Contents

List of Figures

List of Tables

Sammendrag

Det amorfe stoffet silikon dioksid kan inneholde naturlige electron feller. En elektron felle er en vanskelig og tidskrevende defekt å identifisere gjennom dagens ab-initio metoder. En annerledes fremgangsmåte som tar i bruk maskin lærings modeller var utforsket for å drastisk redusere beregnings tid samtidig som å opprettholde presisjon. Den nødvendige dataen var kalkulert fra 324 amorfe silikon dioksid strukturer gjennom flere rutiner skrevet i python. Den utforskende data analysen avslørte at elektroner har en tendens for å fanges ved vide O-Si-O vinkler og lengre bond. Dette gjaldt ikke for alle elektron feller, noe som økte kompleksiteten av maskin læring modellens oppgave. I prossessen av å fange et elektron, blir bondene strukket ut og O-Si-O vinkelen enda breiere rundt silikon atomet som elektronet lokaliseres rundt.

Utvalget av elektron feller gjør opp 0.45% av dataen, noe som resulterer i et ekstremt ubalansert datasett. Flere ulike teknikker var tatt i bruk for å jobbe rundt dette problemet, og to ulike deskriptorer var sammenlignet. Optmiserings prosessen viste seg å være utfordrende på grunn av det lille utvalget med defekter. Resultatene viste tegn til overtilpasning ettersom at modellen presterte mye bedre på treningsdataen i forhold til validerings- og testdataen. I dette prosjektet ble gradient forsterkede beslutningstrær kalt XGBoost og Catboost brukt, ettersom at de utkonkurrerte de andre alternativene under testing. En XGBoost model klarte å klassifisere 87 av 105 defekter med kun 7 falske positive. En annen Catboost model var i stand til å identifisere 104 defekter med et høyt antall falske positive på 1021 tilfeller.

Abstract

The amorphous solid silicon dioxide is suspect to intrinsic electron traps; a difficult and time-consuming defect to identify through today's ab-initio methods. A different approach utilizing machine learning models was explored to drastically reduce computing time while upholding precision. The necessary data was calculated from 324 amorphous silicon dioxide structures through several routines written in python. The exploratory data analysis revealed several points of interest. Electrons tend to trap at wider O-Si-O angles and long bond lengths. However, this is not always the case, which increased the complexity of the machine learning model's task. In the process of electron trapping, the bonds elongate and the O-Si-O angle widens around the silicon atom that the electron localizes by.

The sample size of electron traps makes up 0.45% of the data, which is a heavily imbalanced data set. Several techniques were applied to work around the uneven classes, and two different descriptors were compared. The optimization process proved challenging due to the small sample size of defects. The results showed signs of over-fitting due to the model's superior predictive capability on the training set. For this project, gradient boosted decision trees named XGBoost and Catboost were used, as they outperformed the other alternatives in the initial phase. One XGBoost model was able to correctly classify 87 out of 105 defects with only seven false positives. Another Catboost model was able to correctly classify 104 defects with a significant number of false positives reaching 1021 instances.

1 Introduction

Amorphous materials have a non-crystalline nature, where the only structural requirement is that there is an approximately constant separation of nearest-neighbour atoms [13]. Thus, the atoms can be distributed in such a way that they keep an approximate distance between their nearest neighbour, but without the long-range periodicity that is characteristic of crystals. The short-range order of the amorphous structure ensures distributions of properties such as bond lengths and angles, instead of one or two constant values across the entire structure. These distributions can vary from structure to structure depending on the presence of microstructures and interfaces within the material.

Amorphous silicon dioxide consists of SiO_4 tetrahedra, where the silicon is in the centre and the oxygen atoms are placed in the corners, see Figure 1(a). Amorphous solids, among them silicon dioxide, play a major role in the development of many modern microelectronics [28], [29]. Applications of amorphous solids include a wide range of highly sensitive transducers, sensing devices, and various converters [36], as well as RAM devices and gate insulators in electronic devices [16], [25]. A common application of amorphous silica is in optical fibres, which are used in a wide range of technologies [1], [8].

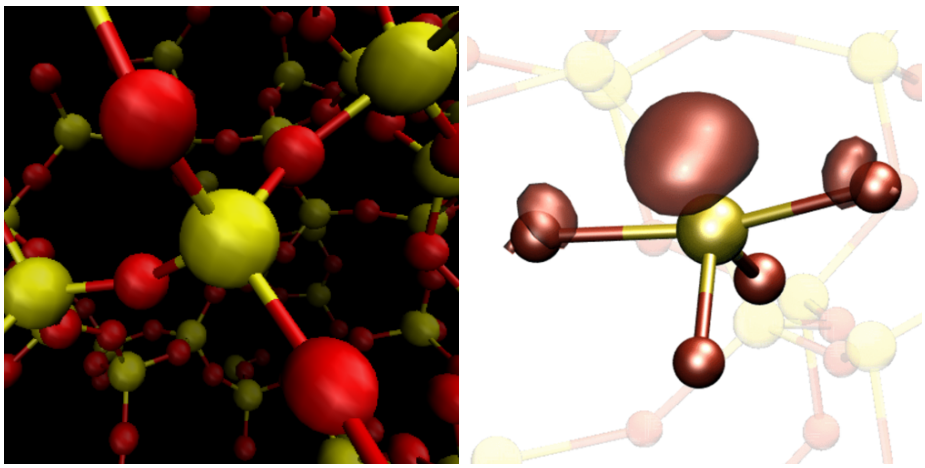

Figure 1: (a) SiO_4 tetrahedron in amorphous silicon dioxide, visualized in vmd [11].(b) An intrinsic electron trap in a-SiO2 with spin distribution [25].

Defects are of interest when utilizing amorphous solids in devices, as they can completely break down due to unwanted defects, while others may depend on the creation and migration of defects [21], [25]. There are different types of defects that can manifest in amorphous silica, among them are the peroxy bridge, O-O bonds, NBOHC (dangling-bond), hydroxyl-E' centres, oxygen vacancies (both relaxed and non-relaxed) and electron traps [27]. The focus of this thesis is the manifestation of spontaneous intrinsic electron traps in amorphous silicon dioxide. An electron can become trapped spontaneously deep in the bandgap by coming too close to a trapping site which intrinsically exists in the structure. This process fills the electronic state, which leads to structural distortion and localization of an additional electron on a Si atom [32]. The structural change can manifest as an opening of the angles centred on the Si atom and elongation of the surrounding bonds. Figure 1(b) is a visualization of an electron trap from [25]. The spin density in Figure 1(b) is the spin density of the trapped electron which appears as lobes on the atoms. These types of defects are difficult to identify without costly calculations since the trapping sites can be seemingly indistinguishable from normal sites.

The variations of the structure make it difficult to accurately fingerprint defects by using simple approximations such as angle or bond length cut-offs. The go-to way of determining whether a site in the solid can trap an electron is by performing costly ab-initio calculations. This project aims to utilize machine learning to predict the presence of defects based on complex fingerprints. Machine learning has become more prominent in physics as it is capable of learning from massive amounts of data, and can compute properties faster with a high level of accuracy compared to conventional

methods. Machine learning requires representative input data which have been constructed to portray the learning cases through properties which are related to the target class. Additionally, most model predictions are highly dependent on the optimization process for good predictive performance. The machine learning models used are the gradient boosted decision trees, XGBoost and Catboost. A descriptor consisting of properties such as bond lengths, angles, dihedrals and ring structures will be compared to DScribe's atom-centred symmetry functions.

2 Theory

2.1 Decision Trees

The machine learning model called decision trees is a very popular model as its process is close to the human mind's way of reasoning and thus easy to understand [14]. The problem in this thesis is a classification problem; Defect site, and Normal site, so there are two classes. The properties describing silicon atoms are the features and they make up the columns in the data set. The decision tree algorithm searches for the feature that has the highest information gain. For a basic decision tree algorithm, the information gain can be expressed as [17],

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v), \tag{1}$$

where the entropy is,

$$\text{Entropy}(S) = -p \log_2(p) - n \log_2(n). \tag{2}$$

Equation (2) is commonly used in information theory to measure uniformity in a collection S. The collection S consists of the classes, Defect sites which is the positive class, and Normal sites which is the negative class. p and n are respectively the proportion of positive and negative occurrences in S. If all instances in the collection are of the same class, then the entropy is 0 and the collection is uniform. If there is an equal distribution of positive and negative classes, then the entropy is 1. The information gain, Equation (1), is defined such that it can measure the effectiveness of a feature in classifying the training data. The information gain is the reduction in entropy caused by the isolation of a feature, or subset S_v.

Figure 2 shows a diagram of an example decision tree that will be able to classify whether one should go outside. The algorithm starts by defining the Root; the feature with the highest information gain, which in this example is the weather. The next level of the tree will consist of the sub-attributes of the Root; the categorical types of weather; cloudy, sunny and rainy. Then the decision tree will check if any of these sub-attributes have a high enough information gain to insert a leaf node; the final prediction. If not, the algorithm seeks among the remaining features to make a new branch with the highest information gain feature. This process recursively repeats itself until all branches end in a leaf node. This is only a simple decision tree, there exist more complicated methods for choosing the nodes. One example is the multivariate decision trees which search for the best linear combination of attributes at the nodes instead of a single attribute [14].

A weakness of the decision tree is that it can be prone to find solutions which are locally optimal, but not necessarily global [17]. The feature with the highest information gain at the Root may not be the globally best Root. To circumvent this, many algorithms make use of randomized sampling when training the tree, and choosing the best root based on final performance. Decision trees are also prone to over-fitting to the training set since they can in theory fit to every single data point if it was allowed. Over-fitting is a common problem in machine learning. The signature symptom of over-fitting is that the model performs excellently on the training data and badly on unseen data, which essentially renders the model useless on all systems that are not the training data. This is where tree-pruning is essential in creating a strong model. Pruning the tree is a way of controlling

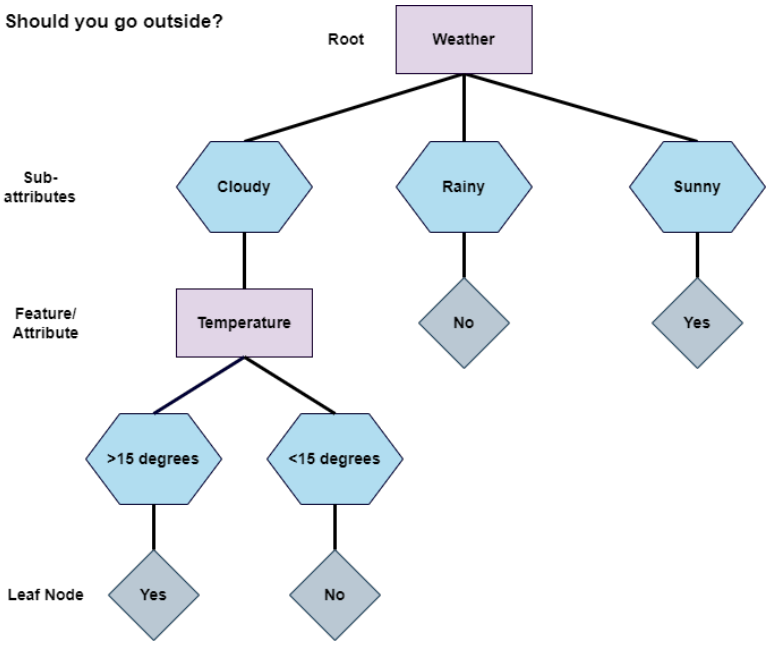

Figure 2: Flow chart of an example decision tree which predicts if it is is a good day to go outside.

the complexity of the tree, a more complex tree will be prone to over-fitting, but on the other hand, it will be robust to noise.

There are several ways of doing this, the specifics depending of course on the choice of algorithm. Limiting the depth of the tree will limit the number of layers with nodes the tree can maximum have, so if the tree has not stopped growing by the time the maximum depth is reached, the tree will stop expanding and insert nodes based on the largest proportion of one of the classes. Similarly one can limit the total number of nodes in the tree. There can also be a limit on the number of data points that can either be split into child-nodes or parent-nodes [14]. Limiting the maximum number of required data points to a high number makes it so that the tree cannot get too specialized, as it will have to stop expanding the tree before the data points are split into uniform groups.

2.1.1 Gradient Boosted Decision trees

Decision trees are a widely used and practical machine learning algorithm. On its own it is simple, but by coupling it together with ensemble learning, one can achieve very strong models [23]. There are three main ways to do this; bagging, boosting and stacking. They are all based on the same concept; many weak learners can outperform one strong learner. The differences between them lie in how the weak learners work together.

Bagging is short for Bootstrap aggregating, and the algorithm employs many weak decision trees in parallel on subsets of the training data and uses a weighted average of the predictions. This method decrease variance by artificially increasing the size of the training data set when making several subsets that may contain some of the same entries. Random Forest is a machine learning algorithm that uses this concept and is very popular for its predictive capability and how simple the parameter tuning is compared to other methods [5]. Stacking is a meta-model approach (learning from learners) where first the algorithm tries to determine the best or most consistent base model among many decision trees. Then, the algorithm will use the outputs of the best base model as inputs to the meta-model, which gives the final outputs [23].

Gradient boosting algorithms are based on decision trees working in sequence, where the first tree is the first iteration. In the first iterations, all entries are weighted equally, In the next iteration, all the wrong predictions will be given a higher weight and the correct classifications will have

a lower weight. The next base-learners should then be constructed with the purpose of being maximally correlated with the gradient of the loss function [18]. The loss function is a generic function describing the error of the model and can be specified by the user.

Two gradient boosting methods are Catboost and XGBoost; two high performing algorithms that are very efficient [34], [7].Catboost builds symmetrical trees such that they will split the same learning instance based on the same condition that was used in the previous decision tree. This results in a tree of depth k with 2^k leaves. Symmetric trees are much faster to build, as the same condition is used in two branches, and it reduces the chance of over-fitting as the algorithm has less control over the choice of features in every branch. Catboost also utilizes ordered boosting. This method allows the algorithm to train on a subset of the data while calculating the residuals of the loss function on a different subset [7]. XGBoost builds asymmetrical trees, and uses unordered boosting [34], [20]. Building asymmetrical trees is time-consuming and increases the complexity of the model, while unordered boosting decreases calculation time. Generally, Catboost is the faster of the two, though either of them can be the better performing one, depending on the data set and the parameter tuning [12], [20].

2.2 Machine learning with imbalanced classes

The machine learning models will be trained on the 324 silicon dioxide structures consisting of 216 atoms. The data will be structured such that each data point will refer to a silicon atom, and the features will describe the local environment around the atom. In total, this results in 23 328 data points. Only 105 of them are defects, which corresponds to 0.45% of the data. This is a highly imbalanced class, which is not optimal for machine learning purposes, as the model can predict that every instance belongs to the majority class with very high accuracy. In this case, the model would have an accuracy of 99.5%, while wrongly classifying all of the defect sites.

There are several ways of dealing with imbalanced classes. Two common methods are under-sampling and over-sampling. If one has a surplus of one of the classes, one can under-sample. This entails not using all of the data points from the majority class during training. This makes the model work with equal instances of both classes, and it will no longer be beneficial to assign all instances to one class. Over-sampling is similar, though instead of dropping data from the larger class, one picks data points from the minority class several times. This artificially increases the minority sample group and has similar consequences as under-sampling. However, over-sampling also carries a larger risk of over-fitting, as the training set will be filled with duplicates. It is also possible to combine these two methods for even better performance [4].

Often one can induce both over-sampling and under-sampling through the use of hyper-parameters. This will force the model to prioritize the minority class. Weights can be scaled so that the model is punished harder for wrongly classifying the defects. Combining this with sample weights and utilizing smaller sub-samples of the data in the training process will also allow the model to more often train on parts of the data with a higher percentage of defect sites.

The data set will be split into three pieces; training, validation and testing. The training set is used to train the model. During optimization, the validation set will be introduced to cross-check for over-fitting. Since the validation set has not been seen by the model, it can act as a neutral reference point for the error of the model. The last part of the data, the testing set, will act as a blind test to better show the generalized performance of the model. This procedure has been shown to improve results for machine learning models [35]. This ensures that the model has not accidentally managed to over-fit both the training and the validation set. Normally the training set is around 60% of the data, validation is 10% and the testing set 30%. If there are many variations within the small sample, the machine learning algorithm will struggle more as there might be cases where some configurations will be unevenly distributed between the training, validation, and testing data set.

```
216
15.031 15.031 15.031
 Si        1.8693464471        6.6674859488        2.1117229310
 Si        3.9199131797        5.7718651154        4.2148588521
 Si        2.1693333986        5.8569794546        6.6692555599
  O        1.7981945980        5.2443815961        1.3280145341
 Si        1.7839249147        0.8708559296        5.2591727626
 Si        5.1752904282        3.9369141767        2.0303308090
  O        4.3737516806        4.5488270046        3.2839993755
  O        3.2891109350        5.2601585556        5.6323015962
  O        4.6142509886        2.4119540425        1.9505532373
 Si        4.1700384810        1.0708977152        1.1772024556
  O        2.7283074916        6.5124581662        3.4449588010
  O        1.1815762163        2.1055612975        4.4654660396
  O        6.7020649594        3.8404447679        2.5118089316
```

Figure 3: The first few lines of one of the xyz-files

2.3 Describing a molecular structure

For the machine learning model to be able to achieve good predictions, it needs a data set that represents the structural properties. The initial data is xyz-formatted files, which contains the xyz-coordinates and name of all atoms in the structure, see Figure 3. Only feeding coordinates to the machine learning model will not work well. The placements of the atoms with regard to an origin is not important, it is how the atoms are distributed compared to other atoms that can convey valuable information about the local environment. This information can be stored under different features, the collection of features is called the descriptor.

The model will rely on the quality of the descriptor when it comes to predictive precision, so creating a good descriptor is vital. Prior knowledge of features that correlate strongly to the target can be used, or more often the features are chosen based on trial-and-error [26]. Through the exploratory data analysis, one should gain some pointers to what properties will perform well as features. Additional pruning or feature manipulation can be done along with the optimization process.

Two approaches to the descriptor have been examined. The first approach is based on calculating simple properties such as bond lengths, angles, dihedral angles and ring structures from the xyz-coordinates. There will be several different angles surrounding the silicon atom, so they will have to be sorted so that there is a consistent sequence of features. If the sequence is not consistent across the system, then the model will struggle. A simple way of sorting the features is by sorting them by type and value. So all angles of the same type; i.e O-Si-O or Si-O-Si, will be together, and each type will be sorted by value from max to min. This method is not perfect, as there could be correlations across features which get lost. Perhaps knowing which angles and bonds go together contains important information, and this will be lost by sorting by value. However keeping track of which features are connected to what is computational expensive, as there will be much more information to structure in a convenient way and it could be difficult to execute without adding a significant number of features.

The second approach will make use of a python library made for describing molecular structures; DScribe [10]. This library offers several descriptors, the one of interest is the Atom-Centered Symmetry Function. In this library, there are three radial and two angular symmetry functions, which can be combined to describe the local environment around the silicon atoms in the material. The symmetry functions are invariant to translation and rotation and continuous with analytical derivatives. The physical interaction between atoms also disappears at large inter-atomic distances. The radial symmetry functions that will be utilized are of the form [2],

$$G_i^1 = \sum_j f_c(R_{ij}), \tag{3}$$

$$G_i^2 = \sum_j e^{-\eta(R_{ij}-R_s)^2} \cdot f_c(R_{ij}), \tag{4}$$

where,

$$f_c(R_{ij}) = \begin{cases} 0.5 \cdot [\cos(\frac{\pi R_{ij}}{R_c}) + 1], & \text{for } R_{ij} \leq R_c \\ 0, & \text{for } R_{ij} > R_c. \end{cases} \tag{5}$$

$f_c(R_{ij})$ is a cut-off function, where R_{ij} is the distance between atom i and atom j. R_c is the cut-off radius. The function essentially removes all interactions from atoms outside the sphere with radius R_c. The first radial function is simply the sum of all cut-off functions. G_i^2 is a sum of Gaussian functions, where η is the width and R_s can be used to shift the centre. By keeping $R_s = 0$, the immediate environment around the atom is in focus. Since it will be important to also examine whether the composition of neighbouring atoms is important, R_s can be increased to the next-nearest neighbour distance around 3.0Å. Then the radial function will calculate the radial distributions along the circumference of a circle centred about the atom of interest.

The angular function is as follows [2],

$$G_i^4 = 2^{1-\zeta} \sum_{j,k \neq i}^{\text{all}} (1 + \lambda \cos\theta_{ijk})^\zeta \cdot e^{-\eta(R_{ij}^2 + R_{ik}^2 + R_{jk}^2)} \cdot f_c(R_{ij}) \cdot f_c(R_{ik}) \cdot f_c(R_{jk}), \tag{6}$$

where $\theta_{ijk} = \arccos(\frac{\vec{R_{ij}} \cdot \vec{R_{ik}}}{R_{ij} \cdot R_{ik}})$. Equation (6) is a summation of cosine functions. The parameter ζ determines the angular resolution, or how many non-zero contributions there are. A high angular resolution results in less contributions overall. This parameter acts similar to η in equation (4). There is also a radial part to this function, so that the angular distribution can be determined from a certain distance from the relevant silicon atom.

Perhaps the biggest advantage of using symmetry function outputs as features is that they are invariant. When using the bond lengths and angles, the simplest way to make the sequence of bonds or angles consistent is to sort them by value. However, this could lead to the machine learning model finding a pattern where there is none. It could appear as if the fourth largest angle is important in the training set, without this being true in general. Symmetry functions eliminate these coincidental patterns, while also containing the same information when using good combinations of the symmetry functions and well-chosen parameters. However, this package was originally built for the machine learning technique neural networks, where having many features does not negatively impact the model. This is not the case for all machine learning algorithms, decision trees included, so the performance of this descriptor relies heavily on the proper choice of parameters.

2.4 Machine learning pipeline

The process of building a machine learning model is referred to as a workflow or pipeline. The first step involves collecting data that the model will learn from. This could be statistical, experimental, or computationally calculated data. Data can be represented in a magnitude of different ways, and not all of them will be purposeful for machine learning. Feature engineering is the process of altering the data to increase performance and effectiveness [15]. Some examples of feature engineering includes removing features, combining features into a new, better feature, or splitting a feature into several properties that describe the system better. Depending on the model used, there will be additional factors to take into account when building the descriptor. Decision trees are sensitive to an abundance of features, especially if any are irrelevant and are not correlated to the target class [14]. Other models cannot handle features that measure the same underlying property, or features with missing values [15].

Through exploratory data analysis (EDA) important characteristics or patterns can be discovered before building the model [31]. There is a wide variety of methods used in EDA, many involve visualizing the features in different combinations or in altered forms. In some cases, the logarithm of a property will reveal a linear correlation to the target class, or perhaps altering longitudinal and latitudinal coordinates to radial coordinates or ZIP codes will reveal important characteristics. The additional knowledge gained from EDA can both be used to confirm how universal the data

is by comparing distributions and plots to other data sets, and it can be used to gain insight into which features and in what combinations they work to improve the predictability of the machine learning model. In the case of many outliers or erroneous data points, it could be necessary to clean the data as this could negatively impact the model predictions.

Once the data has been thoroughly explored, the descriptor will be assembled based on the analysis of the data. The initial descriptor can now be tested on different models. Machine learning models generally require some tuning of the hyperparameters for good results, which is often done by testing a set of different hyperparameters. The set of hyperparameters is chosen based on experience with the model and the initial performance with the descriptor. If the model is showing signs of over-fitting, one should include hyperparameters with values which can reduce over-fitting. The accuracies of the models are then measured and evaluated using a score which fits the problem. The score can simply be a percentage of correctness, some ratio between a combination or sum of false positives, false negatives, true positives and/or true negatives. An additional EDA can be performed on the residuals of the model predictions to gain insight into any potential systematic problems with the model [14]. Potential problems can then be attempted to overcome through feature engineering, and again the models will be tuned, evaluated and finalized.

3 Method

3.1 Modelling amorphous silicon dioxide

In total, 324 molecular structures were produced, each containing 216 atoms. The structures were made using the popular Melt-Quench technique, where the melting and quenching of the crystalline silicon dioxide structure were simulated using molecular dynamics and the classical potential ReaxFF [32]. The structure was then optimized with the ab-initio method, density functional theory (DFT) and a hybrid functional, PBE0_TC_LRC, in the neutral charge state. The target for the machine learning model will be whether the atom is a defect or not. To identify the defects, an electron was added to the system and the structure was optimized again through an additional set of DFT simulations [1]. If the electron localized, i.e there was strong relaxation of the structure, there is an electron trap. When an electron has localized by a silicon atom, the atom's spin moment is drastically increased from about 0.1 to more than 0.8. Using the data from DFT calculations, the silicon atoms with a high spin moment were singled out as the defects.

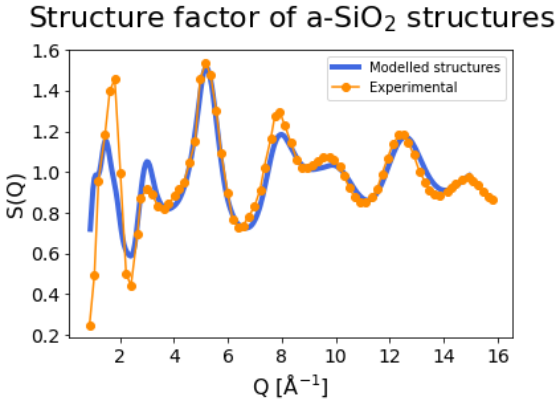

Figure 4: Structure factor of the modelled structures obtained through the melt-quench technique with ReaxFF as the force field in comparison to experimental data from [22]

To verify that the structures are realistic representations of amorphous silicon dioxide, the structure factor of the solid was calculated. The structure factor gives information about the internal structure of amorphous solids and can be found experimentally, or calculated for a given structure

[1] The creation of the structures and the additional DFT analysis to determine the defects were done beforehand by Al-Moatasem El-Sayed

with the atoms' xyz-coordinates [32]. For amorphous silicon dioxide, an isotropic material, the structure factor can be written as [6]

$$S(Q) = \frac{1}{\sum_{i,k} f_i^2} \sum_{i,k} f_i f_k \frac{\sin Q|\vec{R}_i - \vec{R}_k|}{Q|\vec{R}_i - \vec{R}_k|}. \tag{7}$$

For a system of atoms, f_i and f_k are the form factors of two atoms, R_i and R_k are their respective position vectors and Q is the amplitude of the scattering vector. Figure 4 shows the structure factor as a function of Q. The nuclear parameters for the structure factor were retrieved from [22]. The same article produced a plot of the structure factor from inelastic neutron scattering in amorphous silicon dioxide. The experimental data points are included in Figure 4. The signal peak positions match well for all scattering vectors, which indicates good agreement between the amorphous silicon dioxide models and experimental data. The modelled structures match best for large Q values, which is the most important as this is in the reciprocal space, and large Q values correspond to short distances in the solid. When Q is small, the two structure factors do not match as well. This is expected as the sizes of the samples have dimensions around (14-15Å)^3, and the scattering vector will be larger than the sample itself at the smallest Q values.

3.2 Exploratory data analysis

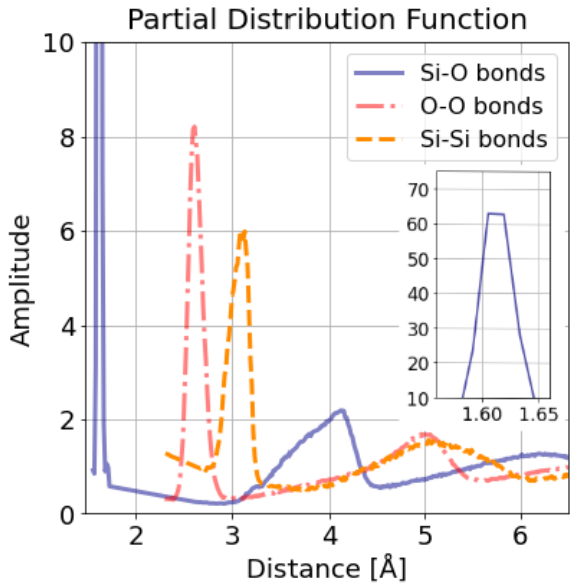

Figure 5: Partial distribution function for O-O, Si-O and Si-Si distances in amorphous oxide.

The structures have been produced in advance, so the next step is to thoroughly examine the data to map all relevant statistics so that the appropriate features can be chosen and potentially modified if necessary. These preliminary calculations will also be used to construct the simple descriptor as it will contain the properties calculated here. The bond lengths, angles, dihedral angles and ring structures were calculated from the xyz-coordinates of each atom. These properties will be examined to reveal their importance for intrinsic electron traps. A python script[2], was written to calculate the partial distribution function for the interatomic distances in the amorphous solid. The function shows the density of distances between certain pairs of atoms, see Figure 5. The distributions were averaged over all data sets to remove noise and improve accuracy. The x-axis is the distance between the atoms. The first peak shows that the largest density of Si and O atoms have a distance of less than 2.0Å between them, these are all the chemical bonds. The first peaks for Si-Si and O-O show that the next nearest neighbours have a distance of less than ca. 3.5Å

[2]See Appendix A for code. Section 3.3 explains the procedures done in greater detail.

and 3Å respectively. The plot compares well with other radial distributions of amorphous silicone dioxide, see [19].

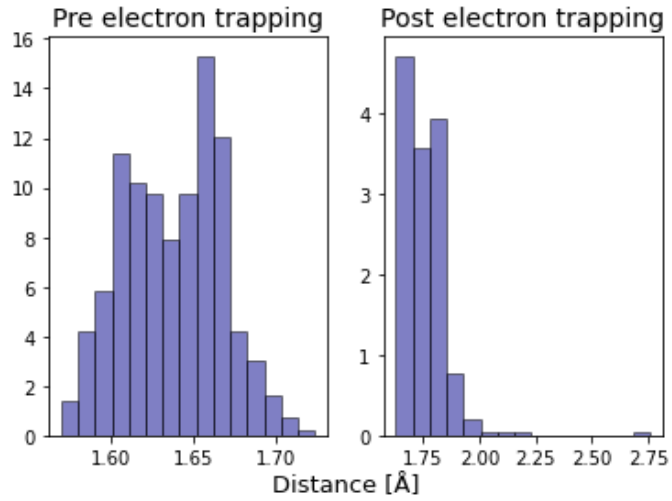

Figure 6: Comparison of bond lengths, before and after electron trapping.

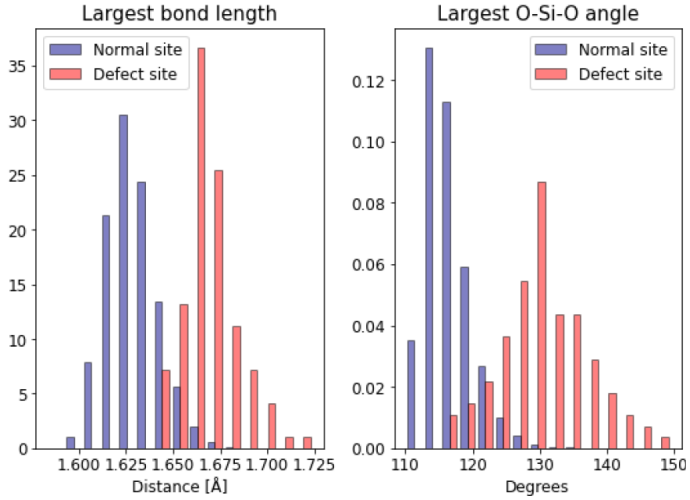

Figure 7: Comparison of normalized angle and bond distributions around normal sites and defect sites.

The chemical bonds were determined by calculating the euclidean distance based on the x, y and z coordinates, and only using the distances that came before the end of the first peak in the partial distribution functions. The cutoffs used were 3.44Å for Si-Si, 3.0Å for O-O and 2.0Å for Si-O. These cutoff limits work well for Si-O bonds, but there were some cases where the cutoff limits included two O atoms or two Si atoms which were not next nearest neighbours. This is due to the irregular structure of the solid. Some adjustments were therefore needed for Si-Si neighbours, so it could be used to identify the ring structures. The bond lengths around defects are shown in Figure 6. It is clear that the bonds elongate significantly in the process of trapping an electron. The first subplot in Figure 7 shows the difference in bond length distributions between normal and defect sites. There, only the longest bond around each silicon atom is shown. It is clear that there is a distinction; defect sites tend to be located by long bonds.

The bond angles were determined using the cosine law,

$$\theta = \arccos \frac{a^2 + b^2 - c^2}{2ab}, \tag{8}$$

where a and b are the lengths of two bonds, c is the distance between the outermost atoms and θ is

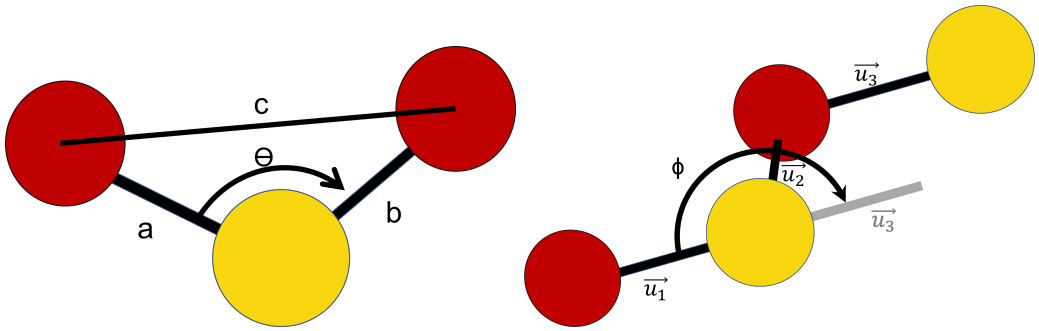

Figure 8: (a) O-Si-O angle diagram. (b) O-Si-O-Si dihedral angle diagram.

the angle between the two bonds, see Figure 8(a). The angular distributions are shown in Figure 9. The distributions have been normalized for comparison purposes. The Si-O-Si angles are generally wider than the O-Si-O angles. Overall, the plots compare well to the angular distributions found in [24]. There are some outliers in the Si-O-Si distributions around 80-90 degrees. These are caused by edge-sharing Si atoms; two silicon atoms sharing two oxygen atoms. In the second subplot of Figure 7, it becomes clear that the differences between normal sites and defect sites are more significant when only looking at the largest O-Si-O angle around a silicon atom.

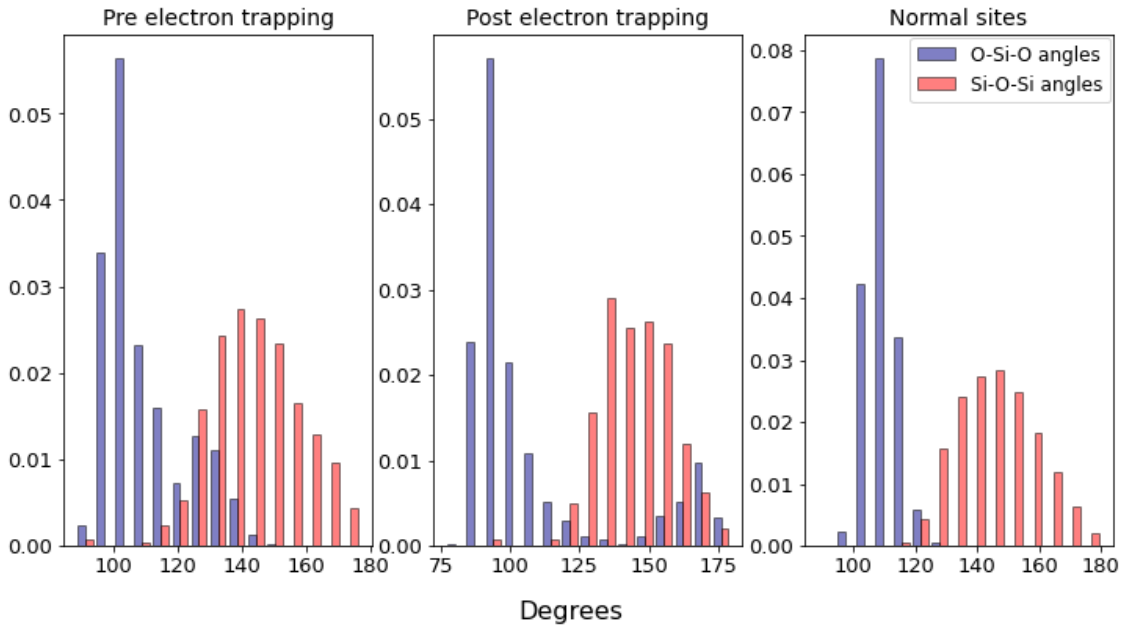

Figure 9: Comparison of the normalized angle distributions, O-Si-O and Si-O-Si, for normal and defect sites.

All angles that are made up of a bond connecting to a defect are considered part of the defect site. This results in six O-Si-O and four Si-O-Si angular configurations per silicon atom. The angles around the defects show different distributions than the ones for normal sites. The plots for the defects sites show two distributions; before the electron has been trapped and after, see Figure 9. The figure shows that the angular O-Si-O distribution for defects generally encompasses a larger range and that one generally finds wider angles around defect sites. Still, it can be observed that some angles narrow after trapping. This could happen when another angle nearby expands because of the electron being trapped and thus forcing an angle to narrow, such that the widening angle can be accommodated. The Si-O-Si distributions have fewer differences between the three scenarios. They all have a similar shape and similar range. These plots do not indicate that Si-O-Si angles have a significant role in electron trapping.

In Figure 10 it is more clear how the O-Si-O distributions differ from each other. The plots show

the percentage of silicon atoms with a certain number of angles larger than a cut-off angle. When this maximum angle is at 120 degrees or more, the defect and normal sites differ drastically. About 94 percent of defects have at least one angle larger than 120 degrees, while only 13.6 percent of the normal sites fulfil the same criteria. However, this also means that 6% of the defects before trapping an electron have all O-Si-O angles smaller than 120 degrees. This raises the question; how do these small-angled defects differ from normal sites?

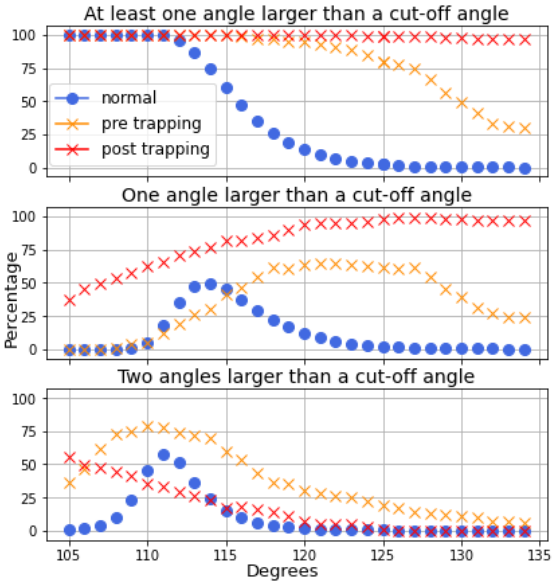

Figure 10: Percentage of silicons with a number of O-Si-O angles larger than a maximum (cut-off) angle

The sites that have trapped an electron, have almost all at least one very large O-Si-O angle. Earlier articles [24], [25], show that wide O-Si-O angles trap electrons. The authors of the second article demonstrate that forcing a narrower O-Si-O angle to widen up to 132 degrees will also trap an electron. Based on the concentration of O-Si-O angles exceeding 132 degrees, they were also able to estimate the concentration of intrinsic electron traps in amorphous silica to be $\approx 4 \times 10^{19}\text{cm}^{-1}$. This number is in excellent agreement with experimental values [32]. The trap density of the structures used for this thesis is $\approx 9.7 \times 10^{-19}\text{cm}^{-1}$.

Another angle of interest is the dihedral angle. This is the angle between the first and last atom in a chain of four atoms, see Figure 8. The dihedral angle was calculated using the bonds between four connecting atoms and representing them as vectors. The equation to find the dihedral angle is [33],

$$\phi = \text{atan2}(|\vec{u_2}|\vec{u_1} \cdot (\vec{u_2} \times \vec{u_3}), (\vec{u_1} \times \vec{u_2}) \cdot (\vec{u_2} \times \vec{u_3}), \tag{9}$$

where the vectors $\vec{u_i}$ are the bond-vector between atom i and $i+1$. All dihedrals where one of the defect silicon atoms helps create the angle are counted as dihedral angles around defect sites. This results in 24 different dihedral angles per silicon atom. These angles have been plotted with the largest dihedral with the relevant silicon atom at the beginning (or end) of the chain along the x-axis, and the largest dihedral with the same silicon atom in the second (or third) place in the chain along the y-axis. Figures 11(a), (b) and (c) show the relation between the dihedral angles where the silicon atom is the first atom in the chain and the second in the dihedral chain. The plot of the normal sites shows a very symmetric relation, where the highest density of angles occurs along a diagonal line, with about 60 degrees spacing in both x- and y-directions. The same relation for the defect sites is significantly different. It is not as symmetric, and the largest density can be found along almost the entire diagonal. This shows that there is a difference, but it could be difficult to quantify it for machine learning purposes.

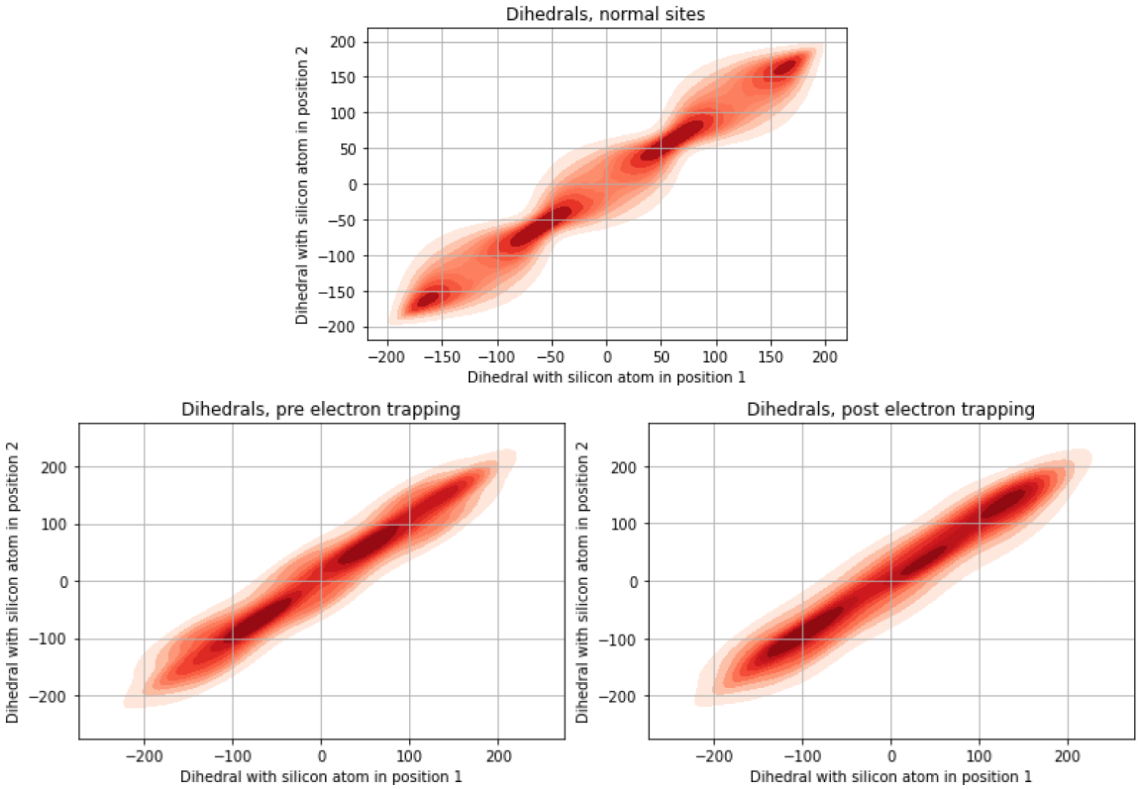

Figure 11: (a) Dihedral angle distribution around normal sites. (b) Dihedral angle distribution around defect sites, before and after electron trapping

The structure contains smaller rings of atoms, and mapping them might provide insight into their relevance to electron trapping. The rings that are identified should be irreducible, i.e. should not consist of smaller rings. The rings were capped at 5 Si atoms. The periodic boundaries and the size of the boxes made it necessary to avoid strings that reached from one end of the box to the other. This would be considered a ring by the computer, while it would be obvious that this is wrong when visually observing the structure. Figure 12 shows the percentages of different sizes of rings around normal sites and defect sites. These plots show that defects have a ring of size 3, about 30% of the time, while normal sites only have such rings about 15% of the time. There are also some differences for the larger rings, but nothing with as high percentages.

3.3 Analyzing structures with python

To represent the individual structures, a class was made. To create a class object, the xyz-file was read and the data was saved in the object. The xyz-files contained the box size, coordinates and names of the atoms. In addition, an index was also given to the atoms based on their row in the file, where the first atom got the index 0. The structures have periodic boundary conditions which were simply implemented by modifying the coordinates,

$$x_i = x_i - L_{x_i} \cdot \text{int}(x_i/L_{x_i}), \tag{10}$$

where $x_i \in [x, y, z]$ and L_{x_i} is the length of the box in direction x_i.

Since python is a scripting language, it does not work well with many for-loops. To avoid this, most of the data was formatted in NumPy matrices [9]. When looking at inter-atomic distances, it was beneficial to structure it in a two-dimensional matrix with the number of columns and rows equal to the number of atoms. Then at index i, j in the matrix, the distance between atom i and atom j is located in vector form. The chemical bonds were picked out as the euclidean distances between

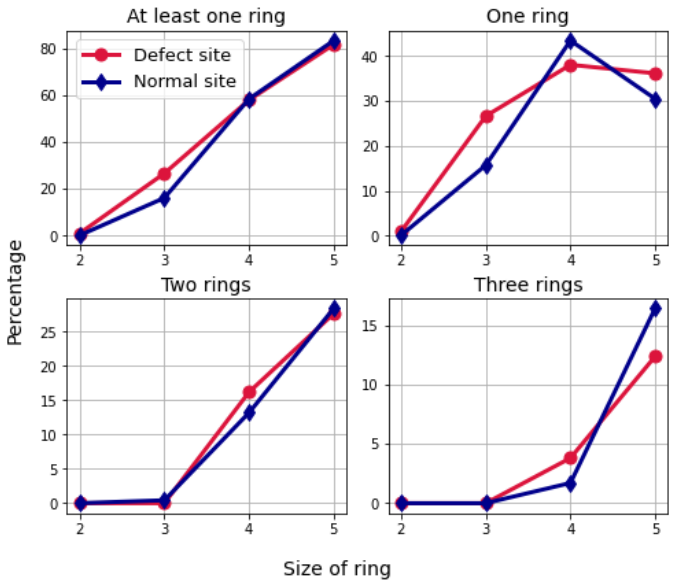

Figure 12: Percentage of rings with different sizes for both defect sites and normal sites

the different pairs of atoms that were less or equal to the cut-off distances 2.0Å, 3Å and 3.5Å. Similarly, also bond matrices were made, to convey what type of bond existed between certain atoms, where the value at index i, j determined what type of bond it was. This was done through logical NumPy statements and Boolean masking to minimize the use of for-loops in the code. To avoid unnecessary computing, these bond matrices were used to determine chains of three atoms, which could be one of four configurations; Si-O-Si, O-Si-O, Si-Si-Si and O-O-O. By looking at an index in the bond matrix, one could instantly pick out all atoms that were bonded (or next-nearest neighbour) based purely on the values in the row, with simple NumPy functions. This information was stored in a three-dimensional matrix, equivalently to the bond matrix, the value at index i, j, k determines the configuration. With the bond indices, the angles could be calculated by using the right indexes from the bond matrices to pick the inter-atomic distances to insert in Equations (8) and (9).

To identify the rings, the Si-Si-Si and Si-Si bond index matrices were utilized to simplify the problem. Then the program would not have to worry about the oxygen indexes in addition to the silicon indexes. The final data set is centred around the silicon atoms, so having information about which oxygen atoms appear in a ring is unimportant. The criteria for a chain to be a ring were that the first and last atom is bonded and that the ring is irreducible; i.e. that it does not consist of any smaller rings. Similarly for the chain of size three, logical NumPy functions and boolean masks were used to determine which silicon atoms were connected. Then the irreducible criteria were checked by confirming that none of the atoms in the chain was bonded across the ring, by checking the value at index i, j in the Si-Si bond matrix. To avoid duplicates of the same ring, they were saved in a set. A set does not allow duplicates and is not ordered. By having a set of frozen sets containing the indexes of the silicon atoms in the ring, no more than one version of the ring would appear in the end.

Originally, there was code to identify rings containing 9 silicon atoms, or of size 9. However, during debugging it was discovered that the boxes were too small, and the code accepted "strings" as rings. Because of the periodic boundaries, strings of silicon atoms which stretched from one end of the box to the other fit the requisites of a ring to the program, but during a visual examination, it was clear it did not qualify as a ring. The only way of working around this was to exclude the rings of size 6 or larger.

3.4 Building and evaluating the descriptors

The simple descriptor consists of the properties calculated during the exploratory data analysis. Each silicon atom will be described through the surrounding bonds, the closest four, and the next-closest four. The closest four bonds will be sorted by value, and the next four bonds will be sorted by which of the closest four bonds the next bond is connected with. The largest bond will be named "Bond 1", while the outer bond connected to Bond 1, which creates a Si-O-Si angle, will be named "Bond 12". There will be four Si-O-Si angles connected to the silicon atom in question sorted by size. Similarly, there will be 6 O-Si-O angles, which are also sorted by size. There will be one feature per ring size, which is 4 features, and the number under the feature will describe the number of rings of that size of the silicon atom is a part of. The last property describing the local environment of the silicon atom is the dihedral angles. In total there are 24 dihedral angles. In twelve of these, the silicon atom is either the first or the last atom in the chain of four atoms. In the other twelve, the silicon atom is the second or third atom in the chain. The two different configurations will be separated, and sorted by value.

The features in this descriptor store a lot of information about the tetrahedron centred around the silicon atom in focus. All of the local angles and bond lengths are there, though the further-reaching features might be slightly lacking. The ring structures can tell something about how well packed the local environment is. If an atom is a part of several smaller rings, then there is little free volume. However, the same could also be for a different atom being a part of larger rings, if the rings are distorted such that there is less free volume. The dihedral angles generally have a very flat distribution, and there is a larger variation in this angle, so it could be difficult to gain consistent information from this property. This type of angle does reach far and covers the neighbouring tetrahedra. The dihedral angle intrinsically contains information about the bond lengths in the neighbouring tetrahedron, but some information will be lost.

To construct the symmetry function descriptor, several different combinations of the symmetry functions will be utilized to gain a wide range of relevant data. The data from the different functions are collected by using several different parameters. For G_2, the parameter R_s was either 0, or neighboring atom distances ranging from 1.65 to 3.5Å. For every R_s, the parameter η had values ranging from 0.40 to 5. The cut-off radius R_c, ranged from 4.0 to 7Å. The angular symmetry function has three variables to determine. The η lies between 0.5 and 2, while ζ is 1 or 2. The λ is 1 and -1 for each combination of the other two variables. The symmetry functions gave in total over a hundred features.

The collection of data will be tested on the models, and the features with the lowest importance will be discarded to remove noise. Because of the extreme imbalance in the data set, one can gain better insight into the feature importance among defects by under-sampling the training data, which results in an even number of defects and normal sites. Through the use of correlation plots, the most influential features were singled out, resulting in a descriptor with 28 features.

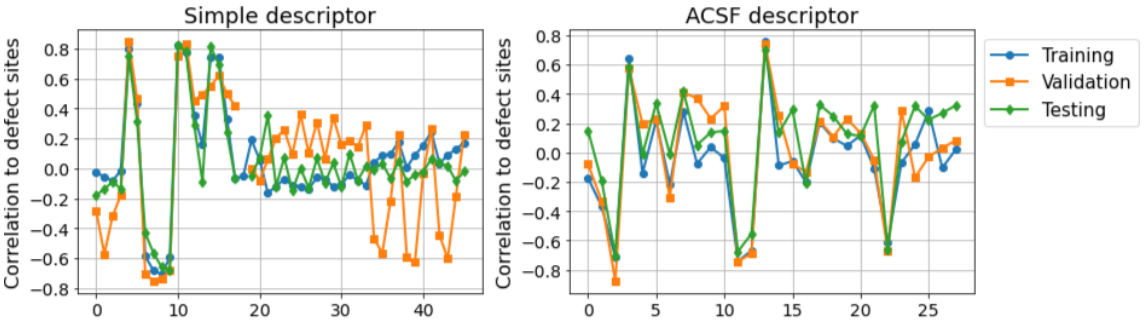

Figure 13: Pearson correlation between the features and whether it is a defect or not. The x-axis is the feature's placement in the descriptor, such that $x = 0$ is the first feature in the descriptor. The two descriptors have a different number of features, so the x-axis differs.

The correlation is calculated using Pearson's standard correlation coefficient through the python library pandas [30]. The Pearson correlation coefficient shows the linear correlation between two

variables. A value close to 1 corresponds to a strong positive linear correlation, but if it is close to -1 it corresponds to a strong negative linear correlation. A coefficient close to 0 means that there is no correlation [3]. Figure 13 show the correlations between the features and whether it is a defect or not for both the simple descriptor and the ACSF descriptor. The plots are differentiating between the different data sets. Generally, they share the same peaks, though it is obvious that the ACSF descriptor looks to have a similar correlation between the data sets. Though, it is worth noting that in some features, in both descriptors, there seems to be a positive correlation in one data set, and a negative in another for the same feature. This can prove to be problematic, as the correlation is not quite consistent across data sets. Essentially, this could result in incorrect classification as the model thinks a criterion is universal when it is not. A larger sample size of positive instances could potentially have evened out the differences between the training, validation and testing set. The noise around the last features in the simple descriptor corresponds to the dihedral angles which have high variance so this is not unexpected. Perhaps the dihedral features will have to be pruned to make the model more stable. However, these correlation plots are not showing for certain what will work badly, the model might find a correlation that is not purely linear. These plots can only give an impression of what may be important to the model, and how these properties are different between the data sets.

4 Results

4.1 Simple descriptor

This descriptor consists of all surrounding O-Si-O and Si-O-Si angles, the angles' corresponding bonds, all dihedral angles and ring structures that the silicon atom is involved in. This results in 46 features per silicon atom. The machine learning models were gradient boosted decision trees as they performed the best without needing optimization, which made it easier to test different combinations of data. Having many features is not always better, as the extra information can create noise, and make the important features have less weight. Having too few features makes the model unstable as it can only base its prediction on very few features, which may exclude some configurations which are supposed to be caught, or aren't supposed to be caught. The selection of features was done by testing different combinations and comparing the performances. From the exploratory data analysis, it is already clear that the O-Si-O angles and bond lengths are important. Only using 3-6 features of the O-Si-O angles and bond lengths sometimes performed better than using all O-Si-O angles and bond lengths.

Optimization	Data set	Correctly predicted defects	False positives	False negatives
None	training	68	0	0
	validation	5	5	3
	test	15	20	14
False positive	training	68	0	0
	validation	5	2	3
	test	14	5	15
False negative	training	68	46	0
	validation	5	11	3
	test	20	27	9
False negative (Under-sampling)	training	68	386	0
	validation	7	61	1
	test	28	187	1

Table 1: Results from XGBoost predictions. Catboost performed slightly worse on all counts.

However, using all features outperformed removing some, so in this section, all features are used. The performance of the models without optimization can be seen in table 1. The false positives are the number of instances the model predicted a normal site to be a defect. The false negatives are the number of times the model predicted a defect site to be a normal site. The models performed excellently on the training set, but could only correctly predict about half of the defects in the

validation and testing set. This is a classical sign of over-fitting. The next row in the table is after optimization with regard to the number of false positives. The false positives were reduced from 25 to only 7, and the correct predictions stayed around the same numbers as before optimization. Attempting to increase the correct predictions, or equivalently, decrease false negatives, have at most resulted in the testing predictions correctly classifying around 5 more defects, but at the cost of three to four times as many false positives as before optimization. The last entry in the table shows the predictive capabilities of a model trained on an under-sampled version of the training set. The majority of normal sites were removed from the training set so that the number of defects and normal sites was even. This model is able to identify nearly all defect sites, except for two. The false positives reach somewhere in the six hundred, which is significant compared to the other models.

4.2 Symmetry function descriptor

This descriptor is based on the output from DScribe's Atom-Centered Symmetry functions. The functions used will be the radial G_2 (4) and angular G_4 (6) symmetry functions. The parameter choice will have a great impact on the results. Both symmetry functions should convey the environment around the tetrahedron, but also the neighbouring tetrahedron. The cutoff radius will be 7.0 Å. It should be as large as possible, but no larger than half the box size. If it were too large, some atoms would be counted twice, which would confuse the results. The parameters for G_2 are η and R_s. To gain information about the local environment around the silicon atom, R_s will be set to 0, and η will have values ranging from 0.4 to 5. To gain information about the neighbouring atoms, with respect to the silicon atom, R_s will be set to the distances 1.65, 2.0, 3.0 and 3.5 Å, which is Si-O, O-O and Si-Si distances. The angular function has three parameters to be determined. λ is always 1 or -1. η will have values ranging from 2 to 0.5 to cover different radial functions. ζ has the value 1 or 2. The parameters were found through testing the descriptors gained by different parameters and examining how well the models performed on them. Analyzing the feature importance also revealed exactly which parameters did badly, and which were the most helpful to the model.

The results of the model are shown in Table 2. The initial predictions before optimization show that the model is naturally finding more defects, but also includes more false positives. However, these results only have one extra false negative and 37 fewer false positives compared to the false-negative optimization in Table 1. This descriptor certainly looks more promising for minimizing the false negatives. The false-negative optimization in Table 2 show rather extreme results. While the model is finding nearly all defects, it is also including a large number of false positives, compared to the other models that do not utilize under-sampling. Generally, the models trained with the ACSF descriptor performed slightly worse than the simple descriptor when the goal was to minimize the false positives, as it predicted around the same defects correctly, but with more false positives.

Optimization	Data set	Correctly predicted defects	False positives	False negatives
None	training	68	4	0
	validation	5	10	3
	test	19	33	10
False negative (under-sampling)	training	68	628	0
	validation	7	97	1
	test	29	296	0

Table 2: Results from Catboost predictions. XGBoost generally performed worse with this descriptor.

5 Discussion

It has been discovered that not all defects follow the rule-of-thumb of wide O-Si-O angles and long bonds. This means that when some defects have similar properties to normal sites, the machine

learning model will struggle to classify the defects correctly, in addition to being suspect to false positives. The majority of defects that were identifiable were in the training set. This is a typical sign of over-fitting, but the results from the optimization could suggest that there is more going on.

The optimization process has been challenging, mainly due to the small number of positive data points. When optimizing, the validation set is supposed to work as a reference point. However, eight data points is a very small sample and there is no guarantee that the changes happening in the validation predictions is representative of a larger sample. Generally, one expects to see accuracies that are similar across the training, validation and testing data set. In most of these predictions, the model outperforms when predicting on the training data with underwhelming results on the two others. During the optimization, these accuracies would be evened out, however, the overall performance worsened, even if it was identifying more defects in the testing set. The only instance where this did not happen, was when training the model on an under-sampled data set. Then, the performance was more balanced compared to the others, but still with a severe number of false positives.

Examination of the wrongly classified defects from the non-optimized model using the simple descriptor showed that about 82% of them had the largest O-Si-O angle smaller than 125 degrees. This corresponds to 15 out of the 17 false negatives. It could indicate that the defects with a smaller largest O-Si-O angle are difficult to differentiate from normal sites for the model. There are of course some of these small-angled defects in the training set as well, around 16%. These are generally predicted correctly, so it could still be an over-fitting problem, or they may coincidentally share some other property that is typical for the wide-angled defects. The optimization with regard to false negatives results in an extreme number of false positives, and only a small reduction in false negatives. It appears as if the only way for the model to find the missing defects is by weakening the criteria so that non-defects fulfil them as well. It could mean that the property that is important among small angled electron traps is not described properly with the given features. This might cause the model to over-fit to the small angled defects in the training set as it struggles with finding a general pattern in the data. Though this was mainly a problem when trying to reduce the number of false positives. When focusing only on correctly predicting as many defects as possible, the models performed well. At most only 1 defect was not found using the ACSF descriptor and the Catboost algorithm.

The two descriptors perform very evenly. The simple descriptor struggles to classify the defects in the testing and validation set, though it is easier to reduce the false positives with this model. The ACSF descriptor catches more defects, but there is a significant number of false positives that is hard to reduce without drastically reducing the correct defect predictions. However, this descriptor still finds more defects with fewer false positives than the simple one, see the false-negative optimization in 1 and the base results in 2. The results might suggest that the ACSF descriptor is describing the local environment around the silicon atoms better, and consequently correctly classifies more defects, while not being able to properly discern defect sites from some normal sites. The ACSF package is built for neural networks, which synergies well with an abundance of features, which the gradient boosted decision trees do not, so reducing the number of symmetry functions used, can also reduce the quality and effectiveness of the descriptor. That being said, using simple cut-off limits perform even worse than the machine learning models. From the EDA it is known that the largest bond length is the distribution that is the most different between defect and normal sites. Using the smallest value among the defects as the cutoff, approx 1.641Å, finds all 105 defects, with 3875 false positives. The model with the most false positives had 1021 false positives, less than a third as many and it only missed one defect. It is clear that the models are undoubtedly outclassing these simple cut-offs.

The EDA was mainly focused on the very local environment around each silicon atom. While this did reveal many interesting properties of the intrinsic electron trap, it is plausible that a more thorough analysis of the neighbouring tetrahedra could have revealed additional, essential information. This information could then have been used to expand the simple descriptor to describe the neighbouring environment, and parameters of the symmetry functions could have been chosen to portray the same in the ACSF-descriptor. Another point to consider is whether the data sets should have been split with more care, as there seem to be a majority of small-

angled defects in the testing and validation set. Potentially adding weight to the defects based on their largest O-Si-O angle could have evened out the differences across the data. This might have increased the predictive capabilities of the model as the other data sets would have been more similar to the training data set.

6 Conclusion

Silicon dioxide is an amorphous solid which is prone to intrinsic electron trapping, which results in an additional electron localizing at a silicon site. The exploratory data analysis showed that the trapping sites tend to have a large (more than 120 degrees) O-Si-O angle, and elongated bonds (more than 1.65Å). The trapping causes distortion of the local environment; further stretching of the bonds, and widening of the O-Si-O angle. The dihedral angle distributions tend to differ between normal and defect sites close to the edges. The normal sites have a visible high concentration towards the maximum/minimum values at ± 180 degrees, which the electron trapping sites are missing. Smaller ring structures, containing three silicon atoms, appear around defects almost thirty percent of the time, while the same is true for circa 15 percent of the normal sites. The gradient boosted decision tree algorithms, Catboost and XGboost, were trained on two different descriptors. The first one consists of simple properties such as bond lengths, angles, dihedral angles and ring structures. The other descriptor consists of output from DScribe's atom-centred symmetry functions, which describe the local environment around each silicon atom through radial and angular symmetry functions. The simple descriptor was prone to false negatives, while the ACSF descriptor was prone to false positives. The machine learning models generally performed very well on the training set compared to the validation and testing set, which is consistent with over-fitting. Analysis of the classification errors of defects revealed that a large portion of them appeared to have a small largest O-Si-O angle. This small-angled configuration also appeared as a much smaller fraction in the training set, which might have affected the performance of the models. The models were optimized to minimize either the false positives or the false negatives. Minimizing the false positives was possible without drastically changing the false negatives when compared to the base results. When optimizing with regard to the false negatives, the number of false positives could quickly increase to the hundreds.

The small sample size turned out to be a larger issue than first assumed, as the defects had a greater variance in their properties than expected from the findings of earlier articles [25], [24]. If something similar is to be attempted, it is strongly recommended to have a much larger sample size of defects to improve results. The neighbouring tetrahedra should also be considered more thoroughly, as the results seem to suggest that a defect's properties are not easily described solely through the closest environment. Another option would be to utilize other properties, *e.g.* electron structure, or similar properties commonly calculated through DFT analysis. However, this raises the issue of calculation time as there is not much point in a machine learning model predicting trapping sites when one still needs to do DFT to gain the electron structure needed for the machine learning model. Potentially, one could try to train a model to predict such properties, which is then fed into another model predicting intrinsic trapping sites. Though, with a larger sample size of defects, additional features may not be necessary. It could be that it is solely the statistical variations of the minority class which is the main culprit in affecting the model predictions.

From an industrial viewpoint, one might want to avoid a certain types of defects as they have the potential to ruin the devices sold to the market[3]. To avoid negative press attention, a company would probably prefer to get rid of all devices that might contain a defect, even though one is losing profit by throwing them away instead of selling them. In such a scenario, one would much rather have many false positives and no, or very few, false negatives. However, too many false positives and the company loses a lot of profit. Then, the model optimized for false negatives in Table 2 would be preferable. If the goal is to have the smallest error, then the false-negative optimization in Table1 would be preferable, as it has the smallest number of errors overall. The calculation cost for the machine learning algorithms is several orders of magnitude smaller than

[3]Intrinsic electron traps would not be one of these, as they are crucial to resistance switching, but the machine learning process explored here can be applied to other defects.

the DFT analysis which produced the data. The models take only a couple of seconds to predict, while it took several weeks to calculate the defect sites with density functional theory. While the models may not be able to both have a low number of false positives and false negatives, a machine learning model could still prove useful.

Consider tuning a model in such a way it catches most defects. Then one would quickly know exactly which silicon atoms have the highest probability of being a defect, allowing one to only perform computationally expensive methods, such as density functional theory, on the relevant structures. This would ease the computational task many times over, with very little extra work. The best model for this task would be the one optimized with regard to false positives in Table 1. This model would pick at most 94 structures as highly likely to contain a defect site, where only seven of these would be defect-free, given that all sites are in individual boxes. This would only require about 30% of the original time spent performing DFT calculations on all molecular structures. Using simpler cut-off limits one could achieve something similar. The best property to use here is the largest O-Si-O angle as it has a smaller overlap with normal sites at large values. Using a cut-off value of 135 degrees singles out 31 structures, only six are defect-free. While the false positives are similar, the cut-off method misses 76% of the defect structures, which means that one would have to produce more than three times as many structures to achieve the same number of defects when using machine learning. Using a smaller value for the cut-off to achieve a similar number of true positives to the machine learning predictions results in over four hundred false positives, which could only exclude a few structures, given that some of the potential defects lie in the same boxes.

The electron traps are difficult to separate from normal sites based on the local environment's structural properties. However, the machine learning models are able to distinguish many, based on the small sample size. The majority of wrongly classified defects appear to have rather narrow O-Si-O angles and thus lie closer to the common angular values of normal sites. Perhaps there is some property that is not properly described by the descriptors, or it is the small sample size of these small-angled configurations, especially in the training set, that hinders the model from learning how to classify these defects. A larger sample size of defects, and a more thorough consideration of the neighbouring tetrahedra, could be what is needed to maximize the predictive performance of the models. However, a machine learning model could with great success be applied to classifying potential defect sites, which could then be further examined to gather important statistics.

Bibliography

[1] J Ballato et al. 'Silicon optical fiber'. In: *Optics express* 16.23 (2008), pp. 18675–18683.

[2] Jörg Behler. 'Atom-centered symmetry functions for constructing high-dimensional neural network potentials'. In: *The Journal of Chemical Physics* (2011), pp. 074106-1–074106-13.

[3] Jacob Benesty et al. 'Pearson correlation coefficient'. In: *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[4] Nitesh V Chawla et al. 'SMOTE: synthetic minority over-sampling technique'. In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[5] Adele Cutler, D Richard Cutler and John R Stevens. 'Random forests'. In: *Ensemble machine learning*. Springer, 2012, pp. 157–175.

[6] S.R Elliot. *Physics of Amorphous Materials*. Longman, 1984.

[7] V. Ershov. Retrieved online from Technical Blog 16/05/2022. 2018. URL: `https://developer.nvidia.com/blog/catboost-fast-gradient-boosting-decision-trees/`.

[8] Sylvain Girard et al. 'Overview of radiation induced point defects in silica-based optical fibers'. In: *Reviews in Physics* 4 (2019), p. 100032.

[9] Charles R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[10] Lauri Himanen et al. 'DScribe: Library of descriptors for machine learning in materials science'. In: *Computer Physics Communications* 247 (2020), p. 106949. ISSN: 0010-4655. DOI: `10.1016/j.cpc.2019.106949`. URL: `https://doi.org/10.1016/j.cpc.2019.106949`.

[11] William Humphrey, Andrew Dalke and Klaus Schulten. 'VMD – Visual Molecular Dynamics'. In: *Journal of Molecular Graphics* 14 (1996), pp. 33–38.

[12] A.B. John. Retrieved online from neptune blog 24/05/2022. 2022. URL: `https://neptune.ai/blog/when-to-choose-catboost-over-xgboost-or-lightgbm`.

[13] Charles Kittel. *Kittel's Introduction to Solid State Physics*. Wiley, 2018.

[14] Sotiris B Kotsiantis. 'Decision trees: a recent overview'. In: *Artificial Intelligence Review* 39.4 (2013), pp. 261–283.

[15] Max Kuhn and Kjell Johnson. *Feature engineering and selection: A practical approach for predictive models*. CRC Press, 2019.

[16] Diego Milardovich et al. 'Machine Learning Prediction of Defect Structures in Amorphous Silicon Dioxide'. In: *ESSDERC 2021-IEEE 51st European Solid-State Device Research Conference (ESSDERC)*. IEEE. 2021, pp. 239–242.

[17] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[18] Alexey Natekin and Alois Knoll. 'Gradient boosting machines, a tutorial'. In: *Frontiers in neurorobotics* 7 (2013), p. 21.

[19] S. P. Niblett et al. 'Pathways for diffusion in the potential energy landscape of the network glass former SiO_2'. In: *The Journal of Chemical Physics* 147.152726 (2017), p. 7.

[20] Didrik Nielsen. 'Tree boosting with xgboost-why does xgboost win" every" machine learning competition?' MA thesis. NTNU, 2016.

[21] Nga Phung et al. 'The role of grain boundaries on ionic defect migration in metal halide perovskites'. In: *Advanced Energy Materials* 10.20 (2020), p. 1903735.

[22] DL Price and JM Carpenter. 'Scattering function of vitreous silica'. In: *Journal of non-crystalline solids* 92.1 (1987), pp. 153–174.

[23] Omer Sagi and Lior Rokach. 'Ensemble learning: A survey'. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.4 (2018), e1249.

[24] A.-M El-Sayed et al. 'Nature of intrinsic and extrinsic electron trapping in SiO2'. In: *Physical Review B* 89.12 (2014), pp. 125201-1–125201-10.

[25] Al-Moatasem El-Sayed et al. 'Identification of intrinsic electron trapping sites in bulk amorphous silica from ab initio calculations'. In: *Microelectronic Engineering* 109 (2013), pp. 68–71.

[26] Atsuto Seko, Atsushi Togo and Isao Tanaka. 'Descriptors for machine learning of materials data'. In: *Nanoinformatics*. Springer, Singapore, 2018, pp. 3–23.

[27] Linards Skuja. 'Optically active oxygen-deficiency related centers in amorphous silicon dioxide'. In: *Journal of Non-Crystalline Solids* 239 (1998), pp. 16–48.

[28] Robert Street. *Technology and applications of amorphous silicon*. Vol. 37. Springer Science & Business Media, 1999.

[29] Peter V Sushko et al. 'Structure and properties of defects in amorphous silica: new insights from embedded cluster calculations'. In: *Journal of Physics: Condensed Matter* 17.21 (2005), S2115.

[30] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[31] John W Tukey et al. *Exploratory data analysis*. Vol. 2. Reading, MA, 1977.

[32] Dominic Waldhoer et al. 'Atomistic Modeling of Oxide Defects'. In: *Noise in Nanoscale Semiconductor Devices*. Springer, Cham, 2020, pp. 609–648.

[33] Wikipedia. *Dihedral angle*. Retrieved online 24/03/2022. 2022. URL: https://en.wikipedia.org/wiki/Dihedral_angle.

[34] XGBoost. Retrieved online from XGBoost Documentation 24/05/2022. 2021. URL: https://xgboost.readthedocs.io/en/latest/tutorials/model.html.

[35] Yun Xu and Royston Goodacre. 'On splitting training and validation set: a comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning'. In: *Journal of analysis and testing* 2.3 (2018), pp. 249–262.

[36] IV Zolotukhin and Yu E Kalinin. 'Amorphous metallic alloys'. In: *Soviet Physics Uspekhi* 33.9 (1990), p. 720.

Appendix

A Python code

This is the most important parts of the code used to calculate the properties used in the simple descriptor. The code used for EDA and machine learning is not included here since it is mainly plotting and utilizing machine learning functions from a library.

```
class AmorphousOxide:
    def __init__(self, nr_atoms, nr_si, nr_o, xyz_path, defect_known = False):
        self.nrAtoms = nr_atoms
        self.nrSi = nr_si
        self.nrO = nr_o
        self.Si = 0
        self.O  = 1


        xyz_df = pd.read_table(xyz_path, skiprows=1, delim_whitespace=True,
            nrows = self.nrAtoms+1, names=['atom', 'x', 'y', 'z'])
        boxsize = xyz_df.iloc[0]
        self.boxX = float(boxsize["atom"])
        self.boxY = float(boxsize["x"])
        self.boxZ = float(boxsize["y"])

        #Retrieving data from xyz file and putting them in arrays:
        self.nameArray = list(xyz_df["atom"].to_numpy(dtype = str))[1:]
        x = xyz_df["x"].to_numpy()[1:]
        y = xyz_df["y"].to_numpy()[1:]
        z = xyz_df["z"].to_numpy()[1:]

        if not defect_known:
            #If there is a defect it will be in the middle of the box
            defectsx = set(np.where(abs(x - self.boxX/2) < 2e-1)[0])
            defectsy = set(np.where(abs(y - self.boxY/2) < 2e-1)[0])
            defectsz = set(np.where(abs(z - self.boxZ/2) < 2e-1)[0])

            is_defect = list((defectsx.intersection(defectsy)).intersection(defectsz))

            if len(is_defect)>0:
                self.defect = is_defect[0]

        #Implementing periodic boundaries:
        self.xArray = x - self.boxX*np.rint(x/self.boxX)
        self.yArray = y - self.boxY*np.rint(y/self.boxY)
        self.zArray = z - self.boxZ*np.rint(z/self.boxZ)

    def get_closest4(self, index):
        """
        Determines the distances between an atom and its closest four neighboring atoms.

        Parameters
        ----------
        index : int
            Index of atom in coordinate arrays.

        Returns
```

```
        -------
        rclose : numpy array, size 4
            Euclidian distance between the atom, and its four neighboring atoms.
        """
        distX = (self.xArray[index] - self.xArray -
            self.boxX*np.rint((self.xArray[index]-self.xArray)/self.boxX))
        distY = (self.yArray[index] - self.yArray -
            self.boxY*np.rint((self.yArray[index]-self.yArray)/self.boxY))
        distZ = (self.zArray[index] - self.zArray -
            self.boxZ*np.rint((self.zArray[index]-self.zArray)/self.boxZ))

        r = np.sqrt(distX**2 + distY**2 + distZ**2)
        rclose = np.sort(r)[1:5]

        return rclose

    def calc_pdf(self, nhdim, bz):
        """
        Calculates the partial distribution function of the system of atoms.

        Parameters
        ----------
        nhdim : int
            Number of bins.
        bz : float
            Bin size in Å (10^(-10)).

        Returns
        -------
        sisi_pdf : numpy array
            Partial distribution function for Si-Si.
        sio_pdf : numpy array
            Partial distribution function for Si-O.
        oo_pdf : numpy array
            Partial distribution function for O-O.
        sisi_dist : numpy array
            Euclidian distance between Si-Si.
        sio_dist : numpy array
            Euclidian distance between Si-O.
        oo_dist : numpy array
            Euclidian distance between O-O.
        """
        #Normalizing factor
        adhoc = np.array([self.boxX*self.boxY*self.boxZ/(self.nrSi*(self.nrSi-1))/bz,
                          self.boxX*self.boxY*self.boxZ/(self.nrSi*self.nrO
                          - (self.nrSi + self.nrO))/bz,
                          self.boxX*self.boxY*self.boxZ/(self.nrO*(self.nrO-1))/bz])

        count = np.zeros((nhdim, 3))
        dist = np.zeros((nhdim, 3))

        for i in range(len(self.xArray)):
            for j in range(len(self.xArray)):
                dx = self.xArray[i] - self.xArray[j] -
                        self.boxX*np.rint((self.xArray[i]-self.xArray[j])/self.boxX)
                dy = self.yArray[i] - self.yArray[j] -
                        self.boxY*np.rint((self.yArray[i]-self.yArray[j])/self.boxY)
                dz = self.zArray[i] - self.zArray[j] -
```

```
                    self.boxZ*np.rint((self.zArray[i]-self.zArray[j])/self.boxZ)

                dr = np.sqrt(dx**2 + dy**2 + dz**2)

                if dr <= self.boxX/2 and
                        self.nameArray[i] + " " + self.nameArray[j] == "Si Si" and i != j:
                    ll = int(np.rint(dr/bz))
                    count[ll, 0] += 1
                    dist[ll, 0] += dr

                elif dr <= self.boxX/2 and
                        self.nameArray[i] + " " + self.nameArray[j] == "Si O":
                    ll = int(np.rint(dr/bz))
                    count[ll, 1] += 1
                    dist[ll, 1] += dr

                elif dr <= self.boxX/2 and
                        self.nameArray[i] + " " + self.nameArray[j] == "O O" and i != j:
                    ll = int(np.rint(dr/bz))
                    count[ll, 2] += 1
                    dist[ll, 2] += dr
        #-----------------------------------------------------------------------
        rbz = np.zeros((len(count), 3))
        i = np.arange(1, len(count)+1, 1)

        rbz[:,0] = bz*i
        rbz[:,1] = bz*i
        rbz[:,2] = bz*i

        pdf = adhoc*count/(4*np.pi*rbz**2)
        dist = dist/count
        dist = np.nan_to_num(dist)

        sisi_pdf = pdf[:,0]
        sio_pdf = pdf[:,1]
        oo_pdf = pdf[:,2]

        sisi_dist = dist[:,0]
        sio_dist = dist[:,1]
        oo_dist = dist[:,2]

        return sisi_pdf, sio_pdf, oo_pdf, sisi_dist, sio_dist, oo_dist

    def structure_factor(self, q0, qn, dq, f1 = 4.149, f2 = 5.805):
        """
        Calculates the structure factor

        Parameters
        ----------
        q0 : float
            minimum value of scattering vector.
        qn : float
            maximum value of scattering vector.
        dq : float
            stepsize for scattering vector.
        f1 : float
            Form factor of silicon atoms. The default is 4.149.
        f2 : float
```

```
            Form factor of oxygen atoms. The default is 5.805.

        Returns
        -------
        numpy array. Normalized structure factor.
        q : numpy array
            Scattering vector.

        """
        q = np.arange(q0, qn, dq)
        s_fac = np.zeros_like(q)

        si_bool = (np.array(self.nameArray) == "Si")
        f_array = np.ones(self.nrAtoms)*f2
        f_array[si_bool] = f1

        r = np.array([self.xArray, self.yArray, self.zArray]).T

        for j in range(len(q)):
            for i in range(self.nrAtoms):
                f_temp = np.copy(f_array)
                f_temp[i] = 0
                r_temp = np.copy(r)
                r_temp[i] = -1

                s_fac[j] += np.sum(f_array[i]*f_temp*
                            np.sin(q[j]*np.linalg.norm(r[i]-r_temp, axis = 1))/
                            (q[j]*np.linalg.norm(r[i]-r_temp, axis = 1)))
                s_fac[j] += f_array[i]**2

        return s_fac/np.sum(f_array**2), q

def make_bonds(x_array, y_array, z_array, name_array, boxx, boxy, boxz, cutoffs):
    """
    Finds atoms connected by chemical bonds and stores the information in matrixes.
    If atom 1 and atom 2 are bonded, then the value at index 1, 2 in the matrix
    will be non-zero.
    Similar bond matrixes will be made to contain bondlengths of the chemical bonds.

    Parameters
    ----------
    x_array : numpy array
        x-coordinates of atoms.
    y_array : numpy array
        y-coordinates of atoms.
    z_array : numpy array
        z-coordinates of atoms.
    name_array : numpy array
        names of atoms. Here: Si or O
    boxx : float
        box size along x-axis.
    boxy :  float
        box size along y-axis.
    boxz :  float
        box size along z-axis.
    cutoffs : list
        Contains the cut-offs which determines if the interatomic distance is
        short enough to be a chemical bonds. [Si-Si, Si-O, O-O]
```

```
    Returns
    -------
    index_2 : 2d numpy array
        Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded, then the value at index 1, 2 in the matrix
        will be non-zero.
    2d numpy array
        Contains euclidian distances between bonded atoms.
    dist : 2d numpy array
        At index 1, 2 is the vectorized distance between atom 1 and atom 2.

    """
    cut_sisi = cutoffs[0]
    cut_sio = cutoffs[1]
    cut_oo = cutoffs[2]

    name_array=np.array(name_array)

    INDEX = np.arange(0,len(name_array),1,dtype=int)
    dist = np.zeros((3, len(name_array), len(name_array)))
    index_2 = np.zeros((len(name_array), len(name_array)), dtype=int)

    OO    = np.full(len(name_array),1)
    SIO   = np.full(len(name_array),2)
    SISI  = np.full(len(name_array),3)

    bondlengths=[]

    for i in range(len(name_array)):
        x1 = x_array[i]
        y1 = y_array[i]
        z1 = z_array[i]

        x2 = x_array
        y2 = y_array
        z2 = z_array

        #find distance between the atoms (with periodic boundary):
        dx = x1 - x2 - boxx*np.rint((x1-x2)/boxx)
        dy = y1 - y2 - boxy*np.rint((y1-y2)/boxy)
        dz = z1 - z2 - boxz*np.rint((z1-z2)/boxz)
        dr = np.sqrt(dx**2 + dy**2 + dz**2)
        dr[i] = np.inf

        if name_array[i] == "Si":
            values_eq=np.logical_and( name_array==name_array[i],dr<cut_sisi)
            values_neq=np.logical_and( name_array!=name_array[i],dr<cut_sio)
            if np.count_nonzero(values_neq) == 3:
                bonds = get_closest4(i, x_array, y_array, z_array, boxx, boxy, boxz)
                n_idx = np.where(dr == bonds[-1])
                values_neq[n_idx] = True

            btype = SISI*values_eq + SIO*values_neq
            btype = btype[btype!=0]

        elif name_array[i] == "O":
            values_eq = np.logical_and(name_array==name_array[i],dr<cut_oo)
```

```
            values_neq = np.full(values_eq.shape, False)

            btype = OO[values_eq]
        else:
            print('Somthing is wrong with data file')


        val=np.logical_or(values_eq,values_neq)
        index_2[i,INDEX[val]] = btype
        bondlengths.append(dr[val])
        dist[:,i] = np.array([dx, dy, dz])

    return index_2, np.array(bondlengths, dtype = object), dist

def match_bonds(index):
    """
    Based on the index matrix this function matches bonds of two atoms
    to make chains of three atoms. For calculating angles later
    The different chain configurations have been seperated for
    convienience

    Parameters
    ----------
    index : 2d numpy array
        Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded, then the value at index 1, 2 in the matrix
        will be non-zero.

    Returns
    -------
    sisisi_index : 3d numpy array
        If atom 1, 2 and 3, is a Si-Si-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    siosi_index : 3d numpy array
        If atom 1, 2 and 3, is a Si-O-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    osio_index : 3d numpy array
        If atom 1, 2 and 3, is a O-Si-O chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.

    """
    sisisi_index = np.zeros((len(index), len(index), len(index)))
    siosi_index = np.zeros((len(index), len(index), len(index)))
    osio_index = np.zeros((len(index), len(index), len(index)))

    SISISI = 7 #Binary 111
    SIOSI = 5 #binary 101
    OSIO = 2 #binary (0)10

    SIO = 2
    SISI = 3

    for i in range(len(index)):
        #types = names[i] #all connected atoms bondtype
        indexes = index[i] #all connected atoms index

        #Finds all Si atoms connected to first Si atom in chain
        sisi_idx = np.where(indexes == SISI)[0]
```

```
        #Finds all O atoms connected to first Si atom in chain
        sio_idx = np.where(indexes == SIO)[0]
        #Finds all Si atoms connected to first O atom
        middle_si = np.where(index[:, i] == SIO)[0]

        for j in range(len(sisi_idx)):
            #finds all Si atoms connected to the middle atom in chain
            last_si = np.where(index[sisi_idx[j]] == SISI)[0]
            last_si = last_si[last_si != i]

            sisisi_index[i][sisi_idx[j]][last_si] = SISISI


        for j in range(len(sio_idx)):
            #Finds all Si atoms connected to the middle atom
            (O-Si bonds are Si-O bonds reversed -> .T)
            last_si = np.where(index.T[sio_idx[j]] == SIO)[0]
            last_si = last_si[last_si != i]

            siosi_index[i][sio_idx[j]][last_si] = SIOSI

        for j in range(len(middle_si)):
            last_o = np.where(index[middle_si[j]] == SIO)[0]
            last_o = last_o[last_o != i]

            osio_index[i][middle_si[j]][last_o] = OSIO

    return sisisi_index, siosi_index, osio_index

def calc_angle(index_matrix, dr):
    """
    Uses the cosine law to calculate the angle between 3 atoms using
    bond lengths.

    Parameters
    ----------
    index_matrix : 3d numpy array of ints and 0s.
        If index_matrix[i, j, k] != 0 there is a bond between atom i, j and j, k.
        The value at this place is different for different bonds.
    dr : 3d numpy array of floats.
        3d numpy array. At dr[i, j] we find the vector between atom i and j.

    Returns
    -------
    angle : 1d numpy array, floats
        Array of angles.

    """
    indexes = np.array(np.where(index_matrix!=0)).T

    index1 = indexes[:, :2]
    index2 = indexes[:, 1:]
    a = np.sqrt(np.sum(dr[:, index1[:,0], index1[:,1]]**2, axis = 0))
    b = np.sqrt(np.sum(dr[:, index2[:,0], index2[:, 1]]**2, axis = 0))
    c = np.sqrt(np.sum(dr[:, index1[:,0], index2[:,1]]**2, axis = 0))

    angle = np.rad2deg(np.arccos((a**2 + b**2 - c**2)/(2*a*b))) # Cosine law ##
    return angle
```

```
def get_dihedrals_fast(siosi_idx, idx):
    """
    Finds the indexes of atoms which is in a dihedral,
    and returns the indexes

    Parameters
    ----------
    siosi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-O-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    idx : 2d numpy array
        Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded, then the value at index 1, 2 in the matrix
        will be non-zero.

    Returns
    -------
    dihed_idx : list of lists
        Contains index of all dihedrals.
        [[atom1, atom2, atom3, atom4], ..., ]

    """
    dihed_idx = []
    SIOSI = 5 #binary 101
    sio_idx = np.array(np.where(idx==2)).T
    chains = np.array(np.where(siosi_idx == SIOSI)).T

    for i in range(len(chains)):
        last_in_chain = chains[i][-1]
        connected_to_last = sio_idx[:,1][sio_idx[:,0] == last_in_chain]

        for atom in connected_to_last:
            if atom not in chains[i]:
                dihed_idx.append([chains[i][0], chains[i][1], chains[i][2], atom])

    return dihed_idx

def calc_dihedral(dihed_idx, dr):
    """
    Calculates the dihedral angle with a list of the index of atoms
    which are a part of the dihedrals.
    Parameters
    ----------
    dihed_idx : list of lists
        Contains index of all dihedrals.
        [[atom1, atom2, atom3, atom4], ..., ]
    dr : 3d numpy array of floats.
        At dr[i, j] we find the vector between atom i and j.

    Returns
    -------
    numpy array
        Contains the dihedral angles.

    """
    dr12 = dr[:,dihed_idx[:,0], dihed_idx[:, 1]].T
    dr23 = dr[:, dihed_idx[:,1], dihed_idx[:,2]].T
```

```
    dr34 = dr[:, dihed_idx[:, 2], dihed_idx[:,3]].T

    x = np.sum(np.linalg.norm(dr23, axis = 1)[:, None]
        *dr12*np.cross(dr23, dr34), axis = 1)
    y = np.sum(np.cross(dr12, dr23)*np.cross(dr23, dr34), axis = 1)

    angle = np.arctan2(x, y)
    return -np.rad2deg(angle)

def two_ring(siosi_idx):
    """
    Finds rings with two silicon atoms in them

    Parameters
    ----------
    siosi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-O-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.

    Returns
    -------
    rings : set of frozensets
        Contains indexes of atoms in a ring of size 2

    """
    rings = set([])
    chains = np.array(np.where(siosi_idx != 0)).T

    for i in range(len(chains)):
        chain = chains[i]
        if frozenset([chain[0], chain[-1]]) not in rings:
            half = np.array(np.where(siosi_idx[chain[0], :, chain[-1]] != 0)).T.flatten()
            two_ring = np.count_nonzero(half[half!=chain[1]])
            if two_ring:
                rings.add(frozenset([chain[0], chain[-1]]))
                #print("Two ring found: ", chain[0], chain[-1])
    return rings

def three_ring(sisisi_idx, sisi_idx):
    """
    Finds rings with three silicon atoms in them

    Parameters
    ----------
    sisisi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-Si-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    sisi_idx : Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded (and both are silicon atoms),
        then the value at index 1, 2 in the matrix will be non-zero.

    Returns
    -------
    rings : set of frozensets
        Contains indexes of atoms in a ring of size 3

    """
    rings = set([])
```

```
    chains = np.array(np.where(sisisi_idx != 0)).T

    for i in range(len(chains)):
        chain = chains[i]
        if sisi_idx[chain[0], chain[-1]]:
            rings.add(frozenset(chain))

    return rings

def four_ring(three_ring, sisisi_idx, sisi_idx):
    """
    Finds rings with four silicon atoms in them

    Parameters
    ----------
    three_ring : set of frozensets
        Contains indexes of atoms in a ring of size 3
    sisisi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-Si-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    sisi_idx : Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded (and both are silicon atoms),
        then the value at index 1, 2 in the matrix will be non-zero.

    Returns
    -------
    rings : set of frozensets
        Contains indexes of atoms in a ring of size 4

    """
    rings = set([])
    chains = np.array(np.where(sisisi_idx != 0)).T

    for i in range(len(chains)):
        chain = chains[i].flatten()

        last_atom = np.array(np.where(sisisi_idx[chain[-1], :, chain[0]] != 0)).flatten()
        if frozenset(chain) not in three_ring:
            for j in range(len(last_atom)):
                if last_atom[j] not in chain:
                    rings.add(frozenset([chain[0], chain[1], chain[2], last_atom[j]]))
    return rings

def five_ring(three_ring, four_ring, sisisi_idx, sisi_idx):
    """
    Finds rings with four silicon atoms in them

    Parameters
    ----------
    three_ring : set of frozensets
        Contains indexes of atoms in a ring of size 3
    four_ring : set of frozensets
        Contains indexes of atoms in a ring of size 4
    sisisi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-Si-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
    sisi_idx : Number of rows and columns equal to number of atoms.
        If atom 1 and atom 2 are bonded (and both are silicon atoms),
```

```
        then the value at index 1, 2 in the matrix will be non-zero.

    Returns
    -------
    rings : set of frozensets
        Contains indexes of atoms in a ring of size 5

    """
    rings = set([])
    chains = np.array(np.where(sisisi_idx != 0)).T

    for i in range(len(chains)):
        chain = chains[i].flatten()
        connections_right = np.array(np.where(sisisi_idx[chain[0], :, :] != 0)).T
        connections_left = np.array(np.where(sisisi_idx[chain[1], chain[-1], :] != 0)).flatten()

        if len(connections_left)>0 and len(connections_right) > 0:
            if frozenset(chain) not in three_ring:
                for j in range(len(connections_right)):
                    for k in range(len(connections_left)):
                        if (sisi_idx[connections_left[k], connections_right[j, 0]] != 0 and
                            connections_left[k] not in chain and
                            connections_right[j,0] not in chain and
                            sisi_idx[connections_right[j,0], chain[1]] == 0 and
                            sisi_idx[chain[1], connections_left[k]] == 0 and
                            sisi_idx[chain[0], connections_left[k]] == 0 and
                            sisi_idx[connections_right[j,0], chain[2]] == 0):
                            if (frozenset([connections_left[k], chain[0], chain[1]])
                                not in three_ring or
                                frozenset([chain[0], chain[1], connections_right[j,0]])
                                not in three_ring or
                                frozenset([connections_left[k], chain[0],
                                connections_right[j,0]]) not in three_ring or
                                frozenset([connections_right[j,0], chain[0], chain[1]])
                                not in three_ring):
                                if (frozenset([connections_right[j,0], chain[0],
                                chain[1], chain[2]])
                                        not in four_ring or
                                        frozenset([chain[0], chain[1],
                                        chain[2], connections_left[k]])
                                        not in four_ring or
                                        frozenset([chain[1], chain[2],
                                        connections_left[k], connections_right[j,0]])
                                        not in four_ring):
                                    rings.add(frozenset([connections_right[j, 0],chain[0],
                                    chain[1], chain[2], connections_left[k]]))
    return rings

def corr_bonds_angles(siosi_idx, dr):
    """
    Finds bonds of Si-O-Si angles and keep track of which bonds
    belong to which angle

    Parameters
    ----------
    siosi_idx : 3d numpy array
        If atom 1, 2 and 3, is a Si-O-Si chain, then at index 1, 2, 3
        there will be a non zero value in the matrix.
```

```
    dr : 3d numpy array of floats.
        3d numpy array. At dr[i, j] we find the vector between atom i and j.

    Returns
    -------
    data : 2d numpy array
        Contains bonds and corresponing Si-O-Si angles.

    """
    si_atoms = np.unique(np.array(np.where(siosi_idx))[0])
    data = np.empty((len(si_atoms), 12), dtype = float)
    idx = 0
    for si in si_atoms:
        chain_idx = np.array(np.where(siosi_idx[si, :,:])).T
        row = []
        for i in range(chain_idx.shape[0]):
            a = np.sqrt(np.sum(dr[:, si, chain_idx[i,0]]**2))
            b = np.sqrt(np.sum(dr[:,  chain_idx[i,0], chain_idx[i,1]]**2))
            c = np.sqrt(np.sum(dr[:, si, chain_idx[i, 1]]**2))

            siosi_angle = np.rad2deg(np.arccos((a**2 + b**2 - c**2)/(2*a*b))) # Cosine law ##

            row.extend([a, b, siosi_angle])

        data[idx] = np.array(row)
        idx += 1
    return data

def find_defect(xyz_directory, cut_off = 0.8, no_post = False):
    """
    Finds the atoms which are considered defect by searching
    the files from the DFT analysis for atoms with
    a spin moment of more than 0.8 under the
    Mulliken Population Analysis

    Parameters
    ----------
    xyz_directory : string
        Directory to DFT files.
    cut_off : float, optional
        Spin moment cut off calue. The default is 0.8.
    no_post : Boolean, optional
        Set it to true if you only want
        pre file indexes. The default is False.

    Returns
    -------
    defect_idx : TYPE
        DESCRIPTION.

    """
    defect_idx = np.array([], dtype = int)
    frame = 0
    nop_frame = 0
    for subdir, dirs, files in os.walk(xyz_directory):
        for file in files:
            filepath = subdir + os.sep + file
            with open(filepath, "r") as f:
```

```
                lines = f.readlines()
                rows = 0
                for line in lines[::-1]:
                    if "Mulliken Population Analysis" in line:
                        break
                    rows+=1
            data = pd.read_table(filepath, skiprows = len(lines)-(rows-2),
                delim_whitespace = True, nrows = 216,
                names = ["Atom", "Element", "Kind", "Atomic population", "(alpha, beta)",
                    "Net charge", "Spin moment"])#lines[rows:rows+219]

            si_moment = data.loc[data["Element"] == "Si"]["Spin moment"].to_numpy(dtype = float)
            if no_post:
                index = np.where(si_moment>cut_off)[0]+nop_frame*72
            else:
                index = np.where(si_moment>cut_off)[0]+frame*72
            defect_idx = np.append(defect_idx, index)
            if index.size>0:
                idx = np.where(si_moment>cut_off)[0]
                print(file, data.loc[data["Element"] == "Si"]["Atom"].iloc[idx], si_moment[idx])
            frame += 2
            nop_frame +=1
    return defect_idx

#-------------------------------------------------------------------------------#
# ANALYZE FUNCTIONS- returns data collected from all files

def analyze_Si(xyz_directory, defect_known = False):
    """
    Returns data sets containing information about bonds and angles
    around each silicon atom in all structures found in
    the xyz-directory

    Parameters
    ----------
    xyz_directory : string
        path to folder containg xyz-files.

    Returns
    -------
    final : 2d numpy array
        for each silicon atom, it contains box id, silicon id, angles and
        bondlengths the atom is involved in.
    idx : list
        indexes of all defects in final.
    corr_final: 2d numpy array
        Contains same information as final. However, it keeps track of which
        Si-O-Si angles corresponds to which bonds.

    """

    frames = 0

    silicons = np.array([])
    angles = np.array([])
    boxid = np.array([])
    bonds = np.array([])
    defects = np.array([], dtype = int)
```

```
defect_idx = []
idx = []

corr_data = np.array([])
corr_box = np.array([])
corr_si = np.array([])
for subdir, dirs, files in os.walk(xyz_directory):
    for file in files:
        filepath = subdir + os.sep + file
        print("Accessing file.... ", file)
        etrap = AmorphousOxide(216, 72, 144, filepath, defect_known)
        ###############################################################################
        #Bonds
        cutoffs = [3.44, 2.0, 3.0]

        index, bond_lengths, dr = make_bonds(etrap.xArray, etrap.yArray, etrap.zArray,
                        etrap.nameArray, etrap.boxX, etrap.boxY, etrap.boxZ, cutoffs)

        ###############################################################################
        #Bond angles and bond lengths
        sisisi_idx, siosi_idx, osio_idx = match_bonds(index)

        siosi_angle = calc_angle(siosi_idx, dr)
        osio_angle = calc_angle(osio_idx, dr)

        r = np.sqrt(np.sum(dr**2, axis = 0))

        siosi_index = np.array(np.where(siosi_idx)).T
        osio_index = np.array(np.where(osio_idx)).T
        si_atoms = np.unique(np.array(np.where(siosi_idx != 0))[0])
        silicons = np.append(silicons, si_atoms)

        if file[-7:] == "pre.xyz":
            if corr_data.size<1:
                corr_data = corr_bonds_angles(siosi_idx, dr)
                corr_box = np.full(len(si_atoms), frames)
                corr_si = si_atoms
            else:
                corr_data = np.append(corr_data, corr_bonds_angles(siosi_idx, dr), axis = 0)
                corr_box = np.append(corr_box, np.full(len(si_atoms), frames))
                corr_si = np.append(corr_si, si_atoms)

        if frames == 0:
             #all surrounding siosi angles
            siosi_angles = np.sort(siosi_angle.reshape(len(si_atoms), 4), axis = 1)[:, ::-1]
            #All surrounding bond lengths
            bonds = r[siosi_index[:,0], siosi_index[:,1]]
            bonds = np.sort(bonds.reshape(len(si_atoms), 4), axis = 1)[:, ::-1]
            ind = np.lexsort((osio_index.T[0], osio_index.T[1]))
            osio_angles = np.unique(np.sort(osio_angle[ind].reshape(len(si_atoms), 12)
            , axis = 1), axis = 1)[:, ::-1]

            angles = np.append(siosi_angles, osio_angles, axis = 1)
            boxid = np.full(len(si_atoms), frames)

        else:
            temp_dr = r[siosi_index[:,0], siosi_index[:,1]]
            ind = np.lexsort((osio_index.T[0], osio_index.T[1]))
```

```
                osio_angle = osio_angle[ind]

                bonds = np.append(bonds, np.sort(temp_dr.reshape(len(si_atoms), 4), axis = 1)
                        [:, ::-1], axis = 0)

                temp_angles = np.append(np.sort(siosi_angle.reshape(len(si_atoms), 4),
                    axis = 1)[:, ::-1], np.unique(np.sort(osio_angle.reshape(len(si_atoms), 12),
                    axis = 1), axis = 1)[:, ::-1], axis = 1)
                angles = np.append(angles, temp_angles, axis = 0)

                boxid = np.append(boxid, np.full(len(si_atoms), frames))

            if file[-8:] == "post.xyz":
                if not defect_known:
                    defects = np.append(defects, np.array([etrap.defect
                        + (frames+1)*72, etrap.defect + frames*72]))
                    #every other defect will come from post and pre
                    def_idx = np.where(si_atoms == etrap.defect)[0][0]
                    idx.append(def_idx + 72*frames)
                    defect_idx.append([frames, etrap.defect])
            frames += 1

    final = np.append(np.array([boxid, silicons]).T,
        np.append(angles, bonds, axis = 1), axis = 1)
    corr_final = np.append(np.array([corr_box, corr_si]).T, corr_data, axis = 1)
    return final, idx, corr_final

def analyze_diheds(xyz_directory, defect_known = False):
    """
    Returns data sets containing information about dihedral angles
    around each silicon atom in all structures found in
    the xyz-directory

    Parameters
    ----------
    xyz_directory : string
        path to folder containg xyz-files.

    Returns
    -------
    dihedral_angles: 2d numpy array
        Contains dihedral angles around atom 1 at index 1.

    dihedral_idx:2d numpy array
        Contains the indexes of atoms participating in the dihedral angles.
        Each row corresponds to an atom

    defect_idx: list
        Contains indexes of the defects

    """
    frames = 0
    dihedral_angles = np.array([])
    dihedral_idx = np.array([])
    defect_idx = []

    for subdir, dirs, files in os.walk(xyz_directory):
        for file in files:
```

```
            filepath = subdir + os.sep + file
            print("Accessing file.... ", file)
            etrap = AmorphousOxide(216, 72, 144, filepath, defect_known)

            ################################################################
            #Bonds
            cutoffs = [3.44, 2.0, 3.0]
            index, bond_lengths, dr = make_bonds(etrap.xArray, etrap.yArray, etrap.zArray,
                           etrap.nameArray, etrap.boxX, etrap.boxY, etrap.boxZ, cutoffs)

            ################################################################
            #dihedral angles
            sisisi_idx, siosi_idx, osio_idx = match_bonds(index)

            dihedrals = np.array(get_dihedrals_fast(siosi_idx, index))
            si_atoms = np.unique(np.array(np.where(siosi_idx != 0))[0])
            temp_angles = np.zeros((len(si_atoms), 24))
            temp_idx = np.zeros((len(si_atoms), 24, 4))

            if file[-8:] == "post.xyz":
                if not defect_known:
                    def_idx = np.where(si_atoms == etrap.defect)[0][0]
                    defect_idx.append(def_idx + 72*frames)

            idx = 0
            for si in si_atoms:
                connected_idx = (dihedrals == si).any(axis = 1)
                dihed_idx = np.array(dihedrals)[connected_idx]
                dihed_angles = calc_dihedral(np.array(dihed_idx), dr)

                first = np.where(dihed_idx[:, 0] == si)[0]
                sec = np.where(dihed_idx[:, 0] != si)[0]

                temp_angles[idx, :12] = dihed_angles[first]
                temp_angles[idx, 12:] = dihed_angles[sec]

                temp_idx[idx, :12] = dihed_idx[first]
                temp_idx[idx, 12:] = dihed_idx[sec]

                idx += 1

            if frames == 0:
                dihedral_angles = temp_angles
                dihedral_idx = temp_idx
            else:
                dihedral_angles = np.append(dihedral_angles, temp_angles, axis = 0)
                dihedral_idx = np.append(dihedral_idx, temp_idx, axis = 0)
            frames += 1

    return dihedral_angles, dihedral_idx, defect_idx

def analyze_rings(xyz_directory, defect_known = False):
    """
    Analyzes xyz-files for ring-structure with 2-5 Si-atoms in them

    Parameters
    ----------
    xyz_directory : string
```

```
    Path to folder containing xyz files.

Returns
-------
rings : 2d numpy array of ones and zeros.
    Each row belongs to a Si atom.
    Each column says if the atom is involved in aring (1) or not (0).
    First column is smallest ring (size 2).
    The size of the ring increases with each column.
defect_idx : 1d numpy array
    Contains the indexes of all defect silicon atoms in the rings array.

"""
frames = 0
rings = np.array([])
defect_idx = []

for subdir, dirs, files in os.walk(xyz_directory):
    for file in files:
        filepath = subdir + os.sep + file
        print("Accessing file.... ", file)
        etrap = AmorphousOxide(216, 72, 144, filepath, False)

        ################################################################################
        #Bonds
        cutoffs = [3.44, 2.0, 3.0]
        index, bond_lengths, dr = make_bonds(etrap.xArray, etrap.yArray, etrap.zArray,
                        etrap.nameArray, etrap.boxX, etrap.boxY, etrap.boxZ, cutoffs)

        ################################################################################
        #rings
        sisisi_idx, siosi_idx, osio_idx = match_bonds(index)

        si_atoms = np.unique(np.array(np.where(siosi_idx != 0))[0])

        sisi_idx = (index == 3)

        rings2 = two_ring(siosi_idx)
        rings3 = three_ring(sisisi_idx, sisi_idx)
        #print(rings3)
        rings4 = four_ring(rings3, sisisi_idx, sisi_idx)
        rings5 = five_ring(rings3, rings4, sisisi_idx, sisi_idx)
        #rings6 = six_ring(rings3, rings4, rings5, sisisi_idx, sisi_idx)


        rings2 = np.sort(np.array([list(x) for x in list(rings2)]).flatten())
        rings3 = np.sort(np.array([list(x) for x in list(rings3)]).flatten())
        rings4 = np.sort(np.array([list(x) for x in list(rings4)]).flatten())
        rings5 = np.sort(np.array([list(x) for x in list(rings5)]).flatten())
        #rings6 = np.sort(np.array([list(x) for x in list(rings6)]).flatten())

        rings2_idx = np.intersect1d(si_atoms, rings2, return_indices = True)[1]
        rings3_idx = np.intersect1d(si_atoms, rings3, return_indices = True)[1]
        rings4_idx = np.intersect1d(si_atoms, rings4, return_indices = True)[1]
        rings5_idx = np.intersect1d(si_atoms, rings5, return_indices = True)[1]
        #rings6_idx = np.intersect1d(si_atoms, rings6, return_indices = True)[1]

        if file[-8:] == "post.xyz":
```

```
                if not defect_known:
                    def_idx = np.where(si_atoms == etrap.defect)[0][0]
                    defect_idx.append(def_idx + 72*frames)

            if frames == 0:
                rings = np.zeros((len(si_atoms), 4))
                rings[rings2_idx, 0] = 1
                rings[rings3_idx, 1] = 1
                rings[rings4_idx, 2] = 1
                rings[rings5_idx, 3] = 1

            else:
                temp_rings = np.zeros((len(si_atoms), 4))
                temp_rings[rings2_idx, 0] += 1
                temp_rings[rings3_idx, 1] += 1
                temp_rings[rings4_idx, 2] += 1
                temp_rings[rings5_idx, 3] += 1

                rings = np.append(rings, temp_rings, axis = 0)

            #Find atoms that appear more than once in one type of ring
            rings = more_than_once(rings2, rings, si_atoms, frames, 0)
            rings = more_than_once(rings3, rings, si_atoms, frames, 1)
            rings = more_than_once(rings4, rings, si_atoms, frames, 2)
            rings = more_than_once(rings5, rings, si_atoms, frames, 3)

            frames += 1
    return rings, defect_idx

def more_than_once(r, rings, si_atoms, frames, n):
    """
    Checks if an atom appear in a single type of ring more than once.

    Parameters
    ----------
    r : numpy array, int
        Rings in a certain size. [[1, 2, 3], ....] <- atoms 1, 2, 3 are in a
        ring of size 3.
    rings : 2d numpy array
        Keeps count of the occurrences of the rings.
    si_atoms : numpy array, int
        All si atoms in a single box (their indexes).
    frames : int
        Which number box/frame we are on.
    n : int
        which column of rings contain the length of rings we are looking at.

    Returns
    -------
    rings : 2d numpy array
        Keeps count of the occurrences of the rings.

    """
    mask = np.ones(len(r), dtype = bool)
    mask[np.unique(r, return_index = True)[1]]= False
    more_than_once = np.unique(r[mask])

    for atom in more_than_once:
```

```
        occurrence = np.count_nonzero(r == atom)
        idx = np.where(si_atoms == atom)[0] + frames*72

        rings[idx, n] += (occurrence-1)
    return rings
```