, averaged over 100 sample reconstructions, are shown in Figs. 10 - 15.

Figs. 10 shows the mean error in the mean position estimation of Gaussians of different amplitudes and a standard deviation of 0.2. As a comparison, the side length of the square samples is equal to 2 in the simulation. The errors in the positions range from about 0.13 to 0.22. The error is largest at amplitudes closer to zero. The lowest values are at -0.5 and 0.4, and the error grows as the Gaussian amplitudes get closer to -1, which is equivalent to zero conductivity in the simulation, the conductivity of vacuum. The losses of the Gaussian reconstructions are shown in Fig. 11. The highest squared loss value, around 2000, can be found

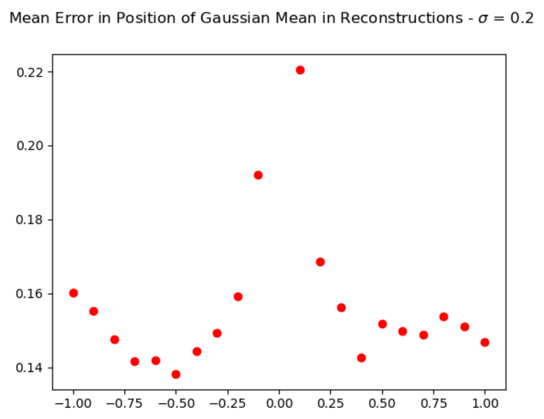


Figure 10: On the above plot the Euclidean distance between the predicted mean and the true mean of the Gaussian anomaly is plotted against the amplitude of the anomaly, used in the simulation. The distance is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The error in the prediction of the mean is averaged over 100 sample reconstructions with different positions of the Gaussian anomaly. The Gaussian anomalies have a standard deviation of 10% of the side length of the sample.

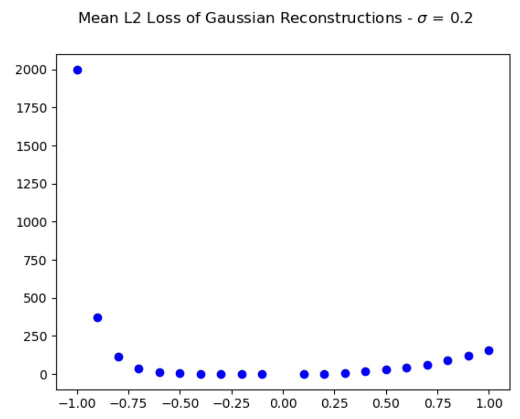


Figure 11: On the above plot the squared L2 loss of the conductivity reconstruction of the Gaussian anomaly is plotted against the amplitude of the Gaussian. The loss is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The loss is averaged over 100 samples with different anomaly position. The Gaussian anomalies have a standard deviation of 10% of the side length of the sample.

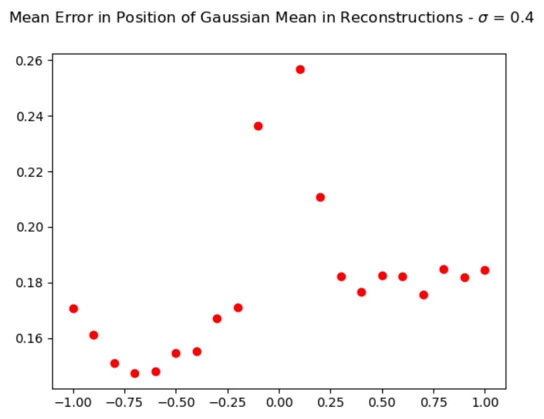


Figure 12: On the above plot the Euclidean distance between the predicted mean and the true mean of the Gaussian anomaly is plotted against the amplitude of the anomaly, used in the simulation. The distance is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The error in the prediction of the mean is averaged over 100 sample reconstructions with different positions of the Gaussian anomaly. The Gaussian anomalies have a standard deviation of 20% of the side length of the sample.

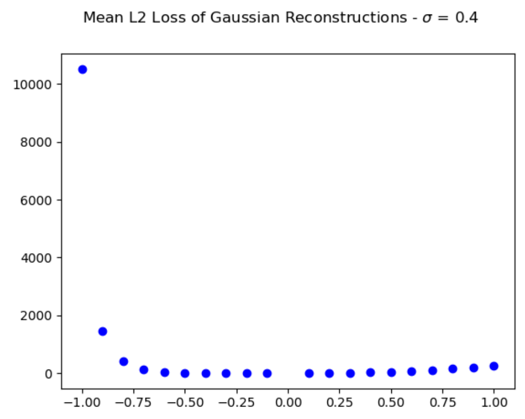


Figure 13: On the above plot the squared L2 loss of the conductivity reconstruction of the Gaussian anomaly is plotted against the amplitude of the Gaussian.The loss is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The loss is averaged over 100 samples with different anomaly position. The Gaussian anomalies have a standard deviation of 20% of the side length of the sample.

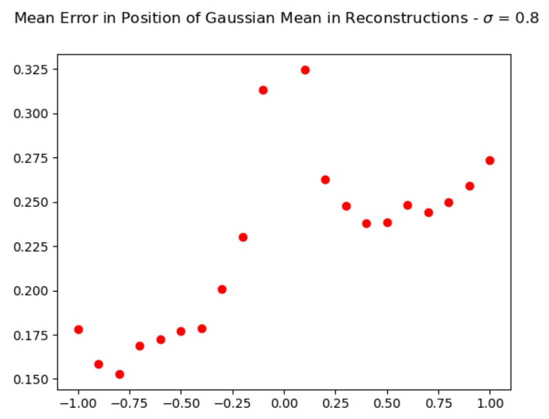


Figure 14: On the above plot the Euclidean distance between the predicted mean and the true mean of the Gaussian anomaly is plotted against the amplitude of the anomaly, used in the simulation. The distance is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The error in the prediction of the mean is averaged over 100 sample reconstructions with different positions of the Gaussian anomaly. The Gaussian anomalies have a standard deviation of 40% of the side length of the sample.

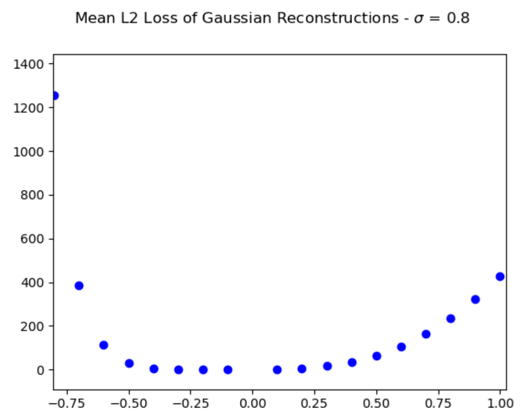


Figure 15: On the above plot the squared L2 loss of the conductivity reconstruction of the Gaussian anomaly is plotted against the amplitude of the Gaussian. The loss is plotted on the y-axis and the amplitude of the corresponding Gaussian - on the x-axis. The loss is averaged over 100 samples with different anomaly position. The Gaussian anomalies have a standard deviation of 40% of the side length of the sample.

at a Gaussian amplitude of -1 (equivalent to zero conductivity). Values are the smallest near an amplitude of 0, the reference amplitude, and grow steadily reaching about 100 at amplitude 1 (2 times the reference conductivity).

The errors in position prediction of Gaussian with a standard deviation of 0.4 are shown in Fig. 12. The errors fluctuate between 0.14 and 0.26. The maximum errors are close to an amplitude of the reference conductivity, while the smallest ones - around amplitude of -0.75, 4 times smaller than the reference conductivity. The losses of the same Gaussians are plotted in Fig. 13. Similarly to the Gaussians of $\sigma = 0.2$, the loss is highest for the Gaussian with the zero conductivity amplitude. However, its value is $\approx 10,500$, more than 5 times over the one of the Gaussians with smaller variance.

The mean errors in position evaluation of the reconstructions of Gaussians with a standard deviation of 0.8 are illustrated in Fig. 14. The range of the errors is between 0.15 and 0.33, which is broader than the ranges of the errors on the Gaussians with lower standard deviations - Figs. 10 and 12. The lowest value is at -0.8, while the highest is at 0.1. It can be also noticed that the error grows from amplitude 0.7 to 1. The squared L2 losses of the Gaussians with a standard deviation of 0.8 are shown between amplitudes of -0.8 and 1 (see Fig. 15). It can be seen that the loss is smallest close to an amplitude of 0, the reference conductivity, and bigger at larger deviations from the reference.

7 Discussion

The results show that the mean loss of the new forward solver over 10,000 samples was $3 \pm 1\%$ lower than that of the old one. One possibility could be that the width of the electrodes in the new model help to obtain more information than the electrodes of infinitesimal width. However, the difference is not statistically significant, due to the limited variety in the test samples. Furthermore, the accuracy of the forward solver with the complete electrode model could be higher than the one of the old solver in real measurements, since factors such as electrode width and contact impedance are taken into account. In any case, further investigation is necessary to assess the accuracy of the new forward solver.

The Bayesian optimisation of the Electrode Selection Algorithm was limited by the bounds set on the optimised parameters. This is hinted by the values of both the q and r parameter, which are at the limit of the corresponding bounds.

The comparison between the novel ESA algorithm and the standard measurement protocols shows that with the characteristic samples described in Section 5.2 the ESA outperforms the two standard techniques, based on L2 loss of the reconstructions. One possible systematic error could be due to the nature of the samples used in the tests. The ESA technique could have dependence on the square shape of the samples. Further tests on samples with different shapes is required in order to establish the observed results.

The sensitivity analysis was conducted by comparing both loss and error in the position estimation. However, there was a significant systematic error due to the way the mean position was inferred from the reconstructions. The weighted mean of the conductivity maps was biased when the Gaussian was not in the middle of the sample, but close to the edges. The reason was that a significant part of the Gaussian was outside the scope of the algorithm and its effects on the weighting were ignored. This might be the reason for the increase in the error in position evaluation at larger deviations of the conductivity (see Fig. 14). However, the loss values were not affected by this systematic error. Overall, the results are not conclusive as to whether and to what extent EIT can be used for non-medical purposes.

8 Conclusion

The newly implemented forward solver shows similar performance to the old one in terms of loss of simulated reconstructions. One way to acquire further information on its efficiency would be to simulate a bigger variety of test samples and compare its reconstructions to the ones of an established algorithm. Another direction of further investigation would be to verify that it reproduces the physical factors surrounding EIT measurements. This could be tested experimentally and the reconstructions could be compared to the ones of verified EIT reconstruction algorithms.

The Bayesian optimisation provided a working set of parameters that surpassed two standard techniques on the square samples described in Section 5.2. The parameters of the algorithm could be further optimised by expanding the range of the Bayesian search or changing the objective function that is minimised. In the evaluation of the ESA algorithm, further investigation on samples with different shapes could be done to increase the statistical significance of the performed analysis. Also, ESA could be compared with other established measurement protocols.

The sensitivity tests did not provide conclusive results due to a systematic error on the deviation of the estimated position. This error could be avoided in future research by fitting 2D normal distributions to the conductivity maps and recording the position of its mean. However, this would require significant time and/or computing power to analyse a statistically significant sample. Also, it is possible to test the sensitivity on samples of different shapes and the range of the Gaussian amplitudes could be changed to study the ability to detect fainter details. Furthermore, a neural network for feature extraction trained with continuous conductivity sample reconstructions could be used to increase the resolving capability of the GREIT algorithm.

References

[1] I. Mihov, "Eit simulation: Conductivity reconstruction and optimisation of electrode selection," *4th Year Report, 1st Sem.*, Jan 2020. [Online]. Available: https://ufile.io/7jewtvpz

[2] B. Brown, "Electrical impedance tomography (eit): a review," *Journal of Medical Engineering & Technology*, vol. 27, no. 3, pp. 97–108, 2003. [Online]. Available: https://doi.org/10.1080/0309190021000059687

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

[4] A. Borsic, R. Halter, Y. Wan, A. Hartov, and K. Paulsen, "Electrical impedance tomography reconstruction for three-dimensional imaging of the prostate," *Physiological measurement*, vol. 31, no. 8, p. S1, 2010.

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

[6] A. Borsic, C. McLeod, W. Lionheart, and N. Kerrouche, "Realistic 2d human thorax modelling for eit," *Physiological Measurement*, vol. 22, no. 1, p. 77, 2001.

[7] B. Galerkin, "On electrical circuits for the approximate solution of the laplace equation," *Vestnik Inzh*, vol. 19, pp. 897–908, 1915.

[8] N. Polydorides and W. R. Lionheart, "A matlab toolkit for three-dimensional electrical impedance tomography: a contribution to the electrical impedance and diffuse optical reconstruction software project," *Measurement science and technology*, vol. 13, no. 12, p. 1871, 2002.

[9] B. Delaunay *et al.*, "Sur la sphere vide," *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, no. 793-800, pp. 1–2, 1934.

[10] 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

[11] 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.

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

[13] M. Soleimani, C. Powell, and N. Polydorides, "Improving the forward solver for the complete electrode model in eit using algebraic multigrid," *IEEE transactions on medical imaging*, vol. 24, pp. 577–83, 06 2005.

[14] C. Gómez-Laberge and A. Adler, "Direct calculation of the electrode movement jacobian for 3d eit," in *13th International Conference on Electrical Bioimpedance and the 8th Conference on Electrical Impedance Tomography.* Springer, 2007, pp. 364–367.

[15] 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.

[16] P. I. Frazier, "A tutorial on bayesian optimization," 2018.

[17] 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

[18] 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, 2015. [Online]. Available: http://doi.org/10.1098/rsta.2015.0329

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

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

A Python Code

The below functions were used as part of the new forward solver. They are the functions that generate the parts of the admittance matrix, described as $\mathbf{A}_{11,\text{ el}}$, $\mathbf{A}_{12}$ and $\mathbf{A}_{22}$ in the text. In the code below np stands for NumPy [19] and cp stands for CuPy [17].

```
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
import cupy as cp
from cupyx.scipy import sparse as sp
from scipy.optimize import minimize
from time import time
def shapeFunctionParameters(self):
    '''
    return arrays of parameters for all shape functions in all triangles on electrodes
        - shape ((n_el * n_per_el - 1), 3, 3)
    '''
    #print(self.tri[twoFromElectrode][isValid])
    pointsTri = self.pts[self.tri[self.twoFromElectrode][self.isValid]] # shape ((n_el
        * n_per_el - 1), 3, 2)
    params = cp.empty((pointsTri.shape[0], 3, 3))
    params[:, :, 0] = cp.multiply(pointsTri[:, [1, 2, 0], 0], pointsTri[:, [2, 0, 1],
        1]) - cp.multiply(pointsTri[:, [2, 0, 1], 0], pointsTri[:, [1, 2, 0], 1])
    params[:, :, 1] = pointsTri[:, [1, 2, 0], 1] - pointsTri[:, [2, 0, 1], 1]
    params[:, :, 2] = - (pointsTri[:, [1, 2, 0], 0] - pointsTri[:, [2, 0, 1], 0])

    return params

def findTrianglesOnElectrodes(self):
    twoFromElectrode = (cp.sum(self.tri < self.ne * self.n_per_el, axis = 1) == 2)
    nodeisElectrode = self.tri[twoFromElectrode][self.tri[twoFromElectrode] < self.ne
        * self.n_per_el].reshape(self.tri[twoFromElectrode].shape[0], 2)
    isValid = ( nodeisElectrode[:, 0]//self.n_per_el - nodeisElectrode[:,
        1]//self.n_per_el ) == 0
    return twoFromElectrode,nodeisElectrode, isValid

def admittanceMatrixC2(self):
    '''
    compute matrix to calculate integral of two shape functions
    over the length of the electrode (assuming they are non-zero) - shape ((n_el *
        n_per_el - 1), 3, 3, 3)
    '''
    shapeParams = self.shapeFunctionParameters()
    whereIsZero = (cp.absolute(shapeParams) - 1e-12 < 0)
    indexZero = cp.where(whereIsZero)
    isConst = indexZero[2] == 2 # 1 for const x, 0 for const y
    zeroShapeFunc = cp.array(indexZero[1])
    indicesShapeFunctions = cp.outer(cp.ones(shapeParams.shape[0]), cp.arange(3))
    indicesShapeFunctions[:, ~zeroShapeFunc] = 0
    outerOfShapeFunc = cp.einsum('ijk, ipq -> ijpkq', shapeParams, shapeParams)
    integratingMatrix = cp.empty((outerOfShapeFunc.shape[0],
        outerOfShapeFunc.shape[3], outerOfShapeFunc.shape[4]))
    sortedElNodeIndices = cp.sort(self.nodeisElectrode[self.isValid], axis=1)
```

```
firstOrderY = cp.empty((outerOfShapeFunc.shape[0]))
secondOrderY = cp.empty((outerOfShapeFunc.shape[0]))
thirdOrderY = cp.empty((outerOfShapeFunc.shape[0]))
constX = cp.ones((outerOfShapeFunc.shape[0], 3))
firstOrderY[:] = self.pts[sortedElNodeIndices, :][:, 1, 1] -
    self.pts[sortedElNodeIndices, :][:, 0, 1] # y2 - y1
secondOrderY[:] = 0.5 * (cp.power(self.pts[sortedElNodeIndices, :][:, 1, 1], 2) -
    cp.power(self.pts[sortedElNodeIndices, :][:, 0, 1], 2)) # 1/2 (y2^2 - y1^2)
thirdOrderY[:] = 1./3. * (cp.power(self.pts[sortedElNodeIndices, :][:, 1, 1], 3) -
    cp.power(self.pts[sortedElNodeIndices, :][:, 0, 1], 3)) # 1/3 (y2^3 - y1^3)
constX[:, 1] = self.pts[sortedElNodeIndices, :][:, 1, 0]
constX = cp.einsum('ij, ik -> ijk', constX, constX)
integratingMatrix[:, 0, 0] = firstOrderY[:]
integratingMatrix[:, 0, 1] = firstOrderY[:]
integratingMatrix[:, 1, 0] = firstOrderY[:]
integratingMatrix[:, 0, 2] = secondOrderY[:]
integratingMatrix[:, 2, 0] = secondOrderY[:]
integratingMatrix[:, 1, 1] = firstOrderY[:]
integratingMatrix[:, 1, 2] = secondOrderY[:]
integratingMatrix[:, 2, 1] = secondOrderY[:]
integratingMatrix[:, 2, 2] = thirdOrderY[:]
integratingMatrix[:] = integratingMatrix * isConst[:, None, None]
firstOrderX = cp.empty((outerOfShapeFunc.shape[0]))
secondOrderX = cp.empty((outerOfShapeFunc.shape[0]))
thirdOrderX = cp.empty((outerOfShapeFunc.shape[0]))
constY = cp.ones((outerOfShapeFunc.shape[0], 3))
firstOrderX[:] = self.pts[sortedElNodeIndices, :][:, 1, 0] -
    self.pts[sortedElNodeIndices, :][:, 0, 0] # x2 - x1
secondOrderX[:] = 0.5 * (cp.power(self.pts[sortedElNodeIndices, :][:, 1, 0], 2) -
    cp.power(self.pts[sortedElNodeIndices, :][:, 0, 0], 2)) # 1/2 (x2^2 - x1^2)
thirdOrderX[:] = 1./3. * (cp.power(self.pts[sortedElNodeIndices, :][:, 1, 0], 3) -
    cp.power(self.pts[sortedElNodeIndices, :][:, 0, 0], 3)) # 1/3 (x2^3 - x1^3)
constY[:, 2] = self.pts[sortedElNodeIndices, :][:, 1, 1]
constY = cp.einsum('ij, ik -> ijk', constY, constY)
indicesConstX = cp.where(isConst)[0]
indicesConstY = cp.where(~isConst)[0]
integratingMatrix[indicesConstY, 0, 0] = firstOrderX[indicesConstY]
integratingMatrix[indicesConstY, 0, 1] = secondOrderX[indicesConstY]
integratingMatrix[indicesConstY, 1, 0] = secondOrderX[indicesConstY]
integratingMatrix[indicesConstY, 0, 2] = firstOrderX[indicesConstY]
integratingMatrix[indicesConstY, 2, 0] = firstOrderX[indicesConstY]
integratingMatrix[indicesConstY, 1, 1] = thirdOrderX[indicesConstY]
integratingMatrix[indicesConstY, 1, 2] = secondOrderX[indicesConstY]
integratingMatrix[indicesConstY, 2, 1] = secondOrderX[indicesConstY]
integratingMatrix[indicesConstY, 2, 2] = firstOrderX[indicesConstY]
integratingMatrix[indicesConstX] = cp.multiply(integratingMatrix[indicesConstX],
    constX[indicesConstX])
integratingMatrix[indicesConstY] = cp.multiply(integratingMatrix[indicesConstY],
    constY[indicesConstY])

admittanceMatrix = cp.einsum('ijklm, ilm -> ijk', outerOfShapeFunc,
    integratingMatrix)

admittanceMatrix[:] = cp.abs(admittanceMatrix)
```

```
        admittanceMatrix[admittanceMatrix < 1e-18] = 0

        return admittanceMatrix

    def admittanceMatrixE(self):
        shapeParams = self.shapeFunctionParameters()
        whereIsZero = (cp.absolute(shapeParams) - 1e-12 < 0)
        indexZero = cp.where(whereIsZero)
        isConst = indexZero[2] == 2 # 1 for const x, 0 for const y
        indicesConstX = cp.where(isConst)[0]
        indicesConstY = cp.where(~isConst)[0]
        sortedElNodeIndices = cp.sort(self.nodeisElectrode[self.isValid], axis=1)
        admittanceMatrixE = cp.zeros((self.n_pts, self.ne))
        shapeMatrix = cp.zeros((shapeParams.shape[0], shapeParams.shape[1], 2))
        integratingMatrix = cp.zeros((shapeParams.shape[0], 2))
        shapeMatrix[indicesConstY, :, 0] = shapeParams[indicesConstY, :, 0] +
            shapeParams[indicesConstY, :, 2] * self.pts[sortedElNodeIndices,
            :][indicesConstY, 1, 1][:, None]
        shapeMatrix[indicesConstY, :, 1] = shapeParams[indicesConstY, :, 1]
        shapeMatrix[indicesConstX, :, 0] = shapeParams[indicesConstX, :, 0] +
            shapeParams[indicesConstX, :, 1] * self.pts[sortedElNodeIndices,
            :][indicesConstX, 1, 0][:, None]
        shapeMatrix[indicesConstX, :, 1] = shapeParams[indicesConstX, :, 2]
        integratingMatrix[indicesConstY, 0] = self.pts[sortedElNodeIndices,
            :][indicesConstY, 1, 0] - self.pts[sortedElNodeIndices, :][indicesConstY, 0, 0]
        integratingMatrix[indicesConstY, 1] = 0.5 *
            (cp.power(self.pts[sortedElNodeIndices, :][indicesConstY, 1, 0], 2) -
            cp.power(self.pts[sortedElNodeIndices, :][indicesConstY, 0, 0], 2))
        integratingMatrix[indicesConstX, 0] = self.pts[sortedElNodeIndices,
            :][indicesConstX, 1, 1] - self.pts[sortedElNodeIndices, :][indicesConstX, 0, 1]
        integratingMatrix[indicesConstX, 1] = 0.5 *
            (cp.power(self.pts[sortedElNodeIndices, :][indicesConstX, 1, 1], 2) -
            cp.power(self.pts[sortedElNodeIndices, :][indicesConstX, 0, 1], 2))
        integrals = cp.einsum('ijk, ik -> ij', shapeMatrix, integratingMatrix)
        integrals[:] = cp.abs(integrals)

        indexElectrode = sortedElNodeIndices[:, 0] // self.n_per_el
        integrals = - integrals / self.z[indexElectrode][:, None, None]
        integrals = integrals.ravel()
        indexElectrode = cp.tile(indexElectrode, (self.n_per_el, 1)).T.ravel()
        indexNode = self.tri[self.twoFromElectrode][self.isValid].ravel()

        indSort = cp.argsort(indexNode)
        indexNode = indexNode[indSort]
        indexElectrode = indexElectrode[indSort]
        integrals = integrals[indSort]

        unique, counts = cp.unique(indexNode, return_counts=True)
        index_pointer = cp.zeros(self.n_pts + 1)
        sum_count = cp.cumsum(counts)
        index_pointer[unique[:]+1] = sum_count[:]
        nonzeroes = cp.nonzero(index_pointer)[0]
        mask = cp.zeros(index_pointer.shape[0], dtype='b1')
        mask[nonzeroes] = True
        mask[0] = True
        zeroes = cp.where(~mask)[0]
        while (index_pointer[1:]==0).any():
            index_pointer[zeroes] = index_pointer[zeroes - 1]

        admittanceMatrixE = sp.csr_matrix((integrals, indexElectrode, index_pointer),
            shape=(self.n_pts, self.ne), dtype=integrals.dtype)
        adm = admittanceMatrixE.toarray();
        return adm;

    def admittanceMatrixD(self):
        all_el_nodes_coords = self.pts[:(self.ne * self.n_per_el)].reshape((self.ne,
            self.n_per_el, 2))
        lengths = cp.linalg.norm((all_el_nodes_coords[:, 0] - all_el_nodes_coords[:,
            (self.n_per_el - 1)]), axis=1)
        admittanceMatrixD = cp.diag(lengths/self.z)
        return admittanceMatrixD
```

The below functions were used to calculate the global stiffness matrix from the local stiffness matrices on each element and then assemble the global admittance matrix **A** by calling the above functions to generate the other parts.

```
def assemble_sparse(self, 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))
    admittanceMatrixC2 = self.admittanceMatrixC2()
    data = cp.multiply(ke[:], perm[:, None, None])
    indexElectrode = cp.sort(self.tri[self.twoFromElectrode][self.isValid], axis=1)[:,
        0] // self.n_per_el
    data[self.twoFromElectrode][self.isValid] =
        (data[self.twoFromElectrode][self.isValid] + ((1/self.z[indexElectrode]))[:,
        None, None] * admittanceMatrixC2)
    data = data.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()

    A = cp.empty((self.n_pts + self.ne, self.n_pts + self.ne), dtype='f8')


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

    A[:self.n_pts, :self.n_pts] = K[:]
    admittanceMatrixE = self.admittanceMatrixE()
    A[self.n_pts:, :self.n_pts] = admittanceMatrixE.T
    A[:self.n_pts, self.n_pts:] = admittanceMatrixE
    A[self.n_pts:, self.n_pts:] = self.admittanceMatrixD()
    return A


def calculate_ke(self):
    '''
    function that calculates the element stiffness matrix on each element

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

def triangle_ke(self, 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]]) # shape (n_tri, 3, 2)
    ke_matrix = cp.empty((len(coord), 3, 3))
    area = cp.abs(0.5 * self.det2x2(s[:, 0], s[:, 1]))
    ke_matrix[:] = cp.einsum('ijk,kli->ijl', s, s.T) / (4. * area[:, None, None])
    return ke_matrix

def det2x2(self, s1, s2):
    '''
    Function that calculates the determinants of two arrays
    '''
    return s1[:, 0]*s2[:, 1] - s1[:, 1]*s2[:, 0]
```

The class below was used to perform the Bayesian optimisation of the parameters of the electrode selection algorithm. The SciKit Optimize (skopt) package was used for the Bayesian optimisation [20]. Elements from PyEIT, connected with the GREIT reconstruction, were used in the optimisation [11].

```
'''
6 April 2020

Python file that optimises the parameters of the custom measurement optimisation
    algorithm

and the most useful voltage measurement electrode pairs.

by Ivo Mihov and Vasil Avramov

in collaboration with Artem Mischenko and Sergey Slizovskiy

from Solid State Physics Group at the University of Manchester


'''

import numpy as np
from controls import *
from scipy.spatial import ConvexHull
from matplotlib.path import Path
from skopt import gp_minimize, dump, load
import matplotlib.pyplot as plt

def L2_Loss(rec, im):
    return np.sum((rec - im) ** 2)

class OptimiseMeasurementAlgorithm(object):
    def __init__(self, n_samples=2e4, ne=20, a=2., npix=64, n_per_el=3):
        init_t = time()
        self.ne = ne
        self.a = a
        self.npix = npix
        self.n_per_el = n_per_el
        self.C_sq = get_centre_squares(npix = self.npix)
        self.all_meas = self.getJacobian()
        self.C_tri =
            np.mean(self.all_meas.mesh_obj['node'][self.all_meas.mesh_obj['element']],
            axis=1)
        self.distance_all = np.linalg.norm(self.C_sq[:, :, None] - self.C_tri[None,
            None, :], axis=3)
        self.weight = 1./(1 + np.exp(1.*(self.distance_all - 0.)))

        self.fwd = self.all_meas.forward_obj
        self.w_mat = self.getWeightSigmoid()
        self.generate_training_set(n_samples)
        _, self.el_coords = meshing.fix_electrodes_multiple(centre=None, edgeX=0.1,
            edgeY=0.1, a=self.a, b=self.a, ppl=self.ne, el_width=0.2,
            num_per_el=self.n_per_el)
        self.pixel_indices, self.voltage_all_possible = self.find_all_distances()
        self.sum_of_pixels_indices = np.sum(self.pixel_indices, axis=(1, 2))
        print('time for init', time() - init_t)

    def getWeightSigmoid(self, ext_ratio=0, gc=False, s=20., ratio=0.1):
        x, y = self.all_meas.mesh_obj['node'][:, 0], self.all_meas.mesh_obj['node'][:, 1]
        x_min, x_max = min(x), max(x)
        y_min, y_max = min(y), max(y)
        x_ext = (x_max - x_min) * ext_ratio
        y_ext = (y_max - y_min) * ext_ratio
        xv, xv_step = np.linspace(x_min-x_ext, x_max+x_ext, num=self.npix,
                          endpoint=False, retstep=True)
        yv, yv_step = np.linspace(y_min-y_ext, y_max+y_ext, num=self.npix,
                          endpoint=False, retstep=True)
        # if need grid correction
        if gc:
            xv = xv + xv_step / 2.0
            yv = yv + yv_step / 2.0
        xg, yg = np.meshgrid(xv, yv, sparse=False, indexing='xy')


        # pts_edges = pts[el_pos]
        cv = ConvexHull(self.all_meas.mesh_obj['node'])
        hull_nodes = cv.vertices
        pts_edges = self.all_meas.mesh_obj['node'][hull_nodes, :]


        # 1. create mask based on meshes
        points = np.vstack((xg.flatten(), yg.flatten())).T
        # 2. extract edge points using el_pos
        path = Path(pts_edges, closed=False)
        mask = ~(path.contains_points(points))
        xy = np.mean(self.all_meas.mesh_obj['node'][self.all_meas.mesh_obj['element']],
            axis=1)
        xyi = np.vstack((xg.flatten(), yg.flatten())).T

        d0 = xy[:, 0].ravel()[:, np.newaxis] - xyi[:, 0].ravel()[np.newaxis, :]
        d1 = xy[:, 1].ravel()[:, np.newaxis] - xyi[:, 1].ravel()[np.newaxis, :]
        distance = np.hypot(d0, d1)
        d_max = np.max(distance)
        distance = 5.0 * distance / d_max
        r0 = 5.0 * ratio
        weight = 1./(1 + np.exp(s*(distance - r0)))
        # normalized
        w_mat = weight / np.sum(weight, axis=0)
        return w_mat

    def getJacobian(self):
        # number electrodes
        el_pos = np.arange(self.ne * self.n_per_el)
        # create an object with the meshing characteristics to initialise a Forward
            object
        mesh_obj = train.mesh(self.ne)

        fwd = Forward(mesh_obj, el_pos, self.ne)

        self.ex_mat_all = train.generateExMat(ne=self.ne)

        f, meas, new_ind = fwd.solve_eit(ex_mat=self.ex_mat_all, perm=fwd.tri_perm)
        #print(f)
        ind = np.arange(len(meas))

        #np.random.shuffle(ind)

        pde_result = train.namedtuple("pde_result", ['jac', 'v', 'b_matrix'])

        ind = np.lexsort((meas[:,3], meas[:, 2], meas[:, 1], meas[:, 0]))

        f = pde_result(jac=f.jac[ind],
                       v=f.v[ind],
                       b_matrix=f.b_matrix[ind])

        all_meas_object = train.namedtuple("all_meas_object", ['f', 'meas', 'new_ind',
            'mesh_obj','forward_obj'])



        all_meas = all_meas_object(f=f,
                                   meas=meas[ind],
                                   new_ind=new_ind[ind],
                                   mesh_obj=mesh_obj,
                                   forward_obj=fwd)
        return all_meas
```

```
def h_mat(self, mask, p=0.2, lamb=0.01):
   jac = self.all_meas.f.jac[mask, :]
   j_j_w = np.dot(jac, jac.T)
   #print('There are nans in jac: ', np.isnan(jac).any())
   r_mat = np.diag(np.power(np.diag(j_j_w), p))
   jac_inv = np.linalg.inv(j_j_w + lamb*r_mat)
   h_mat = np.einsum('jk, lj, lm', self.w_mat, jac, jac_inv, optimize='optimal')
   return h_mat

def get_all_voltages(self, anomaly):
   mesh_new = set_perm(self.all_meas.mesh_obj, anomaly=anomaly, background=None)
   voltage_readings, measurements, new_ind =
       self.fwd.solveAnomaly(ex_mat=self.ex_mat_all, step=1, perm=mesh_new['perm'],
       parser=None)
   ind = np.lexsort((measurements[:,3], measurements[:, 2], measurements[:, 1],
       measurements[:, 0]))
   return voltage_readings[ind], measurements[ind]

def extract_voltages(self, all_measurements, measurements):
   meas_test_src = np.sort(measurements[:, :2], axis=1)
   meas_test_volt = np.sort(measurements[:, 2:], axis=1)
   meas_test = np.concatenate((meas_test_src, meas_test_volt), axis=1)

   index = np.equal(measurements[:, None, :], all_measurements[None, :, :]).all(2)
   index = np.where(index)
   ind = np.unique(index[1])
   mask = np.zeros(len(all_measurements), dtype=int)
   mask[ind] = 1
   mask = mask.astype(bool)

   return mask

def find_all_distances(self):

   if self.el_coords.shape[0] > self.ne:
      self.el_coords = np.mean(self.el_coords.reshape((self.ne,
          int(self.el_coords.shape[0] / self.ne), 2)), axis=1)
   all_meas_pairs = cp.asnumpy(volt_mat(self.ne))
   const_X_index = ( (self.el_coords[all_meas_pairs[:, 0], 0] -
       self.el_coords[all_meas_pairs[:, 1], 0]) == 0 )
   const_Y_arr = np.sort(self.el_coords[all_meas_pairs[const_X_index], 1], axis=1)

   all_slopes = np.empty(all_meas_pairs.shape[0])
   all_const = np.empty(all_meas_pairs.shape[0])
   all_slopes[~const_X_index] = (self.el_coords[all_meas_pairs[~const_X_index, 0],
       1] - self.el_coords[all_meas_pairs[~const_X_index, 1], 1]) / (
              self.el_coords[all_meas_pairs[~const_X_index, 0], 0] -
                  self.el_coords[all_meas_pairs[~const_X_index, 1], 0])
   all_const[~const_X_index] = self.el_coords[all_meas_pairs[~const_X_index, 1], 1]
       - self.el_coords[all_meas_pairs[~const_X_index, 1], 0] *
       all_slopes[~const_X_index]

   p_st = np.empty(shape=(all_meas_pairs.shape[0],2))
```

```
    p_st[:, 0] = np.amin(self.el_coords[all_meas_pairs,0], axis = 1)
    p_st[:, 1] = all_slopes * p_st[:, 0] + all_const

    p_st[const_X_index, 0] = self.el_coords[all_meas_pairs[const_X_index][:, 0], 0]
    p_st[const_X_index, 1] = const_Y_arr[:, 0]

    p_end = np.empty(shape=(all_meas_pairs.shape[0],2))
    p_end[:, 0] = np.amax(self.el_coords[all_meas_pairs,0], axis = 1)
    p_end[:, 1] = all_slopes * p_end[:, 0] + all_const

    p_end[const_X_index, 0] = self.el_coords[all_meas_pairs[const_X_index][:, 0], 0]
    p_end[const_X_index, 1] = const_Y_arr[:, 1]

    indices = (np.abs(np.cross((p_end - p_st)[:, None, None, :],
                      np.transpose(np.array([p_st[:, 0][:, None, None] -
                         self.C_sq[None, :, :, 0],
                              p_st[:, 1][:, None, None] - self.C_sq[None, :, :,
                                 1]]), axes=(1, 2, 3, 0)))
              / np.linalg.norm(p_end - p_st, axis=1)[:, None, None])
              < self.a / (self.npix * np.sqrt(2)))

    return indices, all_meas_pairs

def get_total_map(self, rec, voltages, h_mat, indices_volt, pert=0.05,
     p_influence=-3., rec_power = 1.):
    perturbing_mat = np.ones((len(voltages), len(voltages))) + pert *
       np.identity(len(voltages))
    v_pert = np.dot(np.diag(voltages), perturbing_mat)
    influence_mat = -np.dot(h_mat, (v_pert - voltages[:, None])).reshape(self.npix,
       self.npix, len(v_pert))
    influence_mat = np.abs(influence_mat)
    influence_mat = np.sum(influence_mat, axis=2)


    grad_mat = np.linalg.norm(np.gradient(rec), axis = 0)

    if np.amin(rec) <= -1.:
      rec = - rec / np.amin(rec)

    rec = np.log(rec + 1.0000001)
    rec = np.abs(rec)
    rec[rec < 0.01] = 0.01
    return grad_mat * (influence_mat) ** p_influence * rec ** rec_power, grad_mat,
       rec, influence_mat

def get_next_pair(self, rec, voltages, h_mat, indices_volt, pert=0.05, cutoff=0.8,
     p_influence=1., rec_power = 1.):
    total_map, grad_mat, rec_log, influence_mat = np.abs(self.get_total_map(rec,
       voltages, h_mat, indices_volt, pert=pert, p_influence=p_influence, rec_power
       = rec_power))

    total_maps_along_lines = total_map[None] * self.pixel_indices
    proximity_to_boundary = np.sum(total_maps_along_lines, axis=(1, 2)) /
       self.sum_of_pixels_indices
    proposed_ex_line =
        self.voltage_all_possible[np.argsort(proximity_to_boundary)[::-1]][:10]
    '''
    plt.figure(2)
    im2 = plt.imshow(rec.astype(float), cmap=plt.cm.viridis, origin='lower',
        extent=[-1, 1, -1, 1])
    plt.title("reconstruction")
    plt.colorbar(im2)
    plt.figure(3)
    im3 = plt.imshow(grad_mat.astype(float), cmap=plt.cm.viridis, origin='lower',
        extent=[-1, 1, -1, 1])
    plt.title("Gradient of reconstruction")
    plt.colorbar(im3)
    plt.figure(4)
    im4 = plt.imshow(rec_log.astype(float), cmap=plt.cm.viridis, origin='lower',
        extent=[-1, 1, -1, 1])
    plt.title("Reconstruction logarithmic")
    plt.colorbar(im4)
    plt.figure(5)
    im5 = plt.imshow(influence_mat.astype(float), cmap=plt.cm.viridis,
        origin='lower', extent=[-1, 1, -1, 1])
    plt.title("Influence Map")
    plt.colorbar(im5)
    plt.figure(6)
    im6 = plt.imshow(total_map.astype(float), cmap=plt.cm.viridis, origin='lower',
        extent=[-1, 1, -1, 1])
    plt.title("Total Map")
    plt.colorbar(im6)
    plt.show()
    '''
    return proposed_ex_line, total_map


def get_next_voltage_pairs(self, new_ex_line, total_map, number_of_voltages,
    counter, npix=64, cutoff=0.90):

    #region of interest index
    roi_index = total_map > cutoff * np.amax(total_map)
    weight_on_each_tri = np.sum(self.weight[roi_index], axis=0)
    new_ex_line = np.sort(new_ex_line)
    index_current = np.equal(new_ex_line[None], self.all_meas.meas[:, :2]).all(1)
    jac_of_interest = self.all_meas.f.jac[index_current]
    jac_weighted = jac_of_interest * weight_on_each_tri[None, :]
    voltage_pair_initial_index = np.argsort(np.sum(jac_weighted, axis=1))[::-1]
    voltage_measurement_predictions = self.all_meas.meas[index_current,
        2:][voltage_pair_initial_index]
    return voltage_measurement_predictions[int(counter * number_of_voltages) :
        int((counter + 1) * number_of_voltages)]

def generate_training_set(self, n_samples):
    self.measInfo = np.empty(shape=(n_samples, int((self.ne * (self.ne - 1) *
        (self.ne - 2) * (self.ne - 3)) / 4), 4))
    self.voltageInfo = np.empty(shape=(n_samples, int((self.ne * (self.ne - 1) *
        (self.ne - 2) * (self.ne - 3)) / 4)))
    self.truth_ims = np.empty(shape=(n_samples, self.npix, self.npix))

    for i in range(n_samples):
       anomaly = train.generate_anoms(2., 2.)
       self.truth_ims[i] = train.generate_examplary_output(2., self.npix,
           anomaly=anomaly)
       self.voltageInfo[i], self.measInfo[i] = self.get_all_voltages(anomaly)

def optimise(self, n_meas = 150, meas_step = 10, cutoff = 0.8, influence_mat_power
     = 1., pert = 0.05, rec_power = 1.):
    volt_mat_start = np.empty(shape=(meas_step, 2), dtype=int)
    volt_mat_start[:, 0] = np.arange(meas_step, dtype=int) % self.ne
    volt_mat_start[:, 1] = ( np.arange(meas_step, dtype=int) + 1 ) % self.ne
    ex_mat_start = [0, self.ne // 2 ]

    loss = np.zeros((self.measInfo.shape[0], n_meas // meas_step - 1))
    num_meas = np.zeros((self.measInfo.shape[0], n_meas // meas_step - 1))
    scaled_loss = np.empty((self.measInfo.shape[0], n_meas // meas_step - 2))

    for i in range(self.measInfo.shape[0]):
       meas = np.empty(shape=(meas_step, 4))
       meas[:, :2] = ex_mat_start
       meas[:, 2:] = volt_mat_start
       new_meas = np.empty(meas.shape)
       new_meas[:] = meas[:]
       q = 0
       counter = np.zeros(int(self.ne * (self.ne - 1) / 2))
       mask = np.zeros((int((self.ne * (self.ne - 1) * (self.ne - 2) * (self.ne - 3))
           / 4)))
       while(meas.shape[0]) <= n_meas - meas_step:
           #t_all = time()
           mask_new = self.extract_voltages(self.measInfo[i], new_meas)
           mask = (mask + mask_new).astype(bool)
           indices_volt = np.where(mask)[0]

           h_mat = self.h_mat(mask)
           voltageValues = self.voltageInfo[i, mask]
           delta_v = voltageValues - self.all_meas.f.v[mask]
           rec = - np.einsum('ij, j -> i ', h_mat, delta_v,
               optimize='greedy').reshape((self.npix, self.npix))
           loss[i, q] = L2_Loss(rec, self.truth_ims[i])
           num_meas[i, q] = meas.shape[0]
           '''
           plt.figure(1)
           im1 = plt.imshow(self.truth_ims[i].astype(float), cmap=plt.cm.viridis,
               origin='lower', extent=[-1, 1, -1, 1])
           plt.title("truth")
           plt.colorbar(im1)
           '''
           sugg_ex_line, total_map = self.get_next_pair(rec, voltageValues, h_mat,
               indices_volt, pert=pert, cutoff=cutoff,
               p_influence=influence_mat_power, rec_power = rec_power)
           loc_index_src = self.ex_mat_all[:, None, 0] == sugg_ex_line[None, :, 0]
           loc_index_sink = self.ex_mat_all[:, None, 1] == sugg_ex_line[None, :, 1]
            loc = (loc_index_src * loc_index_sink).astype(bool)
            j = 0
            #t = time()
            while True:
               masked_counter = counter[loc[:, j]]
               if masked_counter < (self.ne - 2) * (self.ne - 3)//(2*meas_step):
                  counter[loc[:, j]] += 1
                  sugg_ex_line = sugg_ex_line[j]
                  #t2 = time()
                  new_volt_pairs = self.get_next_voltage_pairs(sugg_ex_line, total_map,
                      meas_step, masked_counter, npix=self.npix, cutoff=cutoff)
                  #print(time() - t2, 'for voltage recommendation')
                  break
               else:
                  j += 1
            #print('time for loop is ', time() - t)
            new_meas = np.empty((new_volt_pairs.shape[0], 4))
            new_meas[:, :2] = sugg_ex_line
            new_meas[:, 2:] = new_volt_pairs
            meas = np.concatenate((meas, new_meas), axis=0)
            #print('Time iteration ',time()-t_all)
            q += 1
            #get the suggestions and build the meas matrix
      gradient = np.diff(loss, axis=1)/np.diff(num_meas, axis=1)
      scaled_loss[:] = gradient[:]/num_meas[:, :-1]

      return np.mean(np.sum(scaled_loss, 1))

   def objective_function(self, params):
      return self.optimise(n_meas = 150, meas_step = 10, cutoff = params[0],
          influence_mat_power = params[1], pert = params[2], rec_power = params[3])

opt = OptimiseMeasurementAlgorithm(n_samples = int(1000))
#test.optimise()
bounds = [(0.8, 1.), (-10., 2.), (0., 1.), (-2., 10.)]
initial = [0.97, -10., 0.5, 10.]
result = gp_minimize(opt.objective_function, bounds, x0=initial,
              acq_func = "gp_hedge", n_calls=100, n_random_starts=10, verbose=True,
                  noise=1e-3, n_jobs=-1, model_queue_size=3)

print(result)
```

The following class was used to generate 20,000 sample reconstructions with Gaussian anomalies to research the ability of the algorithm to reconstruct continuous distributions. Scikit-Image, PyEIT, H5PY and SciPy were used in the sensitivity test [11, 19, 20].

```
import meshing
import anomalies
from controls import *
import numpy as np
from pyeit.mesh import set_perm
import greit_rec_training_set as train
from skimage.util import random_noise
import pyeit.eit.greit as greit
```

```python
from time import time
from scipy.optimize import curve_fit
import h5py

class SensitivityTest(object):
   def __init__(self, max_stand_dev, max_amplitude, num_meshgrid=10, side_length=2.,
       npix=64, n_el = 20, num_per_el = 3):
      self.max_sd = max_stand_dev
      self.max_amp = max_amplitude
      self.a = side_length
      self.npix = npix
      self.n_el = n_el
      self.num_per_el = num_per_el
      self.centresSq = self.C_sq()
      self.mean_grid = self.meshgrid(num_meshgrid)
      self.mesh_obj = meshing.mesh(n_el=20, num_per_el=3)
      self.truth_ims, self.tri_perm = self.uncorrGaussMap()

   def meshgrid(self, num_meshgrid):
      mean_grid = np.stack(np.mgrid[- self.a/2:self.a/2:self.a/num_meshgrid, -
          self.a/2:self.a/2:self.a/num_meshgrid], axis=-1) +self.a/(2 * num_meshgrid)
      return mean_grid.reshape(num_meshgrid**2, 2)
   def C_sq(self, centre=None):
      if centre is None:
         centre=[0, 0]
      centresSquares = np.empty((self.npix, self.npix, 2), dtype='f4')
       # initialising the j vector to prepare for tiling
      j = np.arange(self.npix)
       # tiling j to itself npix times (makes array shape (npix, npix))
      j = np.tile(j, (self.npix, 1))
       # i and j are transposes of each other
      i = j
      j = j.T
       # assigning values to C_sq
      centresSquares[i, j, :] = np.transpose([self.a / 2 * ((2 * i + 1) / self.npix -
          1) + centre[0], self.a / 2 * ((2 * j + 1) / self.npix - 1) + centre[1]])
      return centresSquares
   def uncorrGaussMap(self):

      # array to store permitivity in different square

      amplitudes = np.linspace(self.max_amp / 10, self.max_amp, 10, endpoint=True)
      amplitudes = np.concatenate((-amplitudes[::-1], amplitudes))
      stand_devs = np.linspace(self.max_sd/10, self.max_sd, 10, endpoint=True)

      pts, tri = self.mesh_obj['node'], self.mesh_obj['element']
       # assumed that background permitivity is 1 (and therefore difference with
          uniform will be 0)
      permSquares = np.zeros((self.npix, self.npix, self.mean_grid.shape[0], 20, 10),
          dtype='f8')
      permTri = np.zeros((tri.shape[0], self.mean_grid.shape[0], 20, 10), dtype='f8')
       # initialises an array to store the coordinates of centres of squares (pixels)
      centresSquares = self.C_sq()
      centresTriangles = np.mean(pts[tri], axis=1)
```

```
        centresSquares = centresSquares.reshape((self.npix * self.npix, 2))
        permSquares[:] = self.multivariateGaussian(centresSquares, amplitudes,
            self.mean_grid, stand_devs).reshape(self.npix, self.npix,
            self.mean_grid.shape[0], 20, 10)
        permTri[:] = self.multivariateGaussian(centresTriangles, amplitudes,
            self.mean_grid, stand_devs)


        '''
         if (np.abs(permSquares) < 5e-2).any():
             a = np.random.randint(low = 4, high = 14) * 0.1
             permSquares += a
             permTri += a
         '''

        '''
        fig, ax = plt.subplots()
        im = ax.tripcolor(pts[:, 0], pts[:, 1], tri, permTri[:, 0, 0, 0],
            shading='flat', cmap=plt.cm.viridis)
        ax.set_title(r'$\Delta$ Conductivities')
        fig.colorbar(im)
        ax.axis('equal')

        fig1, ax1 = plt.subplots(figsize=(6, 4))
        im1 = ax1.imshow(np.real(permSquares[:, :, 0, 0, 0]) , interpolation='none',
            cmap=plt.cm.viridis, origin='lower', extent=[-1,1,-1,1])
        fig1.colorbar(im1)
        ax1.axis('equal')
        plt.show()
        '''
        return permSquares, permTri

    def multivariateGaussian(self, x, amp, mu, sigma):
        amp_const = amp [:, None] * 2 * np.pi * sigma [None]**2
        const_factor = amp_const / (2 * np.pi * sigma[None]**2)
        x_centred = x[:, None] - mu[None]
        numerator = np.exp(-.5 * np.einsum('ijk, kji -> ij', x_centred, x_centred.T)[:,
            :, None] / sigma[None, None])

        gauss = numerator[:, :, None, :] * const_factor[None, None]
        print(gauss.shape)
        return gauss

    def simulate(self, el_dist, step):
        reconstruction_ims = np.empty_like(self.truth_ims)
        loss = np.empty((self.truth_ims.shape[2], self.truth_ims.shape[3],
            self.truth_ims.shape[4]))
        mean_deviation = np.empty((self.truth_ims.shape[2], self.truth_ims.shape[3],
            self.truth_ims.shape[4]))
        predicted_mean = np.empty((self.truth_ims.shape[2], self.truth_ims.shape[3],
            self.truth_ims.shape[4], 2))

        el_pos = np.arange(self.n_el * self.num_per_el)
```

```
forward = train.Forward(self.mesh_obj, el_pos, self.n_el)
ex_mat = self.ex_mat_w_dist(el_dist)
for mean_index in range(self.truth_ims.shape[2]):
   for amp_index in range(self.truth_ims.shape[3]):
      for std_index in range(self.truth_ims.shape[4]):

         f_unp = forward.solve_eit(ex_mat=ex_mat, step=step,
             perm=np.ones(self.mesh_obj['element'].shape[0]))
         f_p = forward.solve_eit(ex_mat=ex_mat, step=step, perm=self.tri_perm[:,
             mean_index, amp_index, std_index] +
             np.ones(self.mesh_obj['element'].shape[0]))
         variance = 0.0009 * np.power(f_p.v, 2)
         volt_noise = train.sk.random_noise(f_p.v, mode='gaussian',
                                  clip=False, mean=0.0, var=variance)
         eit = greit.GREIT(self.mesh_obj, el_pos, f=f_unp, ex_mat=ex_mat,
             step=step, parser='std')
         eit.setup(p=0.1, lamb=0.01, n=self.npix, s=20., ratio=0.1)
         reconstruction_ims[:, :, mean_index, amp_index, std_index] =
             eit.solve(volt_noise, f_unp.v).reshape((self.npix, self.npix))
         '''
         plt.figure(1)
         im2 = plt.imshow(self.truth_ims[:,:,mean_index, amp_index, std_index],
             origin='lower')
         plt.colorbar(im2)


         plt.figure(2)
         im = plt.imshow(reconstruction_ims[:,:,mean_index, amp_index,
             std_index], origin='lower')
         plt.colorbar(im)
         plt.show()

         t1 = time()
         error = 0.03 * np.ones(self.npix**2)

         max_value = np.argmax(reconstruction_ims[:, :, mean_index, amp_index,
             std_index])
         max_coord = self.centresSq[max_value // self.npix, max_value % self.npix]
         initial_params = np.array([max_coord[0], max_coord[1], 1., 1.])
         params_predicted, _ = curve_fit(gauss,
             self.centresSq.reshape(self.npix**2, 2), reconstruction_ims[:, :,
             mean_index, amp_index, std_index].reshape(self.npix**2), p0 =
             initial_params, sigma=error, method='trf')
         predicted_mean[mean_index, amp_index, std_index] =
             np.array([params_predicted[0], params_predicted[1]])
         print(predicted_mean[mean_index, amp_index, std_index])
         print(time()-t1)
         '''
         max_indices = np.argmax(np.abs(reconstruction_ims[:, :, mean_index,
             amp_index, std_index]))

         max_coord = self.centresSq[max_indices // self.npix, max_indices %
             self.npix]
```

```
        pixels_of_interest = np.linalg.norm(max_coord[None, None] -
            self.centresSq[:, :], axis=2) < 0.1 * self.a

        predicted_mean[mean_index, amp_index, std_index] =
            np.sum(self.centresSq[pixels_of_interest] *
            np.abs(reconstruction_ims[pixels_of_interest][:, None, mean_index,
            amp_index, std_index]), axis=0) /
            np.sum(np.abs(reconstruction_ims[pixels_of_interest][:, None,
            mean_index, amp_index, std_index]), axis=0)
        mean_deviation[mean_index, amp_index, std_index] =
            (np.linalg.norm(predicted_mean[mean_index, amp_index, std_index] -
            self.mean_grid[mean_index]))
        #print(mean_deviation[mean_index, amp_index, std_index])
'''
max_indices = np.argmax(np.abs(reconstruction_ims).reshape(self.npix**2,
    reconstruction_ims.shape[2], reconstruction_ims.shape[3],
    reconstruction_ims.shape[4]), axis=0)

max_coord = self.centresSq[max_indices // self.npix, max_indices % self.npix]

pixels_of_interest = np.linalg.norm(max_coord[None, None] - self.centresSq[:, :,
    None, None, None], axis=6) < 0.05 * self.a

self.centresSq[pixels_of_interest]
'''
loss[:] = self.L2_loss(reconstruction_ims)
save_file = h5py.File("./sensitivity data/sensitivity_100520_0.5_new_greit.h5",
    'a')
try:
   save_file.create_dataset('predicted_mean', data=predicted_mean,
       maxshape=(self.truth_ims.shape[2], self.truth_ims.shape[3],
       self.truth_ims.shape[4], 2))
   save_file.create_dataset('mean_deviation', data=mean_deviation,
       maxshape=(self.truth_ims.shape[2], self.truth_ims.shape[3],
       self.truth_ims.shape[4]))
   save_file.create_dataset('loss', data=loss, maxshape=(self.truth_ims.shape[2],
       self.truth_ims.shape[3], self.truth_ims.shape[4]))
   save_file.create_dataset('reconstruction_ims', data=reconstruction_ims,
       maxshape=reconstruction_ims.shape)
   save_file.create_dataset('truth_ims', data=self.truth_ims,
       maxshape=self.truth_ims.shape)
   print('files saved successfully!')
# if not first, it returns an error, so the datasets in the hdf5 file are
    resized and filled
except:
   print('fuckfuckfuck')
   save_file["predicted_mean"].resize((save_file["predicted_mean"].shape[0] + 1),
       axis = 0)
   save_file["mean_deviation"].resize((save_file["mean_deviation"].shape[0] + 1),
       axis = 0)
   save_file["loss"].resize((save_file["loss"].shape[0] + 1), axis = 0)
   save_file['predicted_mean'][mean_index] = predicted_mean[mean_index]
   save_file['mean_deviation'][mean_index] = mean_deviation[mean_index]
   save_file['loss'][mean_index] = loss[mean_index]
```

```
        save_file.close()
        return reconstruction_ims, loss

    def L2_loss(self, recs):
        return np.sum((recs - self.truth_ims)**2, axis=(0,1))

    def ex_mat_w_dist(self, el_dist):
        col_11 = np.arange(self.n_el)
        col_12 = np.arange(self.n_el // 2)
        col_1 = np.concatenate((col_11, col_12))
        col_21 = (np.arange(self.n_el) + 1) % self.n_el
        col_22 = (np.arange(self.n_el//2) + el_dist) % self.n_el
        col_2 = np.concatenate((col_21, col_22))

        ex_mat = np.stack((col_1, col_2), axis=1)
        ex_mat = np.sort(ex_mat, axis=1)

        return ex_mat

def gauss(x, muX, muY, sigma, a):

    x_centred = x - np.array([muX, muY])[None]
    result = a * np.exp(- .5 * np.einsum('ij, ji->i', x_centred, x_centred.T) / sigma)
        / (2 * np.pi * sigma)
    return result



sens = SensitivityTest(1., 0.5, 10)
recs, loss = sens.simulate(10, np.ones(30))
```