©Copyright 2022
Simon Fraser

Using Machine Learning to Generate a Surrogate Model for Plasma-Surface Interactions

Simon Fraser

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2022

Reading Committee:

Uri Shumlak, Chair

Eric Meier

Program Authorized to Offer Degree:

Aeronautical and Astronautical Engineering

University of Washington

Abstract

Using Machine Learning to Generate a Surrogate Model for Plasma-Surface Interactions

Simon Fraser

Chair of the Supervisory Committee:
Professor Uri Shumlak
Aeronautical and Astronautical Engineering

Plasma-surface interactions are an important effect in laboratory plasmas, but too complicated to be modelled directly in plasma simulations. However, plasma surface interactions can be modelled by Transport and Range of Ions in Matter (TRIM) simulations based on a binary collision approximation of energetic ions impinging on a stationary material. In this work artificial neural networks are used to generate a model of TRIM simulations for the energy-angular distribution of ions observed at the boundary of a five-moment simulation of the sheared-flow-stabilized (SFS) z-pinch fusion experiment at the University of Washington for graphite and tungsten walls. The trained network then approximates plasma-surface interactions for conditions relevant to the SFS z-pinch fusion experiment. Connecting this model to a plasma simulation as a boundary condition promises to account for plasma-surface interactions for minimal computational expense. To this end boundary conditions representing graphite and carbon walls have been developed for the 5N moment plasma model using the trained models.

TABLE OF CONTENTS

Page

List of Figures . . . iii

Glossary . . . vi

Chapter 1: Introduction . . . 1
- 1.1 Expected Benefits . . . 1
- 1.2 Plasma-Surface Interactions and TRIM . . . 2
- 1.3 Ion-Material Interactions . . . 7
- 1.4 Plasma Models and Simulation . . . 9
- 1.5 The 5N Moment Plasma Model . . . 10
- 1.6 WARPXM . . . 12
- 1.7 Machine Learning . . . 13
- 1.8 Sheared-Flow-Stabilized Z-Pinch . . . 14

Chapter 2: Methods . . . 16
- 2.1 Initial Simulation . . . 16
- 2.2 Gathering TRIM Data . . . 22
- 2.3 Training the Model . . . 25
- 2.4 Boundary Condition . . . 26

Chapter 3: Results . . . 33
- 3.1 Initial Simulation . . . 33
- 3.2 Gathering TRIM Data . . . 35
- 3.3 Training the Model . . . 36
- 3.4 Boundary Condition . . . 46

Chapter 4: Conclusions . . . 60

4.1 Future Work 61

Bibliography 64

Appendix A: TRIM Data Collection Notebook 66

Appendix B: Model Creation Notebook 70

LIST OF FIGURES

Figure Number Page

1.1 TRIM's GUI, its standard interface 5

1.2 TRIM's GUI while TRIM is running, showing some of the values calculated by TRIM and a graph showing locations of simulated ion collisions with lattice atoms 6

1.3 Sample TRIMOUT.txt, the file output by TRIM containing information about sputtered atoms and backscattered ions that must be parsed 8

2.1 Sample coordinate systems for TRIM and WARPXM visualizing the axes we must convert between when using a model trained on TRIM data as a boundary condition for a WARPXM simulation 30

3.1 Initial simulation ion angle of incidence showing the data informing our choice of TRIM angle of incidence 34

3.2 Initial simulation ion energy flux per particle showing the data informing our choice of TRIM incident ion energy 34

3.3 Loss of the carbon model during training showing a significant reduction during training and good agreement between training and validation datasets . . 41

3.4 Mean absolute error of the carbon model during training showing increasing accuracy during training and good agreement between training and validation datasets 41

3.5 Loss of the tungsten model during training showing significant reduction during training but some divergence between training and validation datasets . . 42

3.6 Mean absolute error of the tungsten model during training showing increasing accuracy during training and good agreement between training and validation datasets 42

3.7 Backscattering coefficient of carbon TRIM data showing an increase for higher angles of incidence and reduction for lower angles of incidence and model predicted values showing good agreement 47

3.8 Backscattered energy of carbon TRIM data showing a linear correlation with incident energy and no dependence on angle of incidence and model predicted values showing good agreement 47
3.9 Backscattered x-directional cosine of carbon TRIM data showing a value that corresponds to roughly 48 degrees with few trends and model predicted values showing good agreement 48
3.10 Backscattered y-directional cosine of carbon TRIM data showing a roughly linear correspondence with angle of incidence and model predicted values showing good agreement 48
3.11 Backscattered z-directional cosine of carbon TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement 49
3.12 Sputtering yield of carbon TRIM data showing a nearly linear decrease for values of 200 eV and above, with only a slight dependence on angle of incidence and model predicted values showing good agreement for values of 200 eV and above 49
3.13 Sputtered energy of carbon TRIM data showing a roughly linear dependence on ion energy with some outliers for higher energy values and model predicted values showing good agreement 50
3.14 Sputtered x-directional cosine of carbon TRIM data showing a correspondence to an angle of approximately 28 degrees and model predicted values showing good agreement 50
3.15 Sputtered y-directional cosine of carbon TRIM data showing no preferential y-direction and model predicted values showing only very small predictions, in agreement 51
3.16 Sputtered z-directional cosine of carbon TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement 51
3.17 Backscattering coefficient of tungsten TRIM data showing a nearly linear dependence on incident ion energy with some dip for angles of incidence between 0 and 45 degrees and model predicted values showing good agreement 52
3.18 Backscattered energy of tungsten TRIM data showing a nearly linear dependence on ion energy and model predicted values showing good agreement . . 52
3.19 Backscattered x-directional cosine of tungsten TRIM data showing values corresponding to ann angle of around 47 degrees with some decrease for increasing angle of incidence and model predicted values showing good agreement . . . 53

3.20 Backscattered y-directional cosine of tungsten TRIM data showing a nearly linear dependence on angle of incidence and model predicted values showing good agreement . . . 53
3.21 Backscattered z-directional cosine of tungsten TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement . . . 54
3.22 Sputtering yield of tungsten TRIM data showing no sputtering for low energies then a roughly linear dependence on incident ion energy above around 4 keV and model predicted values showing good agreement . . . 54
3.23 Sputtered energy of tungsten TRIM data showing no sputtering below around 4 keV then a roughly linear dependence on incident ion energy above around 4 keV and model predicted values showing good agreement . . . 55
3.24 Sputtered x-directional cosine of tungsten TRIM data showing no sputtering below around 4 keV then a decrease to greater angles and model predicted values showing a nearly linear trendline, in agreement . . . 55
3.25 Sputtered y-directional cosine of tungsten TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement . . . 56
3.26 Sputtered z-directional cosine of tungsten TRIM data and model predicted values showing only very small predictions, in agreement . . . 56
3.27 Positive and negative charge in the electron boundary condition investigation simulation to t=0.1 with the original presheath boundary conditions on the electrons . . . 57
3.28 Positive and negative charge in the electron boundary condition investigation simulation to t=0.1 with zero normal gradient boundary conditions and the electrons. The similarity to Fig. 3.27 shows that zero normal gradient boundary conditions preserve charge neutrality as well as the original presheath boundary conditions over short time scales . . . 57
3.29 Positive and negative charge in the electron boundary condition investigation simulation to t=1 with the original presheath boundary conditions on the electrons . . . 58
3.30 Positive and negative charge in the electron boundary condition investigation simulation to t=1 with zero normal gradient boundary conditions and the electrons. The similarity to Fig. 3.29 shows that zero normal gradient boundary conditions preserve charge neutrality as well as the original presheath boundary conditions over longer time scales . . . 59

GLOSSARY

DISCONTINUOUS GALERKIN: A class of numerical methods for solving differential equations, abbreviated as DG.

EPOCH: One iteration through the training dataset by a machine learning algorithm.

HYPERPARAMETER: A parameter of a machine learning algorithm used to control the training of the model.

JUPYTER NOTEBOOK: A computational document combining runnable code, text, and images.

MACHINE LEARNING: The study of algorithms that learn from data.

NEURAL NETWORK: A computational model for approximating functions loosely based on biological neural networks, also called a multilayer perceptron and sometimes abbreviated as ANN for artificial neural network.

REGEX: A method of searching through text using a sequence of special characters.

PYSRIM: A Python package for running TRIM calculations and parsing the output created by Christopher Ostrouchov.

SFS: An abbreviation for sheared-flow stabilized, a Z-pinch plasma with an introduced axial flow that helps with mitigating plasma instabilities.

TENSORFLOW: A library for machine learning, particularly neural networks, created and maintained by Google.

TRIM: Transport and Range of Ions in Matter, a Monte Carlo program simulating ion-material interactions using the binary collision approximation.

WARPXM: A program in the WARP (Washington Approximate Riemann Plasma) group of programs used for plasma simulation developed at the University of Washington.

WARPY: A Python interface for WARPXM providing a convenient level of abstraction.

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to the University of Washington Department of Aeronautics and Astronautics, the Computational Plasma Dynamics Group, my family, and my friends.

DEDICATION

to my father

Chapter 1
INTRODUCTION

Since its inception science has sought to model natural phenomena so that they can be studied and understood. This tradition has extended to the study of plasmas, the fourth state of matter. Analytical models have been developed to study plasmas at the levels of individual particles, fluids, or as a single fluid. These models have proven incredibly useful for studying plasmas and have been used to discover and describe phenomena of all sorts. However, some plasma phenomena have proven resistant to analytical modelling. One such phenomena is plasma-surface interactions, what happens when a plasma is in contact with a material. At the level of ions and electrons (the constituent particles of a plasma) the potential outcomes of the interaction are few, and techniques have been developed to determine which of the possible outcomes are most likely to occur. At the level of a complete plasma more outcomes are possible and determining which of those will occur is difficult. In this work we set out to use machine learning to generate a surrogate model for plasma surface interactions based on techniques developed for individual ions.

1.1 Expected Benefits

A very reasonable question to ask is why accounting for the effects of plasma-surface interactions is important and why machine learning was used to account for this effects instead of some other method. Accounting for the effects of plasma-surface interactions is important for the same reasons that having accurate models of any physical phenomena is important. Having accurate models allows for easier understanding of complex phenomena and the creation of useful devices based on those phenomena. For plasmas specifically fusion energy is a promising concept and devices built to achieve fusion require models of their internal phe-

nomena to determine engineering constraints. If the model isn't accurate the device won't function properly and effort and expense will have to be expended to correct for the mistake. In our case specifically, fusion models based on simulations that don't properly account for plasma-surface interactions will have unexplained losses since sputtering is a significant loss mechanism through bremsstrahlung and the introduction of lower energy particles to the plasma and backscattered ions lose energy in the process of being backscattered. Machine learning is used to account for the effects of plasma-surface interactions on account of their being too complicated to model analytically. Instead an empirical approach can be taken, with the complicated program TRIM being used to generate data about plasma-surface interactions for a range of parameters applicable to a given plasma simulation. This data can then be used to train a machine learning model. Once trained this model can be evaluated very quickly for any given data. Other approaches to generating approximations to the generated data are possible but machine learning is a natural choice to do its dealing with high-dimensional input and output data, robustness to noise, and the speed at which models can be evaluated. Dealing with high-dimensional input and output data is important since TRIM takes multiple inputs to produce output, and outputs many different values about the input conditions. Robustness to noise is important since TRIM uses many random numbers in its computations and this inherently leads to noise in its outputs. The speed at which models can be evaluated is paramount since this model will be forming a boundary condition in the simulation and therefore need to be evaluated at every point on the boundary it is applied to.

1.2 Plasma-Surface Interactions and TRIM

The heart of this work is the program TRIM, short for Transport and Range of Ions in Matter, created by James F. Ziegler[24]. TRIM is part of a group of programs known as SRIM, for Stopping and Range of Ions in Matter, which calculate many features of the transport of ions in matter. TRIM is a Monte Carlo program simulating ion-material interactions using the binary collision approximation. This approximation is that an ion travelling through a

material undergoes a series of collisions with single atoms and travels in a straight line between collisions. TRIM simulates energetic ions impinging on a material given the ion's angle of incidence and energy and some information about the material. Since plasmas contain many energetic ions it is reasonable to consider using TRIM to account for what happens when the ions in a plasma impinge on a wall. However TRIM has some practical issues when being considered for use in a plasma simulation. Firstly TRIM is a standalone program that doesn't easily integrate into plasma simulations using traditional programming languages. Secondly, TRIM considers individual ions while most plasma simulations of practical devices use fluid models due to computing limitations. Finally, TRIM is very slow to get accurate results. Getting accurate results from TRIM requires simulation of thousands of ions and each of these ions undergoes a considerable number of interactions that TRIM accounts for. This work addresses all of these issues and allows for TRIM to be used indirectly in plasma simulations.

In more detail TRIM simulates individual ions impinging on a target material that is defined by the user. The target material can be made of any combination of elements, predefined compounds, or custom defined compounds. Target materials can also be made of up to 8 sequential layers of elements, predefined compounds, or custom defined compounds. TRIM can also perform its calculations for materials in either the solid or gas phase, so the state of matter of the material in each layer must also be specified. TRIM must also be told the density of the layer, its width, and its stoichiometry. TRIM can be given a name for each layer for use in generated plots. If using a custom material, or if desired, some energy values that TRIM uses in its calculations must also be specified. These values are the displacement energy, surface binding energy, and lattice binding energy. The displacement energy is the energy that an atom recoiling from a collision needs to overcome the lattice forces in the material and become displaced by more than a single atomic spacing away from its original position. The surface binding energy is that energy that the atoms of the target must overcome to leave the target of the surface. The surface binding energy is particularly important for sputtering, which makes it particularly important for our usage,

as we will see later. The lattice binding energy is the energy that every recoiling atom loses when it leaves its lattice site and recoils in the target. Unlike the target material, the incident ion is allowed only to be a single element. TRIM also takes the ion energy and angle of incidence as input. The angle of incidence is relative to the surface normal, which in TRIM is known as the x-direction. 0 degrees is then perpendicular to the target material (which is assumed to be perfectly flat) and 90 degrees would be completely parallel to the target material. TRIM allows angles of incidence between 0 and 89 degrees (since a 90 degree angle of incidence wouldn't intersect the target at all). TRIM also uses the angle of incidence to define the y-direction, which the change in ion direction assumed to be in the xy-plane and such that the path of any ions with an angle of incidence other than 0 degrees is tiled towards the positive y-axis so the y coordinate of ions decreases as they approach the target material, reaching the origin where the path of the incident ion intersects the surface of the target material. The final piece of information that TRIM takes as input is the type of calculation that the user would like to perform. TRIM is capable of performing 7 different types of calculations, ion distribution and quick calculation of damage, detailed calculation with full damage cascades, monolayer collision steps, calculation of surface sputtering, neutron/electron/photon cascades, various ion energy/angle/positions, and special multi-layer biological targets. Since we are interested in sputtering effects, we will always use the calculation of surface sputtering option. TRIM is generally run through a GUI where all of the aforementioned information can be conveniently input, but can also be run programmatically through packages such as PySRIM, about which we will have more to say later. The GUI is shown as Figure 1.1. The GUI also generates plots that are useful for visualizing how the TRIM calculation works, one of which is shown as part of Figure 1.2. In this graph, the locations of collisions between the incident ions and lattice atoms are shown as red dots. From this it should be clear that TRIM works by tracing the path of incident ions through the material and simulating interactions through sampling statistical models, hence its being a Monte Carlo program. The full TRIM GUI while TRIM is running is also shown in Figure 1.2.

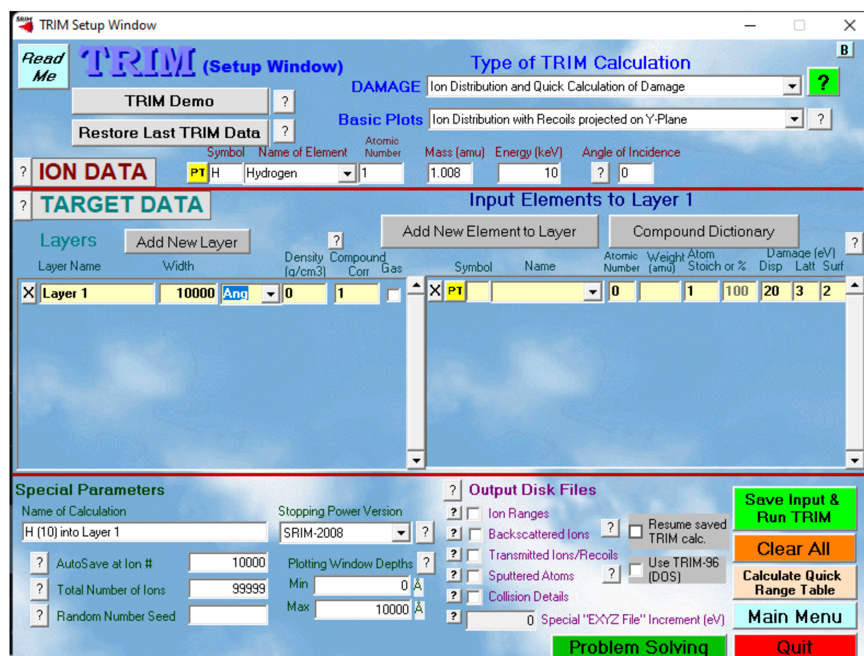

Figure 1.1: TRIM's GUI, its standard interface

As a result of a simulation TRIM outputs several files depending on the type of calculation being performed. Since we perform only calculations of surface sputtering, we focus on the files output as a result of this calculation. Thirteen files are output for this calculation containing information about various parts of the calculation that was just performed. For our purposes, three of these outputs files are most relevant, these files are BACKSCAT.txt, SPUTTER.txt, and TRIMOUT.txt. BACKSCAT.txt contains information about the particles that were backscattered (reflected by the material) in the calculation. Specifically it contains the number of the ion in the simulation (e.g. the first ion incident on the material, second ion, etc.), the atomic number of the ion, the energy of the ion, its last location while being tracked in the simulation, and the cosines of its final direction while being tracked in the simulation. The file SPUTTER.txt gives the same information about sputtered ions, and the file TRIMOUT.txt gives the same information along with whether the ion in question was sputtered, backscattered, or transmitted through the material. The file TRIMOUT.txt

Figure 1.2: TRIM's GUI while TRIM is running, showing some of the values calculated by TRIM and a graph showing locations of simulated ion collisions with lattice atoms

is formed from combining the files SPUTTER.txt, BACKSCAT.txt, and TRANSMIT.txt as those files also have information about whether their atoms were sputtered, backscattered, or transmitted, but the presence of the ion in that file makes that information redundant. A sample TRIMOUT.txt file is included as Figure 1.3. This TRIMOUT.txt file in particular is for carbon with an energy of 1 keV incident at an angle of 0 degrees from normal.

1.3 Ion-Material Interactions

Having now used the terms sputtering and backscattering many times, we now endeavor to more clearly define them. Sputtering refers to the process of energetic ions bombarding the surface of a material and causing particles from the material to be ejected from the material. Sputtering is conventionally quantified using the sputtering yield, which denotes the number of sputtered atoms usually ejected after one incident ion.

$$Sputtering\,Yield = \frac{Number\ of\ Sputtered\ Atoms}{Number\ of\ Incident\ Ions} \tag{1.1}$$

Sputtered atoms are neutral, which is why they are a big problem for fusion plasmas. Atoms that are sputtered by the walls of a fusion experiment will be ionized by the plasma which takes energy away from the plasma and makes it harder to reach ignition conditions. Once being ionized, the ions of the sputtered material will still be present in the plasma and due to their higher atomic number (since no experiment uses Hydrogen walls) they will form a persistent loss mechanism through bremsstrahlung, which scales like the square of the atomic number.

Backscattering refers to a particle that was initially incident on the material being reflected back away from the material. Backscattering is usually quantified using the backscattering coefficient.

$$Backscattering\,Coefficient = \frac{Number\ of\ Backscattered\ Ions}{Number\ of\ Incident\ Ions} \tag{1.2}$$

The other possible interactions when an ion is incident on a material are implantation or

```
============================== SRIM-2013.00 ==============================
==========================================================================
===== TRIMOUT.txt: File of Transmitted / Backscattered / Sputtered Atoms =====
=  This file tabulates the kinetics of ions or atoms leaving the target.     =
=  Column #1: S= Sputtered Atom, B= Backscattered Ion, T= Transmitted Ion.   =
=  Col.#2: Ion Number, Col.#3: Z of atom leaving, Col.#4: Atom energy (eV).  =
=  Col.#5-7: Last location:  X= Depth into target, Y,Z= Transverse axes.     =
=  Col.#8-10: Cosines of final trajectory.                                   =
= *** This data file is in the same format as TRIM.DAT (see manual for uses).=
====== TRIM Calc.=  H(1 keV) ==> None(  100 mm) ==============================
 Ion  Atom   Energy        Depth       Lateral-Position       Atom Direction
 Numb Numb    (eV)          X(A)        Y(A)       Z(A)      Cos(X)  Cos(Y) Cos(Z)
B    8  1 .5157924E+02 -  2954561E-07  .4719E+02 -.1223E+03  -.4361312 -.6020368  .6688358
B   22  1 .3313692E+03 -  7911487E-07 -.1650E+02  .3608E+02  -.9824271 -.1312346 -.1327192
B   27  1 .6855963E+02 -  1537966E-06  .2971E+02 -.3342E+02  -.8143515 -.5283556  .2401500
B   34  1 .5856547E+03 -  9254116E-07 -.2323E+02 -.1430E+02  -.6580908 -.5390222 -.5257106
B   43  1 .4535495E+03 -  9302864E-07  .5937E+02  .5005E+02  -.4982487  .7090930  .4989342
B   66  1 .2237657E+03 -  1319260E-06 -.1701E+03 -.6690E+02  -.9581455 -.2844396  .0324249
S   80  6 .1875439E+02   0000000E+00 -.8505E+01 -.5422E+01  -.8566022 -.5030953  .1145766
S   80  6 .1933824E+02   0000000E+00 -.3857E+01 -.6443E+01  -.9281586  .2412793 -.2833829
S   80  6 .2945482E+02   0000000E+00 -.4210E+01 -.4890E+01  -.9961379  .0824157  .0302817
S   80  6 .1140582E+03   0000000E+00 -.1771E+01 -.5979E+00  -.9957940 -.0374226  .0836296
B   80  1 .7026931E+03 -  7230993E-07 -.1692E+01 -.8257E+01  -.7033084 -.1880610 -.6855585
B   86  1 .2657794E+03 -  1082420E-07  .2378E+02  .2456E+02  -.1437022 -.5660011  .8117835
S   89  6 .3322564E+02   0000000E+00 -.1982E+01  .2143E+02  -.9146796 -.1346651  .3810859
B   89  1 .6965606E+03 -  1184954E-06 -.6350E+01  .9122E+01  -.6697636 -.4010339  .6249708
B   94  1 .4804745E+03 -  5794415E-07  .8313E+02 -.5954E+02  -.7516035  .4495979 -.4826529
S  104  6 .8833131E+01   0000000E+00  .1124E+02 -.9720E+01  -.9860333 -.0102624  .1662318
S  104  6 .2348954E+02   0000000E+00  .1221E+02 -.6679E+01  -.9855675  .0823604  .1478966
B  105  1 .5915358E+03 -  3736401E-07  .3760E+01  .2114E+02  -.8896050 -.2209895  .3997081
B  114  1 .1541328E+03 -  4367949E-07  .4205E+02  .1548E+03  -.5892783  .8071295  .0359599
B  125  1 .5282048E+03 -  7118647E-07  .4769E+02  .1077E+03  -.4784058  .1115198  .8710288
B  158  1 .1163703E+02 -  1745128E-06 -.8798E+02  .7695E+02  -.8975326  .0788930  .4338330
B  162  1 .4196180E+03 -  7113606E-07  .6280E+02 -.2675E+02  -.6968462  .7158221  .0447674
B  204  1 .3630814E+03 -  9065146E-07 -.8523E+01  .7113E+02  -.5835493  .0353651  .8113073
B  212  1 .2382591E+03 -  3828297E-07 -.6421E+02 -.2023E+03  -.3967781  .0398376 -.9170497
B  213  1 .1835664E+03 -  5028205E-07  .5054E+02 -.1338E+03  -.6596795 -.3679237 -.6553283
B  221  1 .5901328E+03 -  5336766E-08  .9023E+02  .2032E+02  -.4073088  .8984266 -.1641013
B  251  1 .3003673E+03 -  1333144E-06  .3439E+02 -.5780E+02  -.8624169  .2416190 -.4448117
```

Figure 1.3: Sample TRIMOUT.txt, the file output by TRIM containing information about sputtered atoms and backscattered ions that must be parsed

transmission. In implantation the incident ion becomes lodged in the lattice of the material, either replacing an atom that was initially there or filling a hole that existed previously. Transmission occurs when the incident ion passes all the way through the material and emerges from the other side. Note that the combined effects of implantation and transmission are accounted for through one minus the backscattering coefficient since all incident ions must be reflected back from the material, pass through the material, or become lodged in the material.

1.4 Plasma Models and Simulation

Plasma simulation refers to any computational attempt to recreate a plasma and its evolution in time and space. Plasma simulations are based on a particular analytical model of plasmas and how they evolve. Simulations allow for the investigation of plasma phenomena at a level of detail that is difficult to achieve in a laboratory environment, and with total knowledge of the plasma. The simplest plasma models used for simulations treat the plasma as a collection of particles evolving according to Maxwell's equations and are known as particle-in-cell or PIC methods. The particles in these methods are often not intended to represent individual physical particles but instead to represent multiple particles as a "superparticle." Despite this simplification, these models are too complex to be used for large scale phenomena as the computational task quickly becomes infeasible. For more complex situations, fluid models of plasmas are used instead. Instead of focusing on particles, these models evolve fluid parameters in time according to governing equations. The most common of these fluid models is the 5N moment model, which assumes a local Maxwellian distribution and achieves a good balance between accuracy and computational complexity. The assumption of a local Maxwellian distribution is motivated by the H-Theorem, which holds that any system with collisions tends towards a Maxwellian distribution. The 5N name denotes that each species in the model has five parameters that must be stored and evolved in time. The 5N moment model is used for all simulations presented here and a derivation is presented in the next section.

1.5 The 5N Moment Plasma Model

The most natural description of a plasma is as a collection of particles of different species subject to electromagnetic forces generated by other particles. This description leads to the Klimontovich equation, which we present as Eq. 1.3. Along with its other defining relations below.

$$\frac{dN_\alpha}{dt} = \frac{\partial N_\alpha}{\partial t} + \frac{\partial}{\partial q} \cdot (\dot{q} N_\alpha) + \frac{\partial}{\partial p} \cdot (\dot{p} N_\alpha) \tag{1.3}$$

$$\dot{q} = \vec{v} \tag{1.4}$$

$$\dot{p} = q_c(\vec{E} + \vec{v} \times \vec{B}) \tag{1.5}$$

Where N represents a number density, q represents a generalized coordinate, p denotes a generalized momentum, the subscript α denotes a specific species, $\vec{v}$ represents velocity, $\vec{E}$ represents the local electric field and $\vec{B}$ represents the local magnetic field. Taking the ensemble average of our discrete particle representation leads to the BBGKY hierarchy. This can be expressed as a Boltzmann equation for each species, which we present as Eq. 1.6.

$$\frac{\partial f_\alpha}{\partial t} + \vec{v} \cdot \frac{\partial f_\alpha}{\partial \vec{x}} + \frac{q_\alpha}{m_\alpha}(\vec{E} + \vec{v} \times \vec{B}) \cdot \frac{\partial f_\alpha}{\partial \vec{v}} = \left.\frac{\partial f_\alpha}{\partial t}\right|_{collisions} \tag{1.6}$$

The zeroth moment of this equation is given by Eq. 1.7

$$\int \frac{\partial f_\alpha}{\partial t} d\vec{v} + \int \vec{v} \cdot \frac{\partial f_\alpha}{\partial \vec{x}} d\vec{v} + \int \frac{q_\alpha}{m_\alpha}(\vec{E} + \vec{v} \times \vec{B}) \cdot \frac{\partial f_\alpha}{\partial \vec{v}} d\vec{v} = \int \left.\frac{\partial f_\alpha}{\partial t}\right|_{collisions} d\vec{v} \tag{1.7}$$

After simplification this leads to the continuity equation for the 5N moment model, given by Eq. 1.8.

$$\frac{\partial n_\alpha}{\partial t} + \nabla \cdot (n_\alpha \vec{v}_\alpha) = 0 \tag{1.8}$$

The first moment of the Boltzmann equation is given by Eq. 1.9.

$$\int \vec{v}\frac{\partial f_\alpha}{\partial t}d\vec{v} + \int \vec{v}\vec{v}\cdot\frac{\partial f_\alpha}{\partial \vec{x}}d\vec{v} + \int \vec{v}\frac{q_\alpha}{m_\alpha}(\vec{E}+\vec{v}\times\vec{B})\cdot\frac{\partial f_\alpha}{\partial \vec{v}}d\vec{v} = \int \vec{v}\left.\frac{\partial f_\alpha}{\partial t}\right|_{collisions} d\vec{v} \tag{1.9}$$

Where we note that two vector quantities next to each other (e.g. $\vec{v}\vec{v}$) is evaluated as a tensor product. After simplification this results in the momentum equation for the 5N moment model, given by Eq. 1.10.

$$\rho_\alpha(\frac{\partial \vec{v}_\alpha}{\partial t} + \vec{v}_\alpha\cdot\nabla\vec{v}_\alpha) + \nabla p_\alpha + \nabla\cdot\overset{\leftrightarrow}{\Pi}_\alpha - q_\alpha n_\alpha(\vec{E}+\vec{v}\times\vec{B}) = \sum_{\beta\neq\alpha}\vec{R}_{\alpha\beta} \tag{1.10}$$

Where $\rho_\alpha = m_\alpha v_\alpha$ is the mass density of species α, p_α is the pressure of species α, $\vec{R}_{\alpha\beta}$ is the momentum transfer vector for species α and β, and $\overset{\leftrightarrow}{\Pi}$ is a stress tensor. Finally for the 5N moment model we consider the reduced second moment of velocity, where instead of evaluating $\vec{v}\vec{v}$ as a tensor product, we evaluate it as a scalar product $\vec{v}\cdot\vec{v} = v^2$, resulting in Eq. 1.11.

$$\int v^2\frac{\partial f_\alpha}{\partial t}d\vec{v} + \int v^2\vec{v}\cdot\frac{\partial f_\alpha}{\partial \vec{x}}d\vec{v} + \int v^2\frac{q_\alpha}{m_\alpha}(\vec{E}+\vec{v}\times\vec{B})\cdot\frac{\partial f_\alpha}{\partial \vec{v}}d\vec{v} = \int v^2\left.\frac{\partial f_\alpha}{\partial t}\right|_{collisions} d\vec{v} \tag{1.11}$$

After simplification this results in the energy equation for the 5N moment model, given as Eq. 1.12.

$$\frac{3}{2}n_\alpha(\frac{\partial T_\alpha}{\partial t} + \vec{v}_\alpha\cdot\nabla T_\alpha) + p_\alpha\nabla\cdot\vec{v}_\alpha + \overset{\leftrightarrow}{\Pi}_\alpha : \nabla\vec{v}_\alpha + \nabla\cdot\vec{h}_\alpha = \sum_{\beta\neq\alpha}Q_{\alpha\beta} \tag{1.12}$$

Where the : operator denotes a tensor contraction in two indices (i.e. $\overset{\leftrightarrow}{P}_\alpha : \nabla\vec{v}_\alpha = \delta_i^k\delta_j^l P^{ij}\partial_k v_l = P^{ij}\partial_i v_j$). While we now have the 5 variables we expect the model is not yet complete as we don't yet know how to evaluate the stress tensor $\overset{\leftrightarrow}{\Pi}$ and heat flux $\vec{h}_\alpha$. These terms are related to the higher moments of the Boltzmann equation, and solving for those moments would introduce even higher moments. This is known as a closure problem as the model cannot be closed without some expression for these terms that can be evaluated. Our expressions for these terms come from a Chapman-Enskog type closure, following Braginskii[3]. The expression for the stress tensor is given by Eq. 1.13.

$$\overleftrightarrow{\Pi} = \nu \nabla^2 \vec{v}_\alpha \tag{1.13}$$

Where ν is a transport coefficient (the viscosity). The expression for the heat flux is given as Eq. 1.14.

$$\vec{h}_\alpha = \kappa \nabla T_\alpha \tag{1.14}$$

Where κ is another transport coefficient (the conductivity). With this closure the 5N moment model is complete and can be used for our simulations.

1.6 WARPXM

WARPXM is a program in the Washington Approximate Riemann Plasma group of programs developed at the University of Washington. This group of programs has been developed to solve hyperbolic conservation laws, especially plasma fluid equations and has been used and detailed in previous work[21][7][8]. WARPXM is implemented in C++ but provides a convenient Python interface known as WARPy which is applicable for running most simulations. Through WARPy discontinuous Galerkin (DG) simulations can be defined through functions at a high level of abstraction without the user needing to worry about how those functions are implemented. To define a simulation in a WARPy script the following process should be followed: define a mesh, define variables, define initial conditions, define applications and spatial solvers, define a writer, define boundary conditions and variable adjusters, define a temporal solver, define a timestep controller, create the simulation, and run the simulation. The mesh is the domain the simulation will occur on, the variables are the parameters that will be adjusted during the simulation, and the initial conditions describe the starting values of the variables. The applications applied to a simulation determine the physical effects that will be accounted for in the simulation and the spatial solvers determine the methods that will be used to solve the differential equations corresponding to those physical effects. The definition of the writer determines which variable values will be output by WARPy and

available for later inspection. The boundary conditions are used in conjunction with the spatial solver to solve the differential equations corresponding to the chosen physical effects and the variable adjusters determine some details of how the variables change including what method for use for calculation of gradients and other details relevant to the DG method. The variable adjusters apply the physics to the variables on selected subdomains and control how gradients are calculated. The temporal solvers determines how the differential equations will be solved in time and the timestep controller determines how large of a timestep the temporal solver will use. Finally the simulation is created and run, combining all of the elements defined above to realize an investigation of interesting phenomena. WARPXM also includes a library of examples of previous work from other members of the Computational Plasmas Group at the University of Washington, who have used WARPXM for simulations for many years.

1.7 Machine Learning

Machine learning refers to the study of algorithms that learn from data. Given nothing more than a dataset these algorithms can learn various actions. The most common machine learning tasks include regression and classification. Regression refers to the approximation of a function given only example outputs and inputs of the function. Classification refers to sorting input data points into one of several output groups given only example sortings of input data. Machine learning has experienced a renaissance in the last decade due to increasing computing power and the increasing availability of large datasets. These features have made possible computational feats that were previously impossible such as effective natural language processing, computer vision, and others. The main model architecture behind recent advances has been the artificial neural network (ANN), a model loosely based on biological neural networks, also called a multilayer perceptron (MLP). In an ANN, a "neuron" is modelled as some activation function acting on the sum of output of "neurons" connected to it from the previous layer or the input data along with a bias. These "neurons" are grouped together to form one or more layers where each "neuron" in the previous layer

is connected to each "neuron" in the next layer. With this architecture, a layer of "neurons" can be represented as a matrix multiplication and evaluated extremely quickly on modern hardware optimized for such calculations such as GPUs. A single layered neural network can approximate any function if it is comprised of enough neurons according to well known universal approximation theorems, many of which exist for different architectures or assuming different activation functions[4][10][9]. An ANN with many layers is called "deep" and many recent successes in machine learning have depended upon this architecture and its derivatives (RNNs, CNNs, Transformers, etc.). The process of using training data to optimize a neural network is called "training" and relies upon the well known technique of gradient descent, although many slight variations of it are often used, such as Adam[11]. After these recent successes, much work has been done to optimize the training and use of neural networks, so they can now be trained very quickly and easily through open source tools. One of the most common of these open source tools is TensorFlow, created and maintained by Google.

1.8 Sheared-Flow-Stabilized Z-Pinch

The specific plasma device that we will be applying our work to is the sheared-flow-stabilized (SFS) Z-pinch experiment ongoing at the University of Washington[17][22][16][19][18]. The standard plasma Z-pinch configuration is based around a strong axial current. This axial current generates azimuthal magnetic fields that tend to confine and heat the plasma. This results in an extremely hot and dense plasma along the pinch axis without any externally applied magnetic fields. A Z-pinch also has the advantage of using its magnetic fields as effectively as possible, as shown by its average β (the ratio of plasma pressure to magnetic pressure) value of 1. The radial force balance describing the Z-pinch equilibrium is given as Eq. 1.15 and the definition of average β is given as Eq. 1.16.

$$\frac{B_\theta}{\mu_0 r}\frac{d}{dr}(rB_r) = -\frac{d}{dr}(p + \frac{B_z}{2\mu_0}) \tag{1.15}$$

$$\langle\beta\rangle = \frac{\langle p\rangle}{\frac{B^2}{2\mu_0}} \tag{1.16}$$

However, the Z-pinch is subject to several instabilities, most notably the so called "kink" and "sausage" instabilities. In the "kink" instability, the current not flowing exactly straight along the pinch axis leads to a self-reinforcing curve. In the "sausage" instability a part of the plasma with a smaller radius experiences larger magnetic field values causing it to contract further. These instabilities prevent equilibrium and if left unchecked can damage the experimental equipment. To deal with these instabilities, the sheared-flow stabilized Z-pinch introduces an axial flow with a sheared profile. The effect of this flow can be intuitively understood as smoothing out the edge of the plasma similar to a bulldozer smoothing out dirt. The equilibrium described by Eq. 1.15 is notably not affected by an axial plasma velocity, so the introduction of a sheared flow doesn't negatively impact the desirable qualities of the Z-pinch configuration. However, one of the main remaining problems of the Z-pinch configuration is that the very hot plasma along the pinch axis is also present at the wall at either end, which are necessary to supply the potential difference driving the axial current. The hot plasma in contact with these walls is a significant loss mechanism both through thermal conduction and the effects of sputtering and backscattering, as mentioned previously. This makes accounting for these effects particularly important for this configuration and motivates this work. The materials used for these electrodes in contact with the plasma in the SFS Z-pinch are either tungsten or graphite so our modelling efforts were focused on these materials. These materials are chosen to minimize the effects of plasma-surface interactions, but these effects are still present and modelling them remains important to get accurate results from simulations.

Chapter 2
METHODS

The overall flow for this work results in four natural stages. In the first stage an initial simulation of the z-pinch fusion experiment was used to gather information about the plasma conditions. In the second stage TRIM was used to collect data for parameters relevant to the plasma conditions observed in the initial simulation. In the third stage, machine learning techniques was used to generate a model that approximates the TRIM data. In the fourth and final stage the machine learning model was used to develop a boundary condition for plasma simulations of the z-pinch to incorporate plasma-surface interactions.

2.1 Initial Simulation

As mentioned above the first stage of this work was to create an initial simulation of the SFS Z-pinch. In this we benefit from the prior work of the Computational Plasma Dynamics Group at the University of Washington, who have previously set up simulations to explore the SFS Z-pinch with a Hydrogen plasma. Specifically work has been done by Dr. Eric Meier to simulate the SFS Z-pinch in one and two dimensions. In one dimension, only the pinch radius is modelled, while in two dimensions the pinch radial and axial directions are modelled. However, both simulations include three dimensions of velocity, making the simulations 1D3V and 2D3V. Both simulations also use the five-moment model with separate fluids for electrons and ions, making them five-moment two fluid models. These simulations are included as part of the WARPXM repository and are therefore available for us to use. Regardless both simulations were rewritten, both to aid in understanding of WARPXM, which is necessary for future development of the final simulation, and to add code to analyze the data generated by the simulation and extract the quantities we need to determine the

applicable parameters for gathering data from TRIM.

To simulate the SFS Z-pinch we use the existing 2D code. The choice to use the 2D code is driven by needing to model two dimensions in our final simulation to see how the sputtered material propagates in the plasma. The domain of this code extends from a radius of 0 (the center of the pinch) to a radius that is four times the pinch radius in the radial direction and has a Z-axis that is as long as the pinch radius. This is much shorter than the physical Z-pinch but as the pinch is in equilibrium and the code uses periodic boundary conditions along the axial direction, modelling only a small axial extent is necessary, and a smaller part saves on unnecessary computational expense. However, we note that using periodic boundary conditions will negatively impact the realism of the data we generate regarding ion energy since at walls sheaths form and tend to accelerate ions significantly. Therefore we are systematically under-representing the energy of ions that we can expect to be incident to the boundary our boundary condition will be applied to. Sheath boundary conditions that would give a better approximation of the energy of ions near an actual wall exist, but are complicated to implement and as implementing those boundary conditions isn't our primary concern, we neglect them. We also note that as our axes correspond to cylindrical geometry, as is the natural choice for a Z-pinch, our fluid and field variables will use cylindrical coordinates. The pinch radius for this code is hard coded to 910 μm. The spatial order chosen for the creation of the mesh defining the domain is 4th order and the mesh is defined to have 64 cells. In keeping with the 5-moment two fluid model, variables are then defined for the ions and electrons that consist of 5 components, the mass density ρ, the components of momentum in all directions p_r, p_θ, and p_z, and the total energy e. A variable is also set up to represent the electromagnetic field with variables E_r, E_θ, E_z, B_r, B_θ, and B_z. The code now turns to setting the initial conditions of the simulation, which requires establishing the value of many quantities used to define an equilibrium and the normalization used by WARPXM. Normalization is important for plasma simulations in particular since plasma values are often extreme in SI or conventional units so using a custom normalization scheme can keep numerical overflow errors from occurring. The

initial condition is set to have an axial current of 3×10^5 A and 5.825 ion gyroradii within the pinch radius. The normalization is then set to have a peak normalized temperature of 1, a characteristic peak magnetic field defined according to Eq. 2.1, a characteristic density defined according to Eq. 2.2, a characteristic speed defined according to Eq. 2.3, a characteristic time defined according to Eq. 2.4, a characteristic pressure given by Eq. 2.5, and characteristic temperature defined by Eq. 2.6.

$$b_0 = \frac{\mu_0 I}{4\pi a} \tag{2.1}$$

$$n_0 = \frac{2Nrgi^2 A_i m_p}{\mu_0 a^2 e^2} \tag{2.2}$$

$$v_0 = \frac{b_0}{\sqrt{m_p n_0 \mu_0}} \tag{2.3}$$

$$\tau = \frac{a}{v_0} \tag{2.4}$$

$$p_0 = \frac{b_0^2}{\mu_0} \tag{2.5}$$

$$T_0 = \frac{p_0}{e n_0} \tag{2.6}$$

Where μ_0 is the vacuum permeability, I is the axial current (which we set to $3 \cdot 10^5$ A), a is the pinch radius, $Nrgi$ is the number of ion gyroradii within the pinch radius, A_i is the normalized mass of the ion, m_p is the mass of a proton, and e is the elementary charge. We also note that these values are all in SI units, so they can function as a conversion between the normalized units of the simulation and SI units for those quantities. The final of these conversion factors we compute at this time is the characteristic current density and current, as equations 2.9 and 2.10 respectively, with $\omega_c \tau$ defined for convenience (it's the proton cyclotron frequency multiplied with the characteristic time and corresponds to the reciprocal of the skin depth as shown in Eq. 2.8).

$$\omega_c \tau = \frac{\tau e b_0}{m_p} \tag{2.7}$$

$$\delta = \frac{1}{\omega_c \tau} \tag{2.8}$$

$$j_0 = \frac{b_0 \omega_c \tau}{a \mu_0} \tag{2.9}$$

$$I_0 = j_0 a^2 \tag{2.10}$$

The code then allows us to choose which type of Z-pinch equilibrium we would like to model. The supported options are a Bennett equilbrium or a Loverich equilibrium. The Bennett equilibrium profile is defined by equations 2.11-2.13[6].

$$B_\theta = \frac{\mu_0 I}{2\pi} \frac{r}{r^2 + a^2} \tag{2.11}$$

$$j_z = \frac{I}{\pi} \frac{a^2}{(r^2 + a^2)^2} \tag{2.12}$$

$$p = \frac{m u_0 I^2}{8\pi^2} \frac{a^2}{(r^2 + a^2)^2} \tag{2.13}$$

The Loverich equilibrium profile is defined by equations 2.14-2.16[14].

$$j_z = \begin{cases} J_0, & r < a \\ 0, & otherwise \end{cases} \tag{2.14}$$

$$B_\theta = \begin{cases} -\frac{1}{2} r \mu_0 J_0, & r < a \\ -\frac{1}{2} \frac{a}{r} \mu_0 J_0, & otherwise \end{cases} \tag{2.15}$$

$$p = \begin{cases} P_0 - \frac{1}{4} \mu_0 J_0^2 r^2, & r < a \\ \alpha \mu_0 J_0^2 a^2, & otherwise \end{cases} \tag{2.16}$$

Where $J_0 = \frac{I_p}{\pi a^2}$ with I_p the normalized pinch current and α a parameter of the equilibrium that sets the minimum pressure (usually set to $\frac{1}{10}$). For this work we use the Bennett

equilbrium due to its wider use in the general community. However, as the simulation is of the SFS Z-pinch a linear shear flow profile is also added to the equilibrium, with the shear flow velocity ranging from 0 at the center of the pinch to v_{sf} at the pinch radius. v_{sf} is defined according to Eq. 2.18.

$$v_{pk} = \frac{b_0}{\sqrt{\mu_0 n_0 A_i m_p}} \tag{2.17}$$

$$v_{sf} = 0.4 \frac{v_{pk}}{v_0} \tag{2.18}$$

Where v_{pk} is a characteristic speed. The initial conditions variable adjusters are then set, incorporating the chosen equilibrium type. With the initial conditions now complete the code turns to the applications and spatial solvers. An Euler solver is chosen with the numerical flux chosen to be a Rusanov flux (we could have also chosen a Roe flux). The code also optionally allows for the inclusion of the effects of inter- and intra-species collisions and we decide to include these effects. The details of these collisions are beyond the scope of this thesis, but are implemented using standard WARPXM applications. We do note that these effects can use different gradient methods in their calculations, and we choose the local DG method (we could have also used an internal penalty method). The normal effects of electromagnetic fields are also accounted for as standard WARPXM applications that this code utilizes. After defining all of these applications a DG spatial solver is defined to use them using Gaussian quadrature.

We now define a writer that outputs the values of all variables (ions, electrons, and fields) in the simulations and their gradients. With that done we move on to defining boundary conditions. For the boundary where the radius is 0, WARPXM has a standard boundary condition for a Z-pinch. In this boundary condition for fluids, copyout boundary conditions are applied to scalar quantities such as ρ and e while reverse copyout boundary conditions are applied to radial and azimuthal components of vector. For the boundary where the radius is at its maximal value WARPXM also has a standard boundary condition for a hard wall. This applies copyout boundary conditions to all fluid variables except the radial momentum,

which it sets according to a reverse copyout boundary condition. Variable adjusters are then defined that specify how the gradient should be calculated (we again use the local DG method). Finally Neumann boundary conditions are set on the variable gradients.

Having now resolved the boundary conditions of the simulation we turn to creating our temporal solver. For this the code uses a third order explicit Runge-Kutta method. The timestep controller is then defined to maximize stability using a built in WARPXM function with details that are again outside of the scope of this work. With everything now defined the simulation can now be created and run.

With the simulation now operational we turn to how we can collect the data we are interested in from the simulation. WARPXM simulations save the data output by the writer to a series of .h5 files, where each file represents a time step in the simulation (not necessarily every time step, as the frequency of output is controlled in the definition of the writer). This filetype is the Hierarchical Data Format and contains multidimensional arrays of data. We can conveniently access these files in Python through the h5py package. To process the data from the simulation in a general way we detect files present in the specified output directory with this extension and process all of them. Since the data we want to collect from this simulation is related to the ion energy and angle of incidence with the boundary of the domain we store the average values of these variables over all timesteps of the simulation. Since we are in equilibrium the timesteps shouldn't be significantly different and all should be equally valid for our use. The angle of incidence we can calculate as Eq. 2.19, which results from the geometry of the situation.

$$\theta = \arccos \frac{p_z}{|p|} \tag{2.19}$$

We perform this calculation for every point in the domain since the simulation uses periodic boundary conditions so points on the boundary are not materially different than points in the middle of the domain along the Z-axis.

Getting the incident ion energy from the simulation is somewhat more difficult since TRIM treats ions as a particle while the simulation treats ions as a fluid, with no regard

for individual particles. To calculate the energy that an ion would have when impacting the wall we settle upon using the ion flux per particle, which considers both the bulk motion and pressure of the fluid. Specifically, the calculation of the ion energy flux per particle begins from the five-moment fluid variables with Eq. 5 from [20], which we present as Eq. 2.20.

$$e = \frac{3}{2}nT + \frac{1}{2}\rho v^2 \tag{2.20}$$

Where $\rho = nZ$ with n as the number density of the species and Z as the normalized charge of the species. We want to use this to get the temperature of the fluid from the five-moment variable e, so we solve Eq. 2.20 for the temperature, resulting in Eq. 2.21

$$T = \frac{2}{3n}(e - \frac{p^2}{2\rho}) \tag{2.21}$$

To convert this temperature to a pressure, we utilize Eq. 6 from [20], which we present as Eq. 2.22.

$$nT = p \tag{2.22}$$

Having now solved for the pressure of the species, we can calculate the ion energy flux per particle in the usual manner, presented as Eq. 2.23

$$\Phi = \frac{(e + p)v}{n} \tag{2.23}$$

This provides our incident ion energy. Having now established how we can get information about the range of ion angles of incidence and ion energies in the SFS Z-pinch, we turn to collecting information about how these ions will interact with a wall.

2.2 Gathering TRIM Data

When gathering data from TRIM our first obstacle is that TRIM is run primarily through a GUI, which is hard to interact with programatically. After some searching, we find that this can be solved through an open source Python package called PySRIM. This package

allows us to run TRIM simulations and parse the files TRIM outputs in a Python script. However, upon further investigation of how we can use this package for our purposes, we discover that it currently doesn't parse the output files most relevant for us. Specifically it can't parse the files related to sputtering, backscattering, and transmission (SPUTTER.txt, BACKSCAT.txt, TRANSMIT.txt, and TRIMOUT.txt). However, the package does have unimplemented classes for parsing these files and asks for anyone that comes across them to implement them. Accordingly, we decide to implement these classes ourselves and contribute to the package.

As mentioned previously the files output by TRIM are text files and can therefore be loaded and parsed with existing tools in Python. Specifically the re module included in Python's standard library allows for the parsing of text using regular expressions, abbreviated as regex. Regular expressions are a convenient way to search bodies of text for sequences of characters and capture certain parts of the text. In our case this means that we have to find a distinctive part of the files to capture the part of the file that contains information relevant for our use. This information is in the table present in the file after a short preamble explaining the columns, as shown in Fig. 1.3. Our task is then to isolate the part of the text file corresponding to the table, after which existing functions in Python packages such as NumPy can be used to create data structures from the table data. After trying multiple options we settle on taking the entirely of the text file after the first appearance of the text "Cos(Z)" followed by a newline. This is used in all of the files relevant to this work, but is different than the format followed by the other files output by TRIM. Following the form of the other parsers in the package we also parse the information present about the input ion, specifically its element and incident energy. This is done through looking for the text "TRIM Calc.= " followed by a letter or two denoting an elemental symbol, then a number followed by eV or keV in parentheses (both the number and eV or keV in parentheses) denoting the energy of the incident ion. As part of this search the elemental symbol and energy of the incident ion are captured and saved. The information in the table is then parsed into a NumPy array using the aforementioned existing tools in NumPy (specifically

the genfromtxt function). Each file that is parsed is represented in PySRIM by a class with attributes corresponding to the data present in the file and we follow this pattern with our newly parsed files. Specifically, each column of the table of data present in the file is represented as a one dimensional NumPy array. After finishing and testing this work to parse the output files we attempt to create a merge request on the PySRIM package to incorporate our changes. However, we find that the package isn't set up to allow merge requests from unauthorized people. We then email the creator and maintainer of the repository with our changes in hopes that the changes could be included through him. However, after weeks no response had been received so we create a fork of the repository hosted on our GitHub that includes the changes so that they are publicly available. We also create an issue on the original repository explaining the situation and linking to our version with the included changes.

With our method for parsing the output of TRIM now established we turn to resolving how we will collect the data we need. Since our initial simulation was set up to simulate a Hydrogen plasma, the incident ion in our TRIM calculations will be Hydrogen as well. The electrodes of the physical SFS Z-pinch are made of either tungsten or graphite so we seek to collect data for both of these cases. TRIM takes three energy values when defining a material that we can accept default values for or set ourselves. In the interest of having the most accurate calculation possible we choose to set them ourselves. These energy values are the displacement energy, lattice binding energy, and surface binding energy, which we discussed in the introduction. ASTM International (formerly the American Society for Testing and Materials) maintains recommendations for these values for many materials. For tungsten the recommended value for displacement energy is 90 eV[2] and the recommended value for lattice binding energy is 0 eV[1]. For the surface binding energy, we follow the recommendation of Yang and Hassanein in using a value of 11.75 eV instead of the default heat of sublimation value (8.68 eV)[23]. Finding these values for graphite is more difficult, but we do end up finding a value for the displacement energy of 25 eV at room temperature[15]. We use the room temperature value instead of the 900 K value (30 eV) since the Z-pinch operates as

a pulsed system so the electrode won't have time to heat up significantly during operation. We aren't able to find references for the lattice and surface binding energies of graphite, so we are forced to use the TRIM default values of 3 eV and 7.41 eV respectively. The final choice we must make before we can use TRIM is the choice of how many ions to simulate at each combination of energy and energy and angle of incidence. TRIM promises errors of less than 10% for 1000 ions, so we decide on using 2000 ions, to have significantly better accuracy, while balancing the additional computation required by adding more ions. With these choices now made, we can perform TRIM calculations for any range of angles and energies that we may find from our initial simulation.

Once the TRIM calculation has been performed we need to parse the output files and get data that can be used to train a machine learning model. To do this we detect and iterate over all files present in a specified TRIM output folder the process the files using our version of the PySRIM package that includes parsers for the files relevant to us. Since each TRIM file contains information about individual ions and we are interested in average quantities for each tested condition we calculate this average for each quantity that TRIM outputs and save this data to an array where each row represents a certain initial condition (combination of ion energy and angle of incidence). This form allows for convenient saving and loading of the data using Python's pickle library and is amenable to training our machine learning model in TensorFlow.

2.3 Training the Model

With the data now collected from TRIM we can create a machine learning model that approximates it. To begin we load the data saved in the previous stage of the process again using Python's pickle library. We explore the data generated by TRIM through generating plots. As we have two input variables and many output variables, we split the data into many 3d surface plots, where the two input dimensions are the ion angle of incidence and ion energy and the output dimension is one of the many variables output by TRIM. Before creating and training the neural network we choose as our machine learning model we follow

the advice of LeCun and Müller and normalize our data to have mean 0 and variance 1 by subtracting out the mean and dividing by the standard deviation[12]. The mean and standard deviation of each variable is also printed so future inputs and outputs of the model can be normalized using the same factors, which is important for getting accurate predictions. We then randomly divide our data into training, testing, and validation sets with a split of 70% training, 20% test, and 10% validation. The training set is the dataset that will be used to update the weights of the neural network, while the testing set is the dataset that will be used to evaluate the model's performance. The validation set is used as a proxy for the testing set during model evaluation. By this we mean that after each epoch (iteration through the training set) the model will be evaluated on the validation set and we can use this to judge if the model still generalizes beyond its training data or is beginning to overfit. Specifically, we use a regularization technique known as early stopping, where model training is stopped after performance on the validation set has stopped improving. With our approach set we can then train the model using TensorFlow. After training the model, its loss over the course of training can be plotted as well as the model's predictions. For the model's predictions we use the same three dimensional surface plot form that we used for the plots of the initial data and for convenience plot them alongside the original data. This makes it easy for us to assess if the model is accurately modelling the data. Once we have a model we are satisfied with, we check the model accuracy on the test set, and save it. This test set should only be used once as using it to evaluate the model's performance multiple times effectively makes the test set the same as the validation set, and doesn't reflect how the model will perform on data it hasn't seen before.

2.4 Boundary Condition

To take advantage of the models we trained, we implement a boundary condition in WARPXM. The simulation we create to test this boundary condition differs from the initial simulation in that the Z-axis domain is now set to be 4 times the length of the radial domain and to no longer be periodic. In the initial simulation the periodic boundary conditions made

resolving the axis unimportant since we expect very little variation along the axis, but the additional axial length is important now that we expect a significant degree of nonuniformity along the axis. This nonuniformity will be due to the boundary conditions now being custom and accounting for plasma-surface interactions such that we expect sheaths to form on both ends of the domain with a region of a more normal Z-pinch between the sheaths. Since this simulation will model sputtered neutrals as well as electrons and ions we must account for them as a fluid in WARPXM. The charge of this fluid is 0, but the mass of this fluid depends on the wall material.

When implementing the boundary condition, the first problem we encounter is that we must access the model we trained in a Python program in a C++ program used by WARPXM. For this we turn to the open source library CppFlow, which provides a convenient wrapper over the TensorFlow C api, which is nonintuitive and hard to use. Through CppFlow we can load the TensorFlow models we've trained and use them to make predictions. To use the models we also need to store the normalization parameters we used in data processing for the model. Before inputting data to the model we first subtract out the mean then divide by the standard deviation and after getting the model predictions we multiply by the standard deviation and add the mean of each parameter.

While we have been talking about this implementation as if it was a single boundary condition, it is really 6 boundary conditions as we need to have conditions for ion, electrons, and neutrals for both carbon and tungsten. The conditions for carbon and tungsten differ only in which model is used so we focus on how the model output is used for ions, neutrals, and electrons. Throughout the creation of the boundary condition we assume that the boundary condition will be applied to a boundary on the z-axis of a system in cylindrical coordinates as this is the case for our simulation. Changes for other simulation geometries would be straightforward modifications of the flux boundary conditions we present below. To implement a boundary condition on the 5N moment plasma model fluxes need to be specified for the momentum and energy. The mass flux for the continuity equation we don't need to specify since we can instead set a zero normal gradient boundary condition, which

will allow the mass to adjust dynamic to remain consistent with the fluxes. The flux for the momentum equation is given by $m_\alpha n_\alpha \vec{v}_\alpha \vec{v}_\alpha + p_\alpha \overleftrightarrow{I} + \overleftrightarrow{\Pi}$ which can be seen from the momentum equation we presented earlier after some manipulation. In accordance with our assumption that all of the energy of the particles in a TRIM simulation is kinetic, we can ignore the pressure and stress tensor terms, leaving us with only $m_\alpha n_\alpha \vec{v}_\alpha \vec{v}_\alpha$. The TRIM contribution to the momentum flux can then be calculated using our assumption that the energy in TRIM particles is entirely kinetic as shown in Eqs. 2.24 - 2.36.

$$v = \sqrt{\frac{2E}{m}} \tag{2.24}$$

$$v_{x,pred} = v \cos\theta_x \tag{2.25}$$

$$v_{y,pred} = v \cos\theta_y \tag{2.26}$$

$$v_{z,pred} = v \cos\theta_z \tag{2.27}$$

$$v_{pred} = \sqrt{v_{x,pred}^2 + v_{y,pred}^2 + v_{z,pred}^2} \tag{2.28}$$

$$A = v/v_{pred} \tag{2.29}$$

$$v_x = Av_{x,pred} \tag{2.30}$$

$$v_y = Av_{y,pred} \tag{2.31}$$

$$v_{z,TRIM} = Av_{z,pred} \tag{2.32}$$

$$v_z = Cv_x \tag{2.33}$$

$$v_r = \sqrt{v_y^2 + v_{z,TRIM}^2} \tag{2.34}$$

$$v_\theta = -C \arctan\frac{v_{z,TRIM}}{v_y} \tag{2.35}$$

$$n_{ion} = Bn_{in} \tag{2.36}$$

Where E is the model predicted average total energy of a backscattered particle, $\cos\theta_i$ is the model predicted directional cosine of the particle along the i-axis (θ_i is the angle between the path of the particle and i-axis), m is the mass of the ion, n_{in} is the number density of the ion fluid used to generate the model predictions, A is a correction factor to

the magnitude of the predicted vector, and C is a correction factor that is 1 if the boundary condition is applied to the minimum z-valued boundary (which we assume to be true if the z-coordinate is 0 or negative) and -1 if the boundary condition is applied to the maximum z-valued boundary (which we assume to be true if the z-coordinate is positive). The factor A arises from two sources, the first is that we are dealing with a machine learning model that has no guarantee of predicting cosines values that combine to correspond to a vector with magnitude 1. The second and more important source is that the data we are training the model on uses average particle values. While each particle output by TRIM will have a combination of directional cosines corresponding to a vector with magnitude 1, the average of each component over many particles will not necessarily create a vector with magnitude 1. The factor A corrects for this by rescaling the predicted vector to have magnitude 1 so the velocity can be calculated appropriately. The factor C can then be used to ensure that the velocity of the particle is being directed into the domain in the case of the z-component of the velocity and in the correct direction in the case of the θ component of the velocity, which is what is shown in Eqs. 2.33 and 2.35. The z-component is multiplied by the factor C since on the boundary with a positive z-coordinate, TRIM's x-axis is aligned with the z-axis of WARPXM so multiplication by C gives a velocity directed back into the simulation domain. The θ-component is multiplied by the negative of C since if TRIM's x-axis is aligned with the z-axis of WARPXM, converting TRIM's r-z coordinates to polar coordinates results in a polar system where the θ-direction is directed similarly to WARPXM's θ-direction and if TRIM's x-axis is aligned with the z-axis of WARPXM, the factor C is -1. This situation is shown in Fig. 2.1, where the x- and z-axes are into the page. This situation corresponds to a boundary with a positive z-coordinate. Note that the y-axis (and subsequently z-axis) of TRIM will rotate in plane according to the direction of the incident ion fluid (recall that the y-axis of TRIM is defined such that the incoming ions approach along the y-axis from the negative side), but the relative orientation of the y- and z-axes will be the same, so the θ-direction will be unchanged.

Recall that the model predicted values use TRIM's Cartesian coordinate system while the

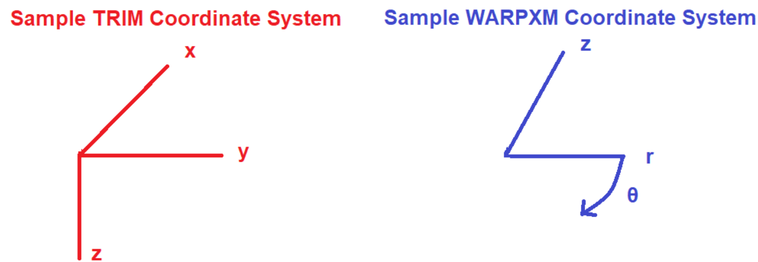

Figure 2.1: Sample coordinate systems for TRIM and WARPXM visualizing the axes we must convert between when using a model trained on TRIM data as a boundary condition for a WARPXM simulation

boundary condition output values use the cylindrical coordinate system of the simulation. We note that "particle" here is imprecise since we are talking about an averaged particle value from TRIM that we are using to calculate fluid properties, but thinking in terms of a single particle is convenient. We also note that the x-axis of TRIM corresponds to the z-axis of the Z-pinch and the y- and z-axes of TRIM form the plane that the r-axis of z-pinch lives in if we set the origin of the two coordinate systems to be the same. Since TRIM is only outputting values about velocities, we don't need to correct for the difference in the location of the origins of TRIM and WARPXM's coordinate systems, which would be important if we were dealing with positions.

The TRIM contribution to the momentum flux represents the flux back into the domain, but we must also account for the flux out of the domain in our boundary condition. To account for this we use the plasma properties at the boundary of the domain in much the same way as when we calculated the TRIM incident energy as the energy flux per particle. In this process we calculated the pressure, and we do so now in the same way. The total momentum flux is then given through the vector summation of the TRIM contribution and

the plasma contribution at the boundary.

With the momentum flux now resolved, we turn to the energy flux. We calculated the plasma contribution to this explicitly earlier before dividing by the number density n_{ion} (Eq. 2.23) and therefore don't need to calculate it again. For the TRIM contribution we calculate the energy from the TRIM predicted values according to Eq. 2.37 then form the energy flux as $e\vec{v}$ since we have no pressure term.

$$e = e_{particle} n \tag{2.37}$$

Where $e_{particle}$ is the model predicted energy of a particle and this value also must be scaled into WARPXM's normalization basis. This energy is notably different than the energy used in the calculation of velocity since that calculation considered the velocity of a single particle and we now want to consider the total energy of the fluid, so we multiply by the number of particles. Finally, the full energy flux is formed through the vector summation of the TRIM contribution and plasma contribution.

For the neutral fluid, the sputtering values predicted by the model are used to generate the momentum flux, since that is the mechanism that acts to change the properties of the neutrals. Since sputtering at the wall creates new neutrals without removing old ones, our boundary condition on the density is given by Eq. 2.38.

$$n_{neutral} = S n_{in} \tag{2.38}$$

Where S is the sputtering yield and n_{in} is the input value of the density of the ion fluid. The components of momentum are calculated in the same manner as for the ions (Eqs. 2.24 - 2.36), but with the total energy E now being given by the model predicted average energy of a sputtered particle. The energy of the neutral fluid is then set according to Eq. 2.37 with $e_{particle}$ now representing the model predicted average energy of a sputtered particle and the energy flux follows naturally.

The effect our model should have on the ions and neutrals falls fairly naturally out from

the definitions of the quantities the model will predict, but the effect the model should have on the electrons is less obvious. The electrons should tend to follow the ions to maintain charge neutrality, whether they implant in the material or get backscattered. However, how to implement this is unclear, so we decide to run a simulation using a toy problem with various electron boundary conditions to investigate how best to model an active wall in WARPXM. We specifically want to ensure that the plasma remains net quasineutral so that the charge lost due to ions implanting in the wall material is reflected in the electron fluid as well. We suspect that setting zero normal gradient boundary conditions on the electrons may let the fluid respond appropriately, but would like to confirm. The setup for our test simulation is of a 1D domain with a presheath on one boundary with a given temperature and number density. The test simulation also includes neutral atoms and molecules. The effects of collisions between the ions and electrons and atomic and molecular reactions between all species are also included. The boundary condition at the other end of the 1D domain from the presheath is a symmetry boundary condition for the center of a one dimensional domain. The temporal solver used for the simulation is a third order Runge-Kutta method. The test consists of changing the electron boundary conditions from original presheath boundary conditions to a zero normal gradient boundary condition and checking if charge neutrality is maintained.

Chapter 3

RESULTS

As the flow for this work resulted in four natural stages, we present the results of this work independently for each stage. The results of each stage depend upon the results of the previous stage, as they form a part of the inputs for that stage.

3.1 Initial Simulation

The calculation of ion angle of incidence performed at every point of the domain in the initial simulation results in Fig. 3.1.

The average angle of incidence in this figure is 0.327 degrees, with the range of angles of incidence ranging from 0 to 94.14 degrees (angles greater than 90 degrees meaning that the fluid z-momentum is in the negative z-direction at that point in the domain). This means that almost all ions in the particle are moving almost entirely along the Z-axis in the positive direction, which we should expect in a Z-pinch with additional sheared (axial) flow. The calculation of ion energy flux per particle at every point in the domain of the initial simulation results in Fig. 3.2

The average value of ion energy flux per particle was 1,092 eV, with values ranging from 189 eV to 12,359 eV. The graph of ion energy flux per particle shows a large cluster of values near the low end and a smaller cluster of values near the high end. This is likely due to the structure of the Z-pinch where energetic particles are clustered around the axis of the pinch, which represents a small fraction of the points present in our mesh. Points further away from the pinch will have less energy and less flux along the z-axis. We may expect a small spike at the value of the flux imparted by the sheared flow, but this flow will dissipate over time and as the presented data is over the entire simulation, the absence of such a small spike

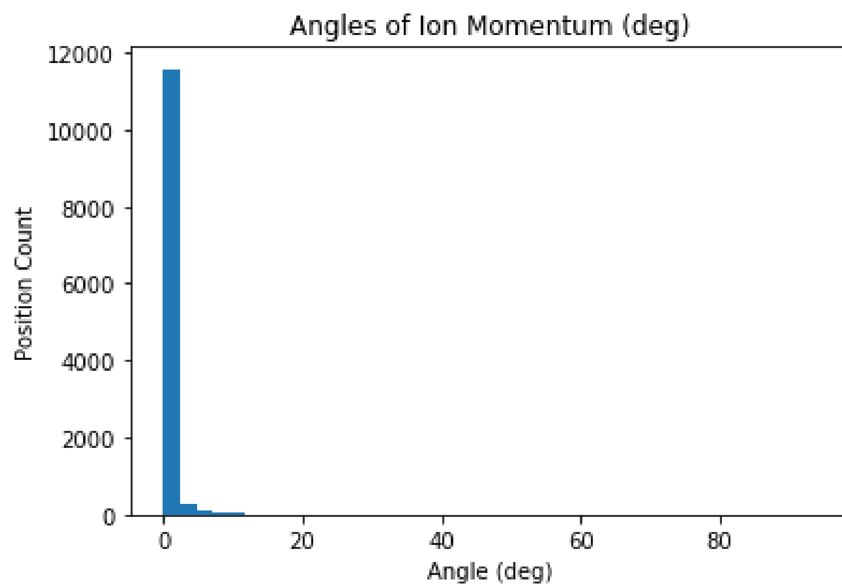

Figure 3.1: Initial simulation ion angle of incidence showing the data informing our choice of TRIM angle of incidence

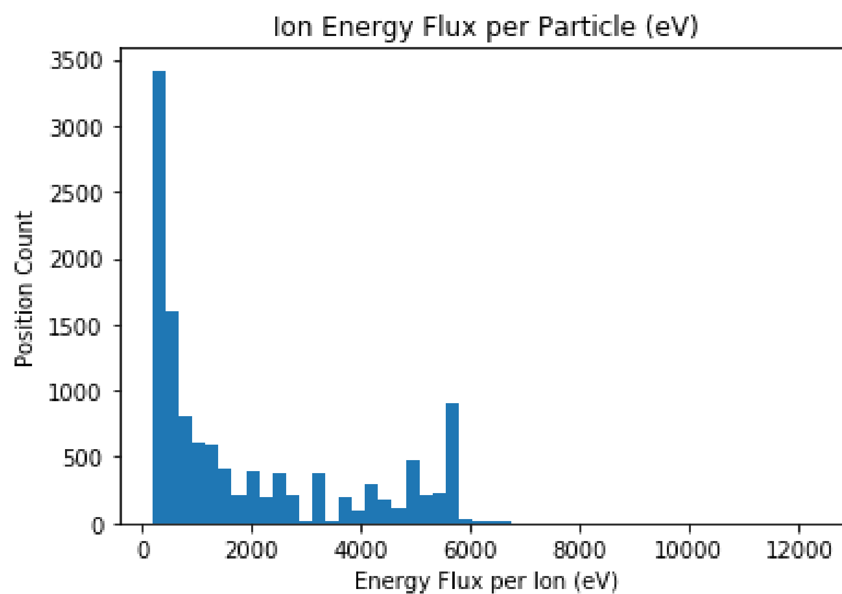

Figure 3.2: Initial simulation ion energy flux per particle showing the data informing our choice of TRIM incident ion energy

isn't concerning.

3.2 Gathering TRIM Data

Based upon the data gathered in the previous section we decide to collect TRIM data on energies from 100 eV to 6000 eV in 100 eV increments and on angles of incidence from 0 to 45 degrees. 45 degrees is chosen somewhat arbitrarily as the data doesn't clearly fall off at that value, but it's important to model a significant portion of the domain as neural networks often give significantly wrong answers for points that aren't in the domain they are trained on and we don't want a larger than expected angle of incidence for a point in the simulation to give dramatically unphysical results. Our chosen value of 45 degrees represents something of a compromise, as it models around half of the domain actually seen in the data, while modelling almost all data. Our choice of stopping at 6 keV is driven by similar concerns with the additional concern that the combination of domain size and resolution in the angles and energy grows the number of points we need to model with TRIM very quickly. Our choice of stopping at 6 keV was also partially driven by a bug in our data analysis code, which showed only data from the last timestep of the simulation, where the largest energy value present was 5779 eV. However, upon the correction of this bug we are reassured by the small amount of additional data from earlier timesteps with energies above 6 keV, and decide that re-generating TRIM data and retraining the model isn't necessary. No data exists at 0 eV since a particle with no energy would never impact the wall. With our chosen domain of angle of incidence and incident energy, we have 2760 conditions to model with TRIM. The data generated is shown in Fig 3.7 - Fig 3.26. We present the data in the next section, alongside the model predicted values for easy comparison and evaluation of model efficacy. The data shown in these figures took days of computation to generate, with each point of the carbon plot taking an average of around 77 seconds to generate on my computer (AMD Ryzen 5 3600 6-core 3.6 GHz CPU and NVIDIA GeForce GTX 1650 SUPER GPU). Each point in the plots represents the values calculated from the 2000 ions TRIM simulated with the initial conditions given by the point. The sputtering yield is calculated according to Eq. 1.1 and

the backscattering coefficient is calculated according to Eq. 1.2. The directional cosines are calculated as the average values of the values TRIM outputs for every ion that sputtered or backscattered. The directional cosines represent the value of the cosine of the angle the particle was travelling at when it left the domain of the TRIM simulation along the direction of the cosine. These directional cosines are therefore TRIM's way of specifying the path the particle was travelling along and are important for us to properly account for the momentum of the particles in the boundary condition we create for the final simulation. The sputtered and backscattered energy at each point is also calculated as the average value of the value TRIM gives for each particle. As shown in these figures, TRIM data is extremely noisy, which is due to TRIM being a Monte Carlo program that depends heavily on random numbers for its computation. This noise also partially justifies our approach using machine learning over an approach using a lookup table. Interpolation of the data would give similarly noisy results while a machine learning approach can generate a model of the data that smoothes over this noise and is probably closer to the real values that would be achieved if more ions were simulated using TRIM. Of course other approaches to smoothing over the noise in the data are possible, but the machine learning approach is convenient due to its simplicity of implementation, good results, and the speed at which the model can be evaluated due to parallelization.

3.3 Training the Model

In our data pipeline, we normalize the data to allow for better training of the machine learning model. The mean and standard deviation of each dimension of our data is summarized in table 3.1 for carbon and table 3.2 for tungsten. These values are used to both normalize the input to the machine learning model and to convert the output of the model back to physical values through doing the normalization process in reverse, i.e. multiplying the value by the standard deviation then adding the mean. Following standard machine learning practices we also batch the data into groups of 32 points, which allows for faster gradient descent over the whole dataset.

Data	Mean	Standard Deviation
Incident Angle (deg)	22.5	13.28
Incident Energy (eV)	3050	1731.81
Backscattering Coefficient	0.064	0.059
Backscattered Energy (eV)	763.29	405.42
Backscattered X-Cosine	-0.67	0.03
Backscattered Y-Cosine	0.1	0.85
Backscattered Z-Cosine	$-3.4 \cdot 10^{-4}$	0.062
Sputtering Yield	0.014	0.0073
Sputtered Energy (eV)	52.62	38.93
Sputtered X-Cosine	-0.88	0.056
Sputtered Y-Cosine	0.0015	0.077
Sputtered Z-Cosine	0.0012	0.08

Table 3.1: Normalization parameters for carbon data

Data	Mean	Standard Deviation
Incident Angle (deg)	22.5	13.28
Incident Energy (eV)	3050	1731.81
Backscattering Coefficient	0.4	0.072
Backscattered Energy (eV)	1513.2	801.15
Backscattered X-Cosine	-0.68	0.011
Backscattered Y-Cosine	0.041	0.034
Backscattered Z-Cosine	$-3.3 \cdot 10^{-5}$	0.017
Sputtering Yield	$5 \cdot 10^{-4}$	0.001
Sputtered Energy (eV)	9.47	18.17
Sputtered X-Cosine	-0.19	0.34
Sputtered Y-Cosine	0.004	0.1
Sputtered Z-Cosine	$4 \cdot 10^{-4}$	0.11

Table 3.2: Normalization parameters for tungsten data

When training a machine learning model, the first choice that must be made is that of model architecture. In our case, a standard feedforward neural network seems appropriate since we want to perform regression. Our data also consists of a series of interrelated variables that are derived from the same input variables, which plays to the strengths of feedforward neural networks. The process of finding the optimal machine learning model is then reduced to the process of hyperparameter optimization. The hyperparameters that we must choose include the number of layers to include in the model, the number of neurons in each layer, the activation function to use in each layer, the optimizer to use for the model, the learning rate of the optimizer, and the number of epochs to wait before stopping the training of the model (termed the "patience"). Through a process of trial and error we settle upon a network with 3 layers, the first layer having 2048 neurons the second layer having 1024 neurons and the last layer having 10 neurons. The number of neurons in the last layer is set by the number of output values we are trying to predict, while we are free to choose the number of neurons in the preceding layers. The number of neurons in the last layer is set by the number of output values since the value of each neuron in the last layer will represent an output value. Typically numbers of neurons that are powers of 2 are chosen for hidden layers since CPUs and GPUs have computing units that come in powers of 2 so choosing the number of neurons this way maximizes computational efficiency. This setup of layers gives us a network with just over 2.1 million parameters, which is a very reasonable size for the complexity of our problem. For the activation function of these layers, we settle upon a regularized linear unit (ReLU) function, which has become something of a standard for neural networks and gives good results on our problem. For the optimizer of the neural network we choose Adam[11], which uses a modified gradient descent algorithm based on estimations of the first- and second-order moments. Adam has also become very standard for neural networks and gives good results for our problem with some refinement of the learning rate. The learning rate is important for our problem in conjunction with the patience of our early stopping since it allows for a balancing between over- and under-fitting. If the learning rate is too large the gradient descent won't be able to learn fine features of the data since the step size will "step

over" the minima while if the learning rate is too small the gradient descent will overfit to the training data and the model won't generalize. After experimentation, we settle at a learning rate of 10^{-4}, which is a standard value but gives the best results for our problem. For our loss function we use the mean squared error, which is standard for regression problems. For the early stopping of training we use a patience of 5 and monitor the loss on the validation set. This means that if the model's loss on the validation set doesn't improve for 5 epochs, we consider the model to be overfitting and stop training. We also set the weights of the model to be the weights that performed best on the validation set (before the 5 epochs where it didn't improve) since after this point, the model was simple overfitting to the training set.

The training loss for the final carbon and tungsten models are shown as Fig. 3.3 and Fig. 3.5 respectively. The mean absolute error during training for the carbon and tungsten models are shown as Fig. 3.4 and Fig. 3.6 respectively. These values are calculated at the end of the corresponding training epoch. The actual values in these graphs aren't particularly meaningful since they are over a training batch and using the normalized values of the data, but the patterns in them can provide useful information about the training of our model. Specifically the relation between the training and loss curves can tell us the degree to which we are overfitting our data. Generally models will perform better on the training set than the validation set and the two curves will begin to separate with the loss curve for the training set dipping to lower values as training goes on. This is what is seen in Fig. 3.4 - Fig. 3.6 but the difference between the curves never gets too large, indicating only a modest degree of overfitting that is unavoidable in training neural networks. In Fig. 3.3 however, the pattern is inverted and the validation loss of the model generally below the training loss until the very end of training. This is likely an artifact of the validation set randomly chosen for carbon to be points that are less noisy than other points in the domain, close to the local average value. Notably the training mean absolute error is below the validation mean absolute error while the losses are inverted so the tendency of the mean squared error (our loss function) to be disproportionately dominated by outliers is likely also a factor. We also note that the carbon model trained for 36 epochs while the tungsten model trained for 62 epochs.

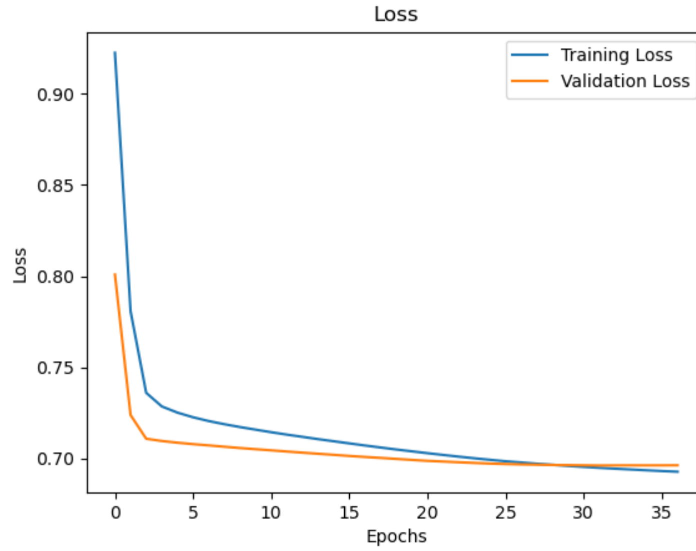

Figure 3.3: Loss of the carbon model during training showing a significant reduction during training and good agreement between training and validation datasets

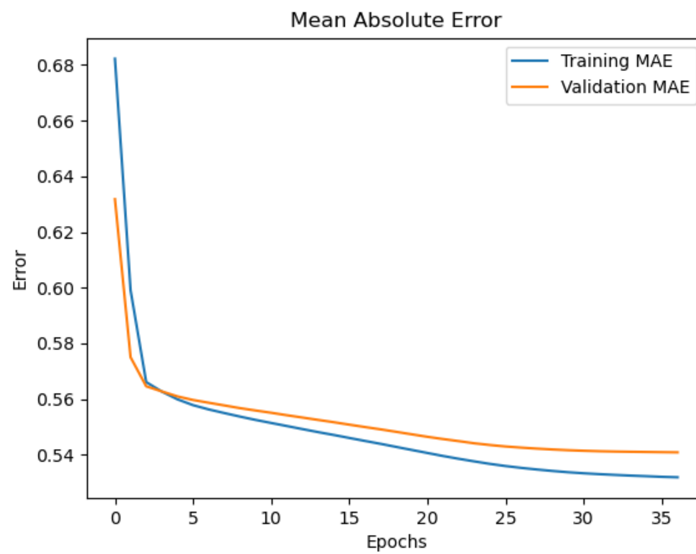

Figure 3.4: Mean absolute error of the carbon model during training showing increasing accuracy during training and good agreement between training and validation datasets

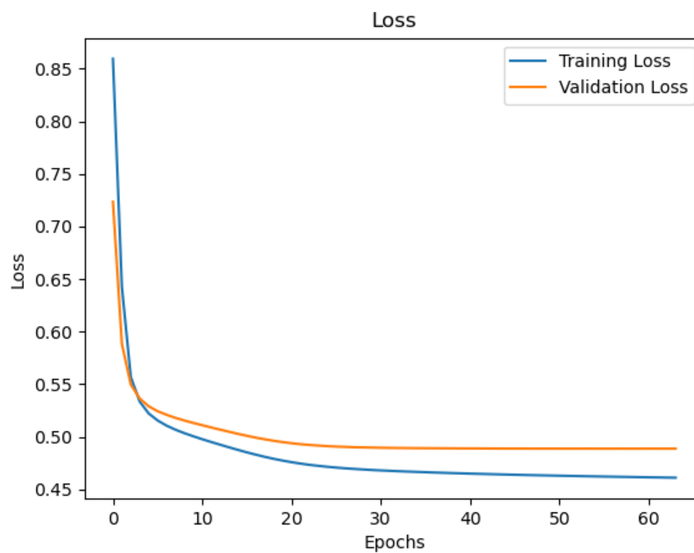

Figure 3.5: Loss of the tungsten model during training showing significant reduction during training but some divergence between training and validation datasets

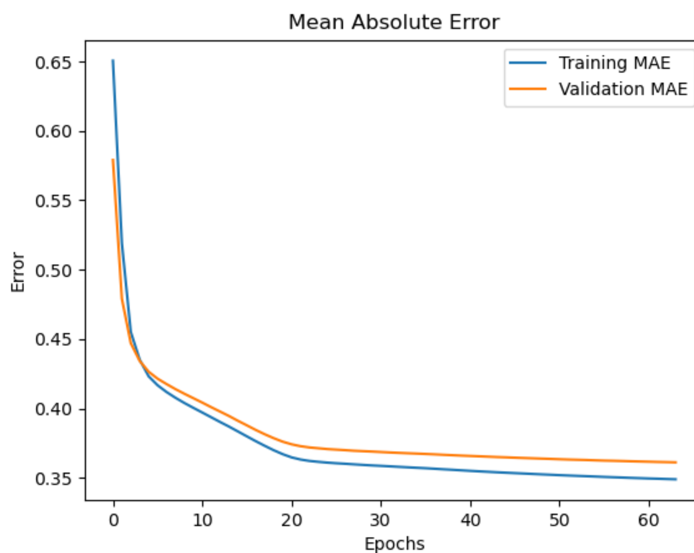

Figure 3.6: Mean absolute error of the tungsten model during training showing increasing accuracy during training and good agreement between training and validation datasets

After training a model for each material, we can visualize its predictions in the same way we initially visualized the TRIM data. Instead of the TRIM data we now plot the model predictions given the ion angle of incidence and energy. These figures form surfaces that are shown alongside the data the model was trained on in Fig. 3.7 - Fig. 3.16 for carbon and Fig. 3.17 - Fig. 3.26 for tungsten. In all of these figures, the model acts to smooth out the data and give very reasonable predictions. Fig. 3.7 shows that the backscattering coefficient of carbon decreases with increasing energy of the incident particle and decreasing angle of incidence and Fig. 3.17 shows the same effect for tungsten. The dependence on energy is particularly strong and is justified by increasing likelihood of the ion implanting in the material as its energy increases and the ion can penetrate further into the material. At higher angles of incidence the ion is less likely to implant in the wall and reflection becomes easier. Our results also show good agreement with previous calculated values, such as those by Eckstein[5]. Fig. 3.8 shows a nearly linear relationship between the incident ion energy and the energy of the backscattered ion for carbon and Fig. 3.18 shows the same effect for tungsten. The two figures do notably have different slopes, which indicates that the energy of backscattered particles depends on the nature of the material the backscattering is occurring off of. This makes sense since the interactions that lead to reflection will be affected by the relative masses of the particles and the binding energies of the material. Ions backscattered off of carbon retain around $\frac{1}{4}$ of their initial energy while ion backscattered off of tungsten retain just under $\frac{1}{2}$ of their initial energy. This shows that during the process of backscattering a significant fraction of the energy of the incident ion is lost to the material, indicating significant heating of the material is occuring. Fig. 3.9 and Fig. 3.19 both show that backscattered particles have an x-directional cosine of around -0.68. Recall that TRIM's coordinate system has the x-direction as normal to the surface with the negative direction pointing away from the surface. An incident ion with 0 angle of incidence travels along the negative x-axis in the positive direction until it intersects the surface at the origin of TRIM's coordinate system. The negative sign in the backscattered x-cosine is then simply necessary for the particle to be backscattered. The value of 0.68 is a bit more opaque,

indicating that the direction of the backscattered particle tends to not be directly along the x-axis, but have components in other directions. Specifically these components tend to result in a vector that makes around a 48 degree angle with the x-axis. This is around what would be expected if the velocity was on average uniformly spread between 0 and 90 degrees from the x-axis (45 degrees would be expected), which are all the possible values so this could also indicate no preferential direction for backscattering with some margin of error. Of course other distributions with an average of 48 degrees are possible. Fig. 3.10 and Fig. 3.20 show that as the angle of incidence increases the angle of the particles with the y-axis also increases. This is logical since the particles are then coming in with some momentum in the y-direction (recall that the angle of incidence occurs in the xy-plane with the y-axis tangential to the surface) so this momentum being conserved makes sense. That this momentum isn't entirely conserved on average (the outgoing cosine reaches a maximum of around 0.1, corresponding to an angle of 84 degrees with the y-axis for an incident angle of 45 degrees) is an indication how how complicated the interactions between the incident ion and the surface material lattice are. Fig. 3.11 and Fig. 3.21 show that the backscattered particles have no preference in the z-direction, averaging to very nearly zero. This is to be expected since the only effect that would give an ion some momentum in the z-direction is a random interaction with a lattice ion and this should be equally likely in the positive and negative z-directions (recall that the z-direction is orthogonal to both components of the initial momentum of an incident ion). We also recall that every point in the cosine figures is itself an average of 2000 incident ions simulated for those initial conditions and some things that may hold true for individual ions need not hold true for the average of many ions. For example, the combination of the x-, y-, and z-cosines for an individual particle should result in a particle that has some velocity on the unit sphere but this isn't the case for many particles since some components (e.g. the z-cosine) may average to 0 while other components average to nonzero values. The sputtering yield plots, Fig. 3.12 for carbon and Fig. 3.22 show that the sputtering yield increases with increasing energy and angle of incidence. Increasing with increasing incident ion energy is intuitive since an incident ion

with more energy will be more able to dislodge lattice atoms. Increasing with increasing angle of incidence is initially less intuitive but means that after incident ions penetrate into the material they will remain closer to the material surface than incident ions with less angle of incidence (recall that angle of incidence is measured from the surface normal) so lattice atoms that become dislodged from their initial location will tend to have less material to travel through to escape from the material and accordingly a greater likelihood of escaping from the material. Carbon also shows an effect where the sputtering yield is initially low (for 100 eV) before jumping to a much higher value. This is likely due to the incident ion needing to supply enough energy to overcome the displacement and surface binding energies and this likelihood increasing as the energy of the incident ion's energy increases past some threshold value that we would need greater resolution to resolve. This effect isn't captured by our model since we only have one point showing this effect and the model views this point as an outlier (and it must view such points as outliers to avoid overfitting in the rest of the dataset). Our model not predicting this value isn't concerning however, as we recall that the mean energy value in our initial simulation was 1901 eV and the minimum value in the initial simulation was 189 eV. Our model predicts the value for 200 eV and above quite well so not capturing this point shouldn't be relevant for our conditions of interest. We also note that for all conditions in our dataset, tungsten has almost no sputtering, with the maximum value of the sputtering yield being an outlier around 0.006. This justifies the use of tungsten as a wall material in devices desiring a hot plasma as sputtering represents a signifcant loss mechanism that should be minimized. Fig. 3.23 shows that tungsten sputtered energy has almost no dependence on ion angle of incidence while Fig. 3.13 shows that carbon sputtered energy has a dependence on ion angle of incidence above around 3 keV and a 25 degree angle of incidence. Above this energy and angle of incidence the sputtered energy remains roughly constant at a value just over 40 eV. The causes of this effect aren't obvious but may be related to the minimum amount of energy an atom may have when dislodged from its initial location in the material lattice or leaving the surface of the material. However, the value of 40 eV has no obvious relation to the displacement energy (25 eV) or surface binding energy

(7.41 eV) we set for carbon. Additionally, the effect isn't visible for tungsten. This may be due to tungsten not sputtering at all below around 4 keV and therefore not reaching the equivalent critical value of energy, and the effect may appear if we calculated higher energies. Finally we note that energy of sputtered particles is much less than the energy of the incident particle so much energy has been lost to the material. Fig. 3.14 shows that the sputtered carbon particles have an average x-directional cosine around -0.88 and Fig. 3.24 shows that after sputtering begins to occur for tungsten, the x-directional cosine quickly approaches similar values. The negative sign here is again necessary for the particle to be sputtered as it indicates a direction of travel out of the material. The value of 0.88 corresponds to an angle of 28 degrees, so the average sputtered particle's velocity makes an angle of around 28 degrees with the x-axis. The causes of this average angle aren't immediately obvious but indicate that most sputtered particles leave the surface at an angle almost normal to the surface. Fig. 3.15 and Fig. 3.16 for carbon and Fig. 3.25 and Fig. 3.26 for tungsten show no preferential direction for sputtering in the y- and z- directions. This is expected in the y-direction since sputtering is a fundamentally different phenomenon then backscattering or reflection and has more to do with interactions within the lattice, which should tend to eject the sputtered particles roughly normally to the surface. We also note that since the directional cosine for both the z- and z-directions are zero on average, the distribution of sputtered particles is symmetric about this axes, implying some uniform cone around the x-axis or some "lumpy" distribution symmetric over the y- and z-axes.

3.4 Boundary Condition

To develop the boundary condition that uses our trained model with the 5N moment plasma model we turn to the sheath investigation that we mentioned in the second chapter that informs our understanding of how to implement this boundary condition for the electron fluid. The experiment took place as described in the second chapter, with the length of the simulation originally going from 0 to 0.1 then 0 to 1. The results of the simulation from t=0 to t=0.1 are shown in Fig. 3.27-Fig. 3.28.

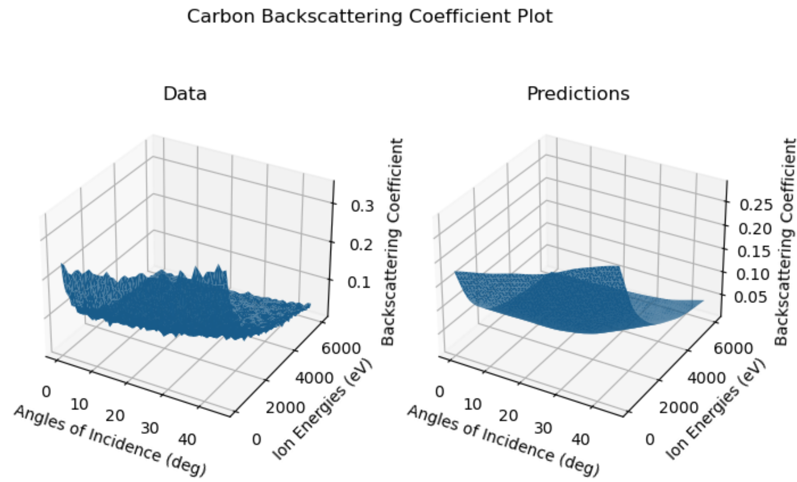

Figure 3.7: Backscattering coefficient of carbon TRIM data showing an increase for higher angles of incidence and reduction for lower angles of incidence and model predicted values showing good agreement

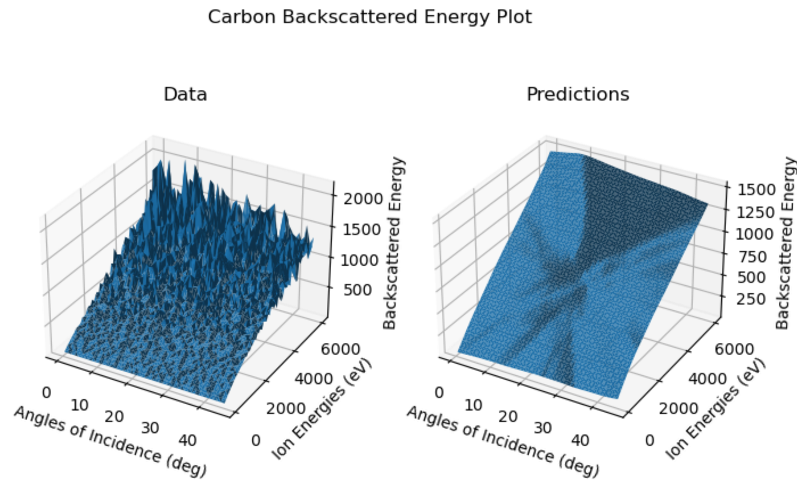

Figure 3.8: Backscattered energy of carbon TRIM data showing a linear correlation with incident energy and no dependence on angle of incidence and model predicted values showing good agreement

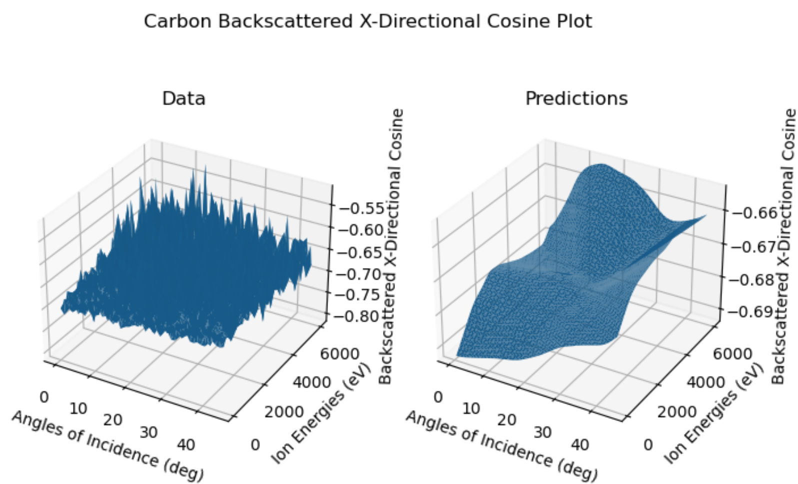

Figure 3.9: Backscattered x-directional cosine of carbon TRIM data showing a value that corresponds to roughly 48 degrees with few trends and model predicted values showing good agreement

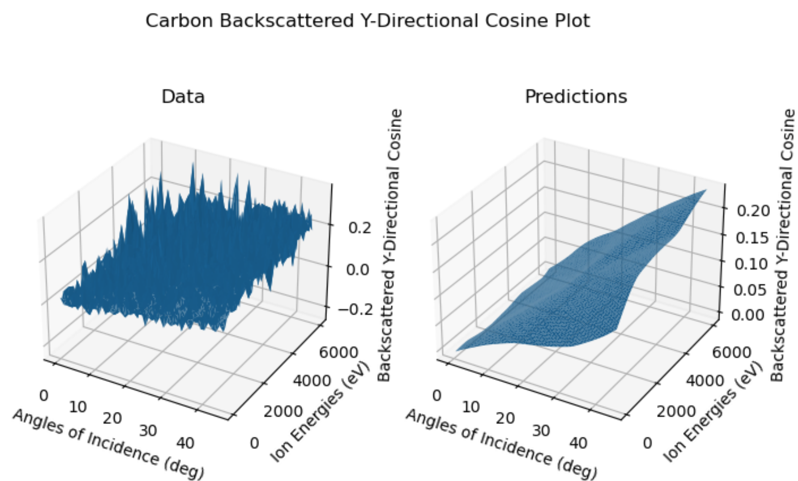

Figure 3.10: Backscattered y-directional cosine of carbon TRIM data showing a roughly linear correspondence with angle of incidence and model predicted values showing good agreement

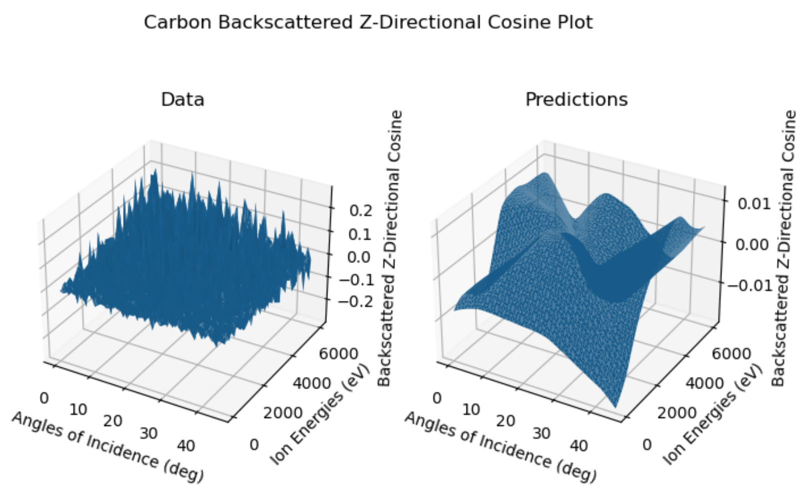

Figure 3.11: Backscattered z-directional cosine of carbon TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement

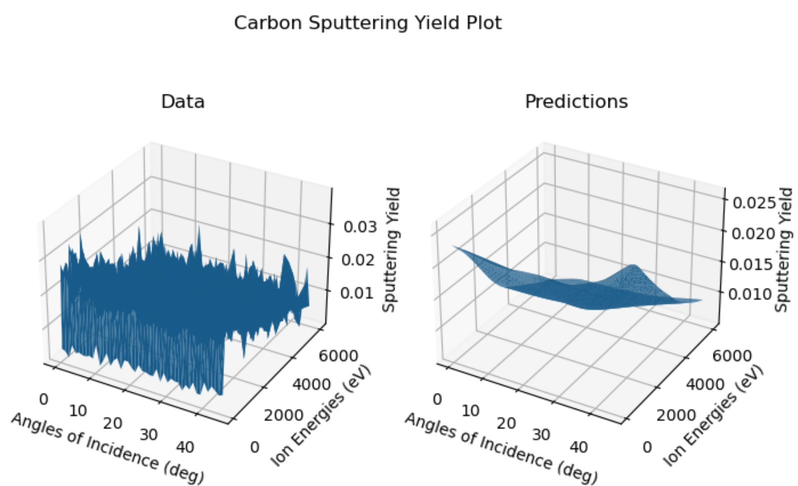

Figure 3.12: Sputtering yield of carbon TRIM data showing a nearly linear decrease for values of 200 eV and above, with only a slight dependence on angle of incidence and model predicted values showing good agreement for values of 200 eV and above

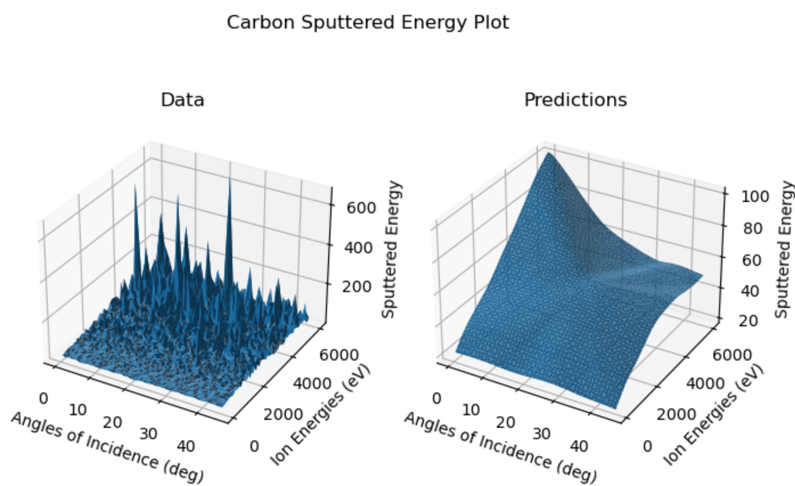

Figure 3.13: Sputtered energy of carbon TRIM data showing a roughly linear dependence on ion energy with some outliers for higher energy values and model predicted values showing good agreement

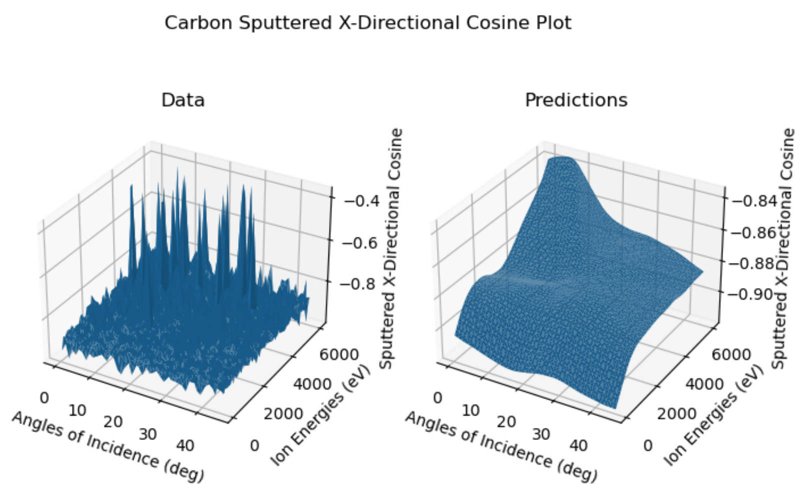

Figure 3.14: Sputtered x-directional cosine of carbon TRIM data showing a correspondence to an angle of approximately 28 degrees and model predicted values showing good agreement

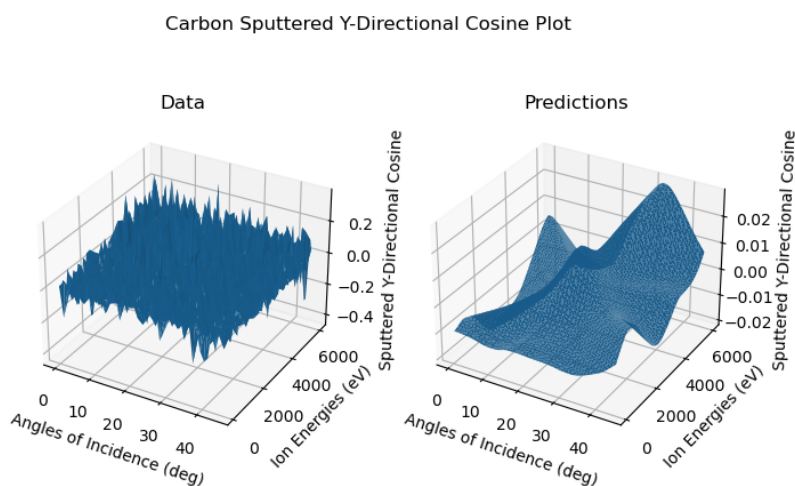

Figure 3.15: Sputtered y-directional cosine of carbon TRIM data showing no preferential y-direction and model predicted values showing only very small predictions, in agreement

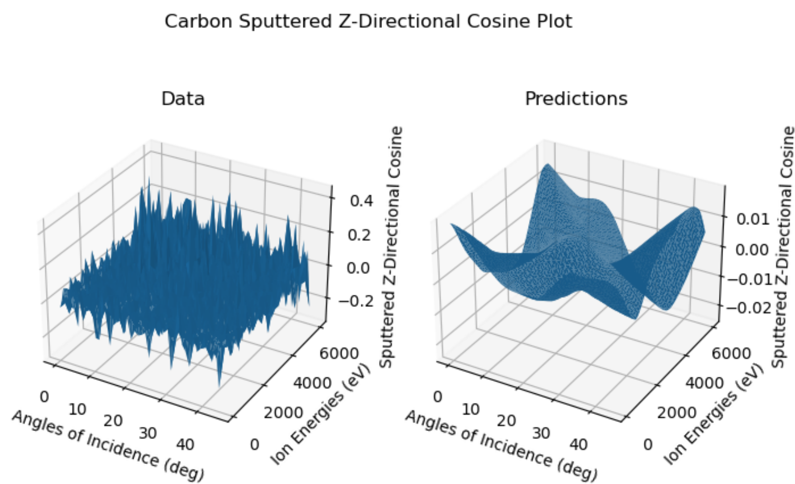

Figure 3.16: Sputtered z-directional cosine of carbon TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement

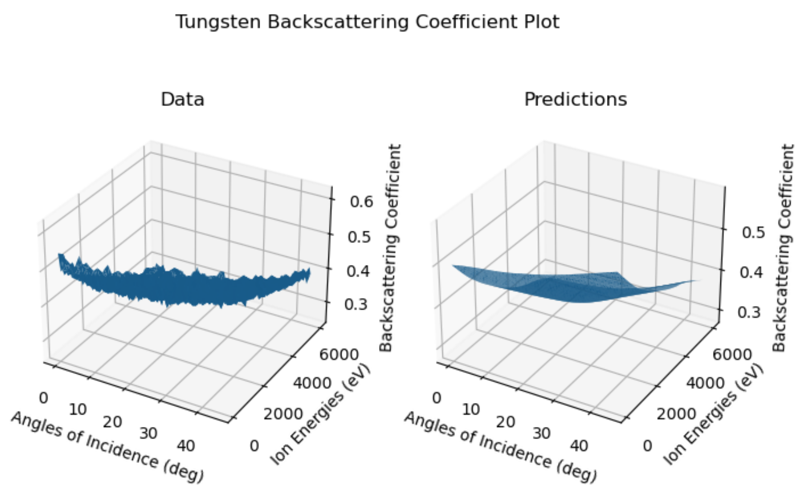

Figure 3.17: Backscattering coefficient of tungsten TRIM data showing a nearly linear dependence on incident ion energy with some dip for angles of incidence between 0 and 45 degrees and model predicted values showing good agreement

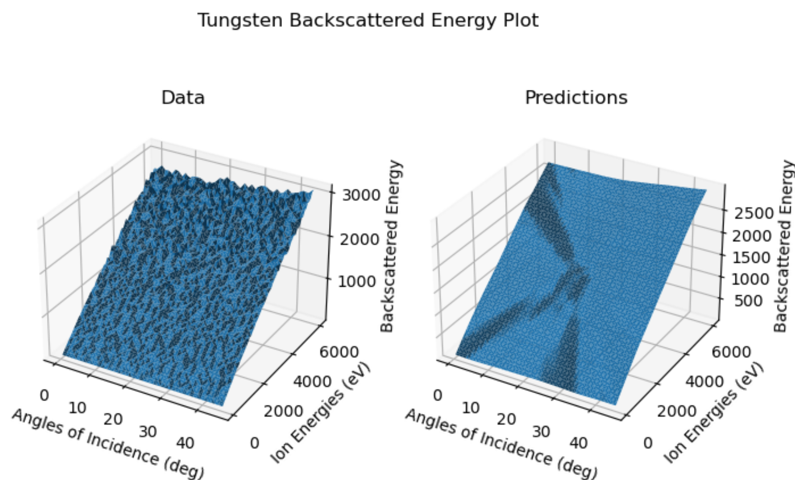

Figure 3.18: Backscattered energy of tungsten TRIM data showing a nearly linear dependence on ion energy and model predicted values showing good agreement

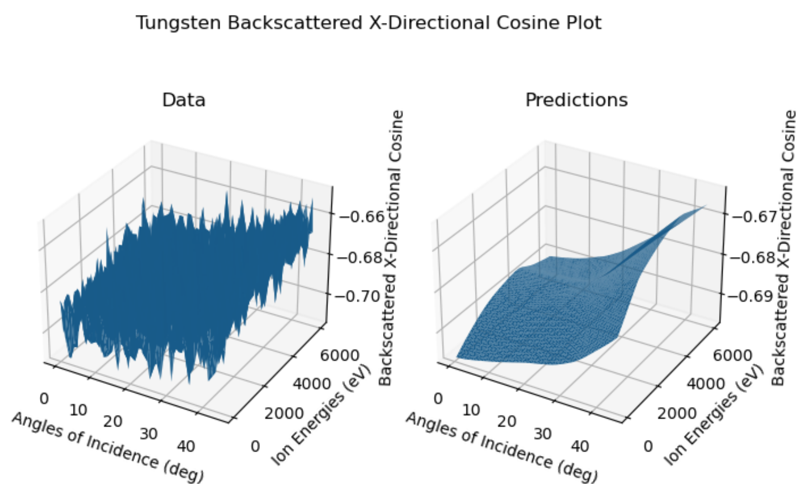

Figure 3.19: Backscattered x-directional cosine of tungsten TRIM data showing values corresponding to ann angle of around 47 degrees with some decrease for increasing angle of incidence and model predicted values showing good agreement

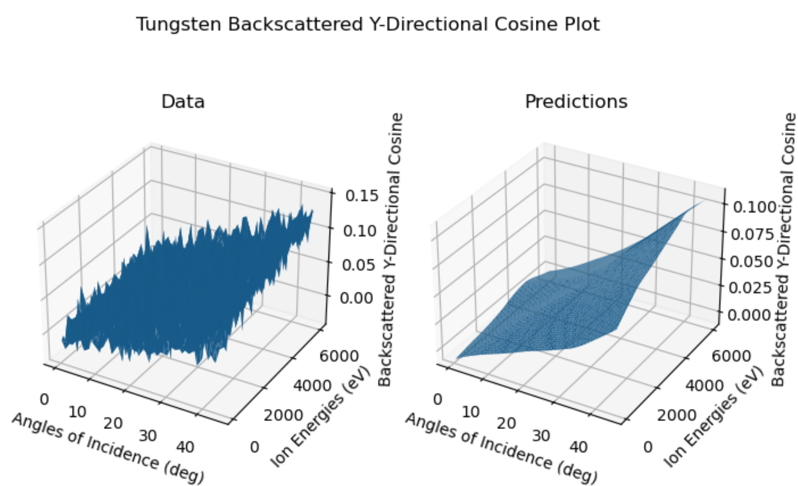

Figure 3.20: Backscattered y-directional cosine of tungsten TRIM data showing a nearly linear dependence on angle of incidence and model predicted values showing good agreement

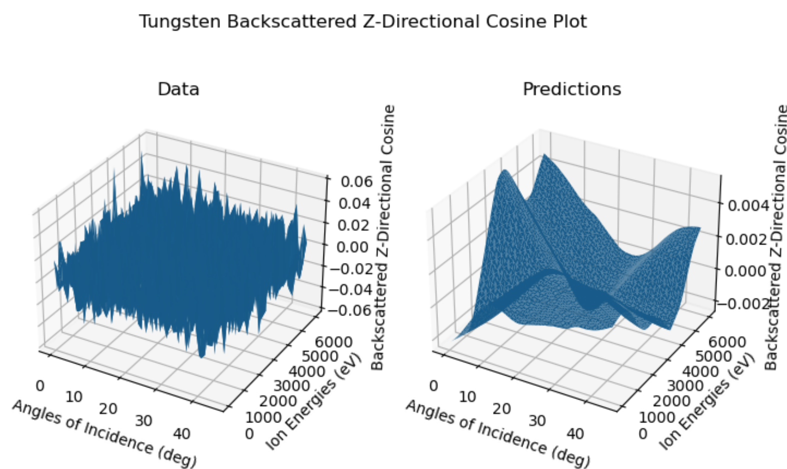

Figure 3.21: Backscattered z-directional cosine of tungsten TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement

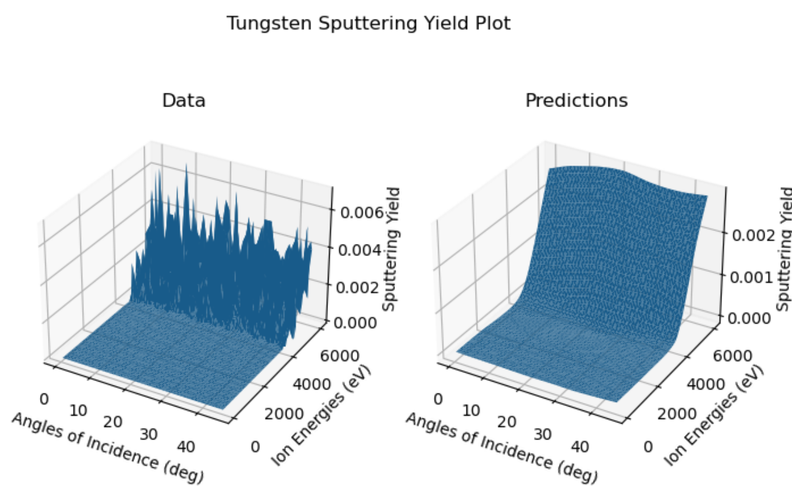

Figure 3.22: Sputtering yield of tungsten TRIM data showing no sputtering for low energies then a roughly linear dependence on incident ion energy above around 4 keV and model predicted values showing good agreement

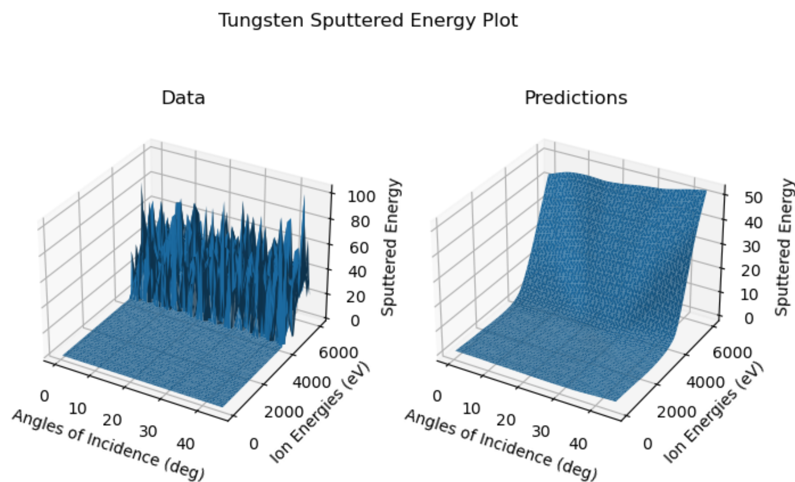

Figure 3.23: Sputtered energy of tungsten TRIM data showing no sputtering below around 4 keV then a roughly linear dependence on incident ion energy above around 4 keV and model predicted values showing good agreement

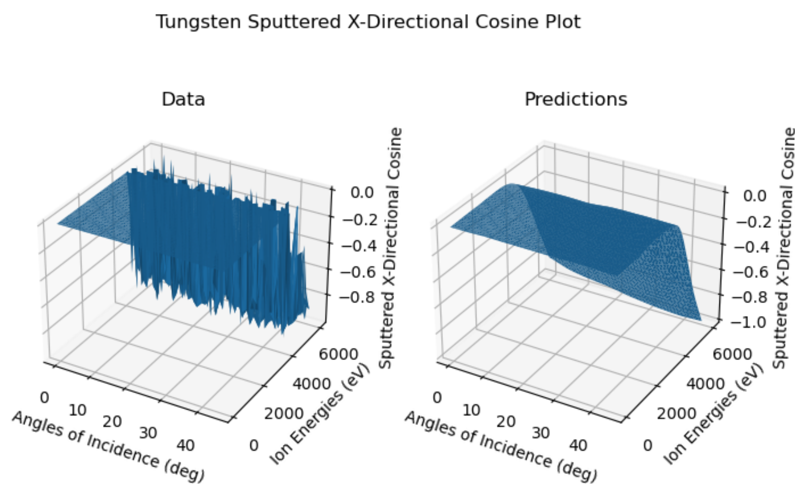

Figure 3.24: Sputtered x-directional cosine of tungsten TRIM data showing no sputtering below around 4 keV then a decrease to greater angles and model predicted values showing a nearly linear trendline, in agreement

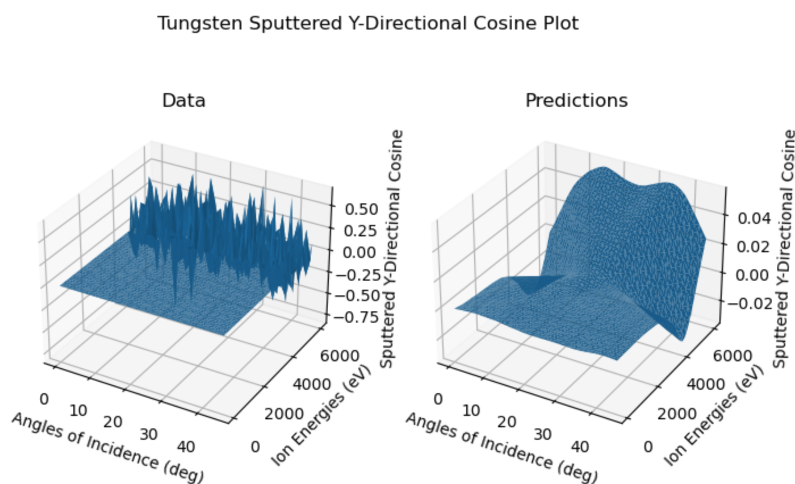

Figure 3.25: Sputtered y-directional cosine of tungsten TRIM data showing no preferential direction and model predicted values showing only very small predictions, in agreement

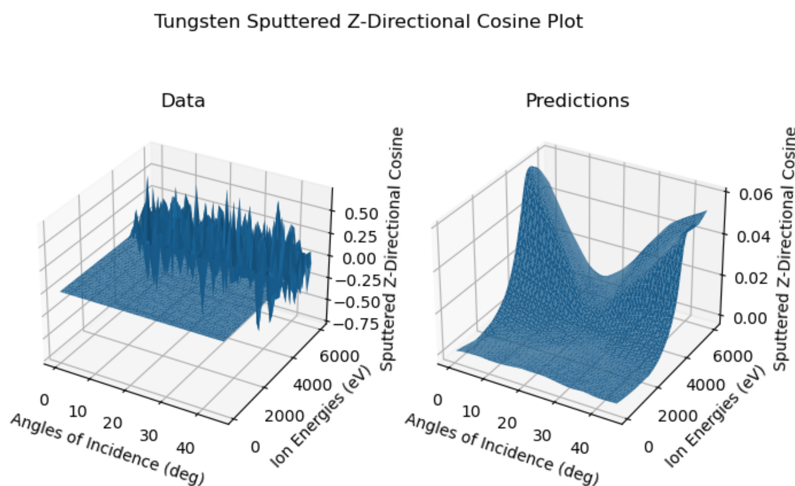

Figure 3.26: Sputtered z-directional cosine of tungsten TRIM data and model predicted values showing only very small predictions, in agreement

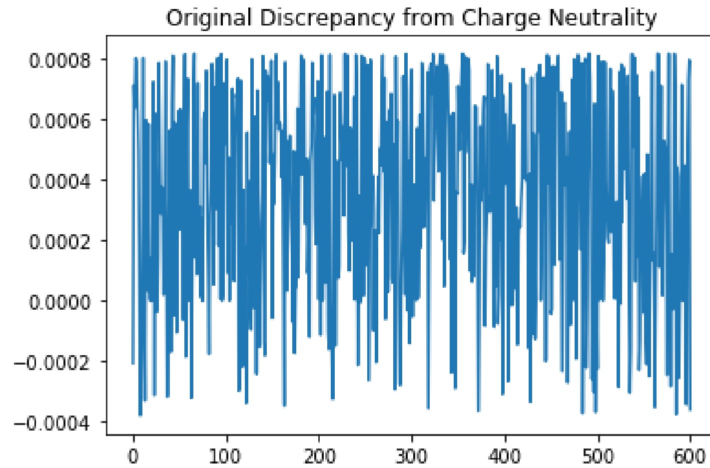

Figure 3.27: Positive and negative charge in the electron boundary condition investigation simulation to t=0.1 with the original presheath boundary conditions on the electrons

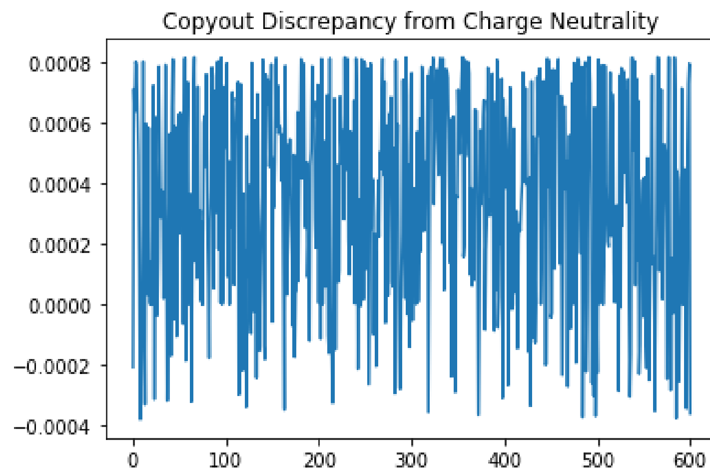

Figure 3.28: Positive and negative charge in the electron boundary condition investigation simulation to t=0.1 with zero normal gradient boundary conditions and the electrons. The similarity to Fig. 3.27 shows that zero normal gradient boundary conditions preserve charge neutrality as well as the original presheath boundary conditions over short time scales

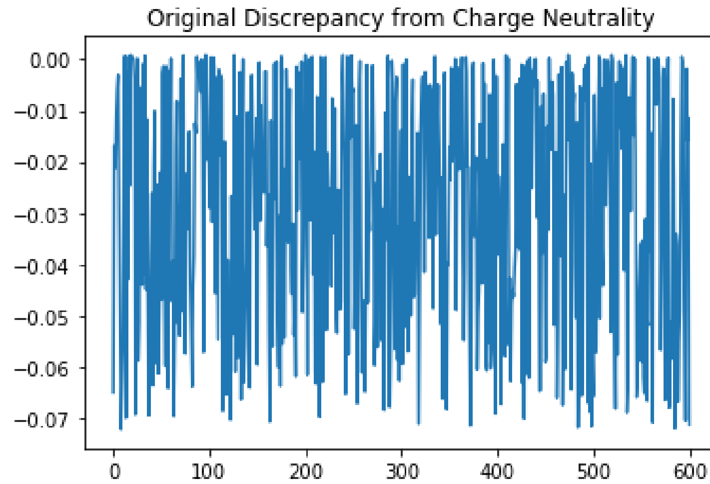

Figure 3.29: Positive and negative charge in the electron boundary condition investigation simulation to t=1 with the original presheath boundary conditions on the electrons

These results support that the zero normal gradient boundary conditions maintain charge neutrality, but for general use we would like to be sure the results hold up over longer time intervals, so we repeat the simulation up to t=1. The results of this simulation are shown in Fig. 3.29

Upon visual inspection, these figures appear identical and both maintain charge neutrality. With these results, we conclude that zero normal gradient boundary conditions on electrons are appropriate. This is likely due to the zero gradient boundary conditions making it neutrally favorable for electrons to leave the domain, allowing them to freely respond to ions that leave the domain. This rounds out the boundary condition for our model, as we now have conditions for backscattered ions, sputtered neutrals, and electrons.

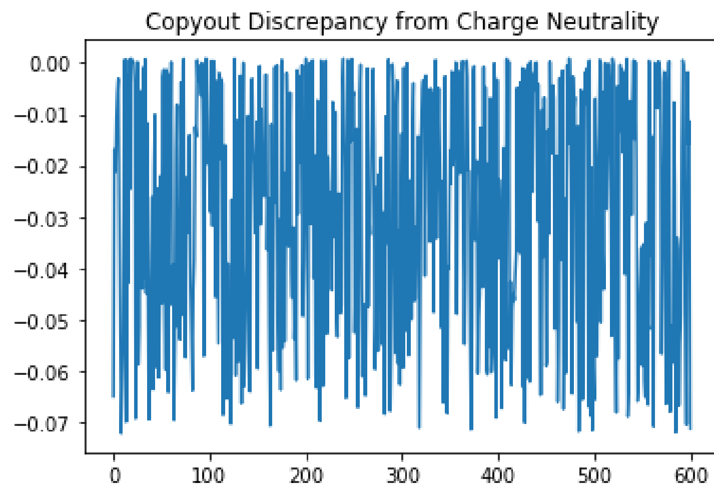

Figure 3.30: Positive and negative charge in the electron boundary condition investigation simulation to t=1 with zero normal gradient boundary conditions and the electrons. The similarity to Fig. 3.29 shows that zero normal gradient boundary conditions preserve charge neutrality as well as the original presheath boundary conditions over longer time scales

Chapter 4

CONCLUSIONS

This work achieved several meaningful results. The first of these meaningful results is the development and open release of a parser for TRIM files relevant to backscattering, sputtering, and transmission. The second result is the generation of a dataset of TRIM data for conditions relevant to a SFS Z-pinch for two different materials. The third result is the training of two machine learning models on this dataset making accurate predictions for these conditions. The final result of this work was the development of a boundary condition for WARPXM that incorporates the trained models for two different materials.

Firstly, the development and open release of a parser for TRIM files relevant to several important phenomena is significant for advancing the use of TRIM. This work has made is faster and easier to simulate ion-matter interactions for any ion-material combination through the advancement of a convenient Python interface. Previously this work would have required extensive use of TRIM's GUI, which is highly inconvenient for simulation of multiple conditions, requiring a manual restart for each condition.

The simulation of the SFS Z-pinch and generation of relevant TRIM data in the form of a usable dataset is also significant. This dataset supports the generation of models like the one generated in this work, while also serving as a reference for other projects using TRIM.

The generation of models approximating TRIM for relevant parameters is also significant. Our models also have the important feature of effectively denoising the TRIM data through effectively taking a local average through the process of loss minimization. This demonstrates that TRIM doesn't need to be run in full precision to generate models that are likely more accurate than the individual data points themselves. The completion of this process demonstrates an effective way to approximate TRIM results for parameters of inter-

est in plasma simulation, but also any of the other many and varied applications of TRIM results. In this way this work is applicable to everything from semiconductor manufacturing to spacecraft development. The models generated can be evaluated orders of magnitude faster than time it took to generate the data, making is extremely attractive and fulfilling one of the initial promises of the work.

Finally the development of a boundary condition for three fluid 5N moment plasma simulations based on the TRIM models generated demonstrates an application of the models to create a plasma simulation with enhanced accuracy. This functions as a proof of concept for a process that could be repeated for other plasma simulations of various devices to increase the accuracy of their own simulations. This process uses the generated TRIM data efficiently, creating a model that runs fast enough to be evaluated at hundreds of positions in negligible time on modern hardware.

4.1 Future Work

Future work could improve upon the methods in this work or extend this work. To extend this work one suggestion would be to simply run TRIM for more conditions and gather additional data. This additional data will expand the range of applicability of the models that could be trained on this new data. Additional data could come in the form of increasing the range of ion energy and angle of incidence, increasing the resolution used, or generating TRIM data for other materials than those used in this work. This would allow them to be used in simulations of other devices, or with enough data perhaps as general drop-in replacements for wall boundary conditions based on the materials used in generating the data. To improve this work additional data could be gathered at each point in the domain this work used. Instead of TRIM simulating 2000 ions at each combination of ion energy and angle of incidence, 3000, 4000, or even 5000 ions could be used instead. This would alleviate some of the noise in the data and make training the model easier and improve the accuracy of the model (although likely not by a huge amount as the model already smoothes out a great deal of the noise in the data). This would be relatively straightforward, with the way the

existing data has been saved, allowing for simply adding more data while retaining use of the existing data. Generally, the creation of open datasets of TRIM data for various parameters and materials would make applications of this approach easier and more convenient. Models trained on these datasets could then be openly released themselves, allowing anyone to have easy access to simulation tools for any wall material they desire. The models trained in this work take up less than 25 MB of disk space to store, so a repository of these models, are likely larger models that are applicable over a broader range of parameters is very feasible. Similarly, while the total TRIM dataset generated for carbon takes up 735 MB and the dataset for tungsten takes up 966 MB, most of this data isn't particularly relevant for this work, and isn't in a compressed form. An array containing the data used for our purposes and compressed using Python's pickle format takes up only 259 KB for carbon or tungsten. The use of the data could also be improved through acknowledgement of the fact that fluid models rely on an assumed Maxwellian function at each point in the simulation. This is currently unaccounted for in the boundary condition or TRIM data generation processes, where we assume the fluid consists of a beam of monoenergetic particles. One way to deal with this is to generate TRIM data for a more expansive range of parameters than we have in this work then sample the distribution of ion energies and angles of incidence assumed and generate predictions for all these values. These values can then be weighted according to the value of the distribution function at their location in parameter space. Alternatively, the whole process of generating TRIM data could be overhauled to account for this effect for the 5N moment model. By this we refer to reframing the data generated to directly consider the plasma properties it was generated for. The TRIM data could be generated using some incident ions with lower energies and angles of incidences and some incident ions with higher energies and angles of incidence, with the number of ions simulated for each condition determined from the value of the distribution function for those parameters. In this way, more accurate data could be generated effectively as a function of the plasma parameters as represented in the 5N moment model. Finally, we note that if the current setup is used, it may be prudent set some of the directional cosines that are approximately

zero to zero and eliminate the small amount of noise present even in the model predicted values that isn't meaningful.

Improving this work could also take the form of using more sophisticated machine learning methods to approximate the data. These models could perhaps incorporate physical understanding or known dependencies instead of working purely from the given data. Another way to extend this work would be to apply the trained model to different fusion devices. The work could also be extended by checking the results of a simulation using a trained model on actual data from the experiment being modelled by the simulation. For instance, the density of sputtered neutrals in the plasma could be measured and compared with the propagation observed in the simulation. The process of hyperparameter optimization could also be automated through existing techniques such as random search, Bayesian optimization, and the Hyperband algorithm[13]. Although the complete process would involve significant amounts of computation, the automation of the process would allow for human efforts to be directed elsewhere while the work is taking place, maximizing productivity. Another way to improve the work would be to further optimize the implementation of the boundary condition. The boundary condition currently makes predictions for each point its applied to individually but as machine learning packages are optimized for predictions on batches, the implementation could be parallelized to realize a significant speedup, saving on loading the model each time the boundary condition is evaluated. This work could also be improved by using an initial simulation that incorporates sheath boundary conditions, giving a more accurate expectation for the energy that ions would have near a physical wall. Overall, this work represents a novel approach to accurately modelling physical walls in plasma simulation and accordingly significant opportunities exist for improvement and extension.

BIBLIOGRAPHY

[1] NRT Astm et al. Standard practice for neutron radiation damage simulation by charged-particle irradiation. *Annu. B. ASTM Stand.*, 12:E521, 1996.

[2] Mosab Jaser Banisalman, Sehyeok Park, and Takuji Oda. Evaluation of the threshold displacement energy in tungsten by molecular dynamics calculations. *Journal of Nuclear Materials*, 495:277–284, 2017.

[3] SI Braginskii. Transport processes in a plasma, ma leontovich. *Reviews of Plasma Physics, Consultants Bureau, New York*, 1965.

[4] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[5] Wolfgang Eckstein. Calculated sputtering, reflection and range values. 2002.

[6] Jeffrey P Freidberg. Ideal magnetohydrodynamic theory of magnetic fusion systems. *Reviews of Modern Physics*, 54(3):801, 1982.

[7] A. Hakim, J. Loverich, and U. Shumlak. A high resolution wave propagation scheme for ideal two-fluid plasma equations. *Journal of Computational Physics*, 219(1):418–442, 2006.

[8] A. Hakim and U. Shumlak. Two-fluid physics and field-reversed configurations. *Physics of Plasmas*, 14(5):055911, 2007.

[9] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[13] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[14] J. Loverich and U. Shumlak. Nonlinear full two-fluid study of m=0 sausage instabilities in an axisymmetric z pinch. *Physics of Plasmas*, 13(8):082310, 2006.

[15] A.J. McKenna, T. Trevethan, C.D. Latham, P.J. Young, and M.I. Heggie. Threshold displacement energy and damage function in graphite from molecular dynamics. *Carbon*, 99:71–78, 2016.

[16] U. Shumlak. Z-pinch fusion. *Journal of Applied Physics*, 127(20):200901, 2020.

[17] U. Shumlak, C.S. Adams, J.M. Blakely, B.-J. Chan, R.P. Golingo, S.D. Knecht, B.A. Nelson, R.J. Oberto, M.R. Sybouts, and G.V. Vogman. Equilibrium, flow shear and stability measurements in the z-pinch. *Nuclear Fusion*, 49(7):075039, jul 2009.

[18] U Shumlak, J Chadney, RP Golingo, DJ Den Hartog, MC Hughes, SD Knecht, W Lowrie, VS Lukin, BA Nelson, RJ Oberto, et al. The sheared-flow stabilized z-pinch. *Fusion Science and Technology*, 61(1T):119–124, 2012.

[19] U Shumlak, RP Golingo, BA Nelson, and DJ Den Hartog. Evidence of stabilization in the z-pinch. *Physical review letters*, 87(20):205005, 2001.

[20] U. Shumlak, R. Lilly, N. Reddell, E. Sousa, and B. Srinivasan. Advanced physics calculations using a multi-fluid plasma model. *Computer Physics Communications*, 182(9):1767–1770, 2011. Computer Physics Communications Special Edition for Conference on Computational Physics Trondheim, Norway, June 23-26, 2010.

[21] U. Shumlak and J. Loverich. Approximate riemann solver for the two-fluid plasma model. *Journal of Computational Physics*, 187(2):620–638, 2003.

[22] U. Shumlak, B. A. Nelson, E. L. Claveau, E. G. Forbes, R. P. Golingo, M. C. Hughes, R. J. Oberto, M. P. Ross, and T. R. Weber. Increasing plasma parameters using sheared flow stabilization of a z-pinch. *Physics of Plasmas*, 24(5):055702, 2017.

[23] Xue Yang and Ahmed Hassanein. Atomic scale calculations of tungsten surface binding energy and beryllium-induced tungsten sputtering. *Applied Surface Science*, 293:187–190, 2014.

[24] James F. Ziegler and Jochen P. Biersack. *The Stopping and Range of Ions in Matter*, pages 93–129. Springer US, Boston, MA, 1985.

Appendix A
TRIM DATA COLLECTION NOTEBOOK

The programming for this work took place partially in a Jupyter notebook for convenience. The Jupyter notebook used to generate TRIM data for a given range of energies and angles of incidence is given in the following pages.

Data Collection

This notebook handles the collection and processing of data using TRIM

Imports

In [ ]:
```
import numpy as np
from pathlib import Path
import pickle
from shutil import rmtree
from sys import exit, path
import time

path.append('..')
path.append('../PySRIM/pysrim/') # The version of pysrim used here can be found at https://github.com/SimonF24/python-srim-implementation

from local_secrets import SRIM_2008_dir, SRIM_2013_dir, TRIM_output_dir
from PySRIM.pysrim import srim
```

Calculation Parameters

In [ ]:
```
aoir = [44, 45] # angle of incidence range [start end] in degrees
delete_old_TRIM_files = False
ier = [100, 6000] # ion energy range [start end] in eV
ions_per_calculation = 2000
ions_per_fragment = 1000
num_aoi_values = 2 # angle of incidence number of values
num_ier_values = 60 # ion energy range number of values
SRIM_to_use = '2013'
target_material = 'W' # Either 'W' or 'C' for Tungsten or Carbon (graphite) respectively

angles = np.linspace(aoir[0], aoir[1], num_aoi_values)
ion_energies = np.linspace(ier[0], ier[1], num_ier_values)
print('Angles:', angles)
print('Ion Energies:', ion_energies)
```

Helper Functions

In [ ]:
```
if SRIM_to_use == '2008':
    SRIM_dir = SRIM_2008_dir
elif SRIM_to_use == '2013':
    SRIM_dir = SRIM_2013_dir

def clear_output_directories():
    '''
    Clears the output directories of the calculation if they exist (after user confirmation)
    '''
    if Path(TRIM_output_dir).is_dir():
        response = input('Are you sure you want to delete old TRIM files? Enter y or n:')
        if response == ('y' or 'Y'):
            rmtree(TRIM_output_dir)
        else:
            exit()

# Modified from https://gitlab.com/costrouc/pysrim/blob/master/examples/notebooks/SiC.ipynb

def run_fragmented_calculation(ion, target, number_ions, ion_energy, trim_settings, step):
    '''
    step is the number of ions per step
    '''
    fragment = 1
    ions_remaining = number_ions
    while ions_remaining > 0:
        if step > ions_remaining:
            step = ions_remaining
        print('Total ions completed: {:06d}\tion: {}\tions in step: {:06d}'.format(number_ions-ions_remaining, ion.symbol, step))
        trim = srim.TRIM(target, ion, calculation=3, number_ions=step, **trim_settings)
        # Calculation=3 is a sputtering calculation, which is what we are interested in
        # Full details at https://gitlab.com/costrouc/pysrim/-/blob/master/srim/srim.py
        trim.run(SRIM_dir)
        output_file_dir = f'{TRIM_output_dir}/{trim_settings["angle_ions"]}/{ion_energy}/{fragment}'
        Path(output_file_dir).mkdir(exist_ok=True, parents=True)
        srim.TRIM.copy_output_files(SRIM_dir, output_file_dir)
        ions_remaining -= step
        fragment += 1
```

Running the Calculations

In [ ]:
```
total_start_time = time.perf_counter()
trim_calculation_times = []

# Defining the target layer

if target_material == 'W':
    target_layer = srim.Layer({
        'W': {
            'E_d': 90.0, # displacement energy in eV (recommended value from ASTM, found here: https://doi.org/10.1016/j.jnucmat.2017.08.019)
            'lattice': 0.0, # lattice binding energy in eV (recommended value from ASTM, found here: https://doi.org/10.1088/1741-4326/ab9a65)
            # Article references the recommendation here:
            # Astm, N. R. T. "Standard practice for neutron radiation damage simulation by charged-particle irradiation." Annu. B. ASTM Stand. 12 (1996): E521.
            'stoich': 1.0, # stoichiometric ratio of Tungsten in the layer
            'surface': 11.75 # surface binding energy in eV (value from https://doi.org/10.1016/j.apsusc.2013.12.129)
            # TRIM's default value for the surface binding energy is 8.68 eV, which is the heat of sublimation,
            # the paper discusses why their estimate is better
            # ASTM paper referenced above:
            # Standard practice for neutron radiation damage simulation by charge-particle irradiation 2009
            # Technical report ASTM E521-96(2009)e2 ASTM Int., West Conshohocken, PA
        }},
        density=19.35, # g/cm^3 (default value in TRIM, standard density of Tungsten at room temperature)
        width=1e9 # angstroms (0.1 m, essentially infinitely thick)
    )
elif target_material == 'C':
    target_layer = srim.Layer({
        'C': {
            'E_d': 25.0, # displacement energy in eV (room temperature value from here: https://doi.org/10.1016/j.carbon.2015.11.040)
            # They also note 30 eV as the value for temperatures of 900 K
            'lattice': 3.0, # lattice binding energy in eV (default value in TRIM)
            'stoich': 1.0, # stoichiometric ratio of carbon in the layer
```

```
            'surface': 7.41, # surface binding energy in eV (default value in TRIM, presumably the heat of sublimation)
        }},
        density=2.26, # g/cm^3
        width=1e9 # angstroms (0.1 m, essentially infinitely thick)
    )

target = srim.Target([target_layer])

if delete_old_TRIM_files:
    clear_output_directories()

for angle in angles:
    trim_settings = {
        'angle_ions': angle,
        'backscattered': True,
        'ranges': True,
        'sputtered': True,
        'transmit': True
    } # All options listed here: https://gitlab.com/costrouc/pysrim/-/blob/master/srim/srim.py
    for ion_energy in ion_energies:

        start_time = time.perf_counter()

        ion = srim.Ion('H', energy=ion_energy)

        print(f'Angle of Incidence: {angle}, Ion Energy: {ion_energy}')
        run_fragmented_calculation(ion, target, ions_per_calculation, ion_energy, trim_settings, ions_per_fragment)

        end_time = time.perf_counter()
        time_elapsed = end_time-start_time
        print(f'Calculation Time: {time_elapsed} s')
        trim_calculation_times.append(time_elapsed)

    print(f'Average Calculation Time: {np.mean(trim_calculation_times)}')

total_end_time = time.perf_counter()
total_calculation_time = total_end_time - total_start_time
print(f'Total Calculation Time: {total_calculation_time}')
```

Processing the Data

We here compute the reflection and sputtering coefficients from the TRIM output files. We separate this step from the previous step so this analysis can be run on saved data. Note that the data ends up in a bit of a weird order since iterdir alphabetizes the files, which in this case means that 1000 is after 100 and the like, i.e. the data isn't in numerical order. The form of the output data array is [incident_angle, incident_ion_energy, backscat_coef, backscat_energy, backscat_cosx, backscat_cosy, backscat_cosz, sput_yield, sput_energy, sput_cosx, sput_cosy, sput_cosz]. The energies are in eV and the x-direction is the direction into the wall.

In [ ]:

```
output_file_dir = TRIM_output_dir # This can also be hardcoded to something else if desired

angles = [f.parts[-1] for f in Path(output_file_dir).iterdir() if f.is_dir()]
counter = 0
ion_energies = [f.parts[-1] for f in next(Path(output_file_dir).iterdir()).iterdir() if f.is_dir()]

data = np.zeros((len(angles)*len(ion_energies), 12))
for angle in angles:
    for ion_energy in ion_energies:

        # Discovering the folders corresponding to fragments then combining the
        # results from each of them

        fragment_folders = [f for f in Path(f'{output_file_dir}\\{angle}\\{ion_energy}').iterdir() if f.is_dir()]

        backscat_ions = 0
        backscat_energy = 0
        backscat_cosx = 0
        backscat_cosy = 0
        backscat_cosz = 0
        counter2 = 0
        sput_ions = 0
        sput_energy = 0
        sput_cosx = 0
        sput_cosy = 0
        sput_cosz = 0
        total_ions = 0

        for folder in fragment_folders:

            counter2 += 1

            results = srim.output.Results(folder)

            total_ions += results.ioniz.num_ions

            trimout = results.trimout

            if trimout == None:
                # Some conditions have no sputtering or backscattering
                # TRIM occasionally errors so some folders don't generate properly
                # Skipped folders are still accounted for by 0's for the data
                print(f'Skipped folder {folder}')
                continue

            # We calculate the below values as running averages since they must be incremented for every fragment
            # We need the if statements to avoid numpy giving nans as the average of an empty array
            # Note that if the if statements evaluate to False the value is left as 0
            if (trimout.interaction_type == 'B').any(): # Checking if any particles were backscattered
                backscat_indices = trimout.interaction_type == 'B'
                backscat_ions += np.sum(backscat_indices)
                backscat_fragment_mean_energy = np.mean(trimout.energy[backscat_indices])
                backscat_energy += (backscat_fragment_mean_energy-backscat_energy)/counter2
                backscat_fragment_cosx = np.mean(trimout.atom_dir_x[backscat_indices])
                backscat_cosx += (backscat_fragment_cosx-backscat_cosx)/counter2
                backscat_fragment_cosy = np.mean(trimout.atom_dir_y[backscat_indices])
                backscat_cosy += (backscat_fragment_cosy-backscat_cosy)/counter2
                backscat_fragment_cosz = np.mean(trimout.atom_dir_z[backscat_indices])
                backscat_cosz += (backscat_fragment_cosz-backscat_cosz)/counter2
            if (trimout.interaction_type == 'S').any(): # Checking if any particles were sputtered
                sput_indices = trimout.interaction_type == 'S'
                sput_ions += np.sum(sput_indices)
                sput_fragment_mean_energy = np.mean(trimout.energy[sput_indices])
                sput_energy += (sput_fragment_mean_energy-sput_energy)/counter2
                sput_fragment_cosx = np.mean(trimout.atom_dir_x[sput_indices])
                sput_cosx += (sput_fragment_cosx-sput_cosx)/counter2
                sput_fragment_cosy = np.mean(trimout.atom_dir_y[sput_indices])
                sput_cosy += (sput_fragment_cosy-sput_cosy)/counter2
                sput_fragment_cosz = np.mean(trimout.atom_dir_z[sput_indices])
                sput_cosz += (sput_fragment_cosz-sput_cosz)/counter2

        backscat_coef = backscat_ions/total_ions
```

```
        sput_yield = sput_ions/total_ions

        data[counter, :] = [angle, ion_energy, backscat_coef, backscat_energy, backscat_cosx, backscat_cosy, backscat_cosz, sput_yield, sput_energy, sput_cosx, sput_cosy, sput_cosz]

        counter += 1
```

Saving the Data

This may overwrite the old data file, be sure you are ready to do this

In [ ]:
```
with open('data.pickle', 'wb') as f:
    pickle.dump(data, f)
```

Appendix B

MODEL CREATION NOTEBOOK

The Jupyter notebook used to train our machine learning models is included in the following pages.

Generating an ML Model

Imports

In [ ]:
```
%matplotlib
import pickle
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import time

matplotlib.style.use('default')
```

Helper Functions

In [ ]:
```
data_col_to_name = [
    'Incident Angle',
    'Incident Energy',
    'Backscattering Coefficient',
    'Backscattered Energy',
    'Backscattered X-Directional Cosine',
    'Backscattered Y-Directional Cosine',
    'Backscattered Z-Directional Cosine',
    'Sputtering Yield',
    'Sputtered Energy',
    'Sputtered X-Directional Cosine',
    'Sputtered Y-Directional Cosine',
    'Sputtered Z-Directional Cosine'
]

def unnormalize(data, data_col, data_norms):
    return data[:, data_col]*data_norms[data_col][1]+data_norms[data_col][0]

def unnormalize_preds(preds, data_col, data_norms):
    return preds[:, data_col-2]*data_norms[data_col][1]+data_norms[data_col][0]

def plot_data(data, data_col, material):
    name = data_col_to_name[data_col]
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.plot_trisurf(data[:, 0], data[:, 1], data[:, data_col])
    plt.title(f'{material} {name} Plot')
    plt.xlabel('Angles of Incidence (deg)')
    plt.ylabel('Ion Energies (eV)')
    ax.set_zlabel(f'{name}')
    plt.show()

def plot_normalized_data(data, data_col, data_norms, material):
    '''
    Data is assumed to be normalized
    '''
    name = data_col_to_name[data_col]
    angles = unnormalize(data, 0, data_norms)
    energies = unnormalize(data, 1, data_norms)
    values = unnormalize(data, data_col, data_norms)
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.plot_trisurf(angles, energies, values)
    plt.title(f'{material} {name} Plot')
    plt.xlabel('Angles of Incidence (deg)')
    plt.ylabel('Ion Energies (eV)')
    ax.set_zlabel(f'{name}')
    plt.show()

def plot_predictions(data, data_col, data_norms, material, model_output):
    '''
    Data is assumed to be normalized
    '''
    if data_col <= 2:
        raise ValueError("That data_col doesn't correspond to a model prediction")
    name = data_col_to_name[data_col]
    angles = unnormalize(data, 0, data_norms)
    energies = unnormalize(data, 1, data_norms)
    values = unnormalize_preds(model_output, data_col, data_norms)
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.plot_trisurf(angles, energies, values)
    plt.title(f'Predicted {material} {name} Plot')
    plt.xlabel('Angles of Incidence (deg)')
    plt.ylabel('Ion Energies (eV)')
    ax.set_zlabel(f'{name}')
    plt.show()

def compare_data_and_prediction(data, data_col, data_norms, model, material):
    '''
    The data is assumed to be normalized since it must be to get predictions from the model
    '''
    if data_col < 2:
        raise ValueError("That data_col doesn't correspond to a model prediction")
    name = data_col_to_name[data_col]
    model_output = model.predict(data[:, 0:2])
    preds = unnormalize_preds(model_output, data_col, data_norms)
    angles = unnormalize(data, 0, data_norms)
    energies = unnormalize(data, 1, data_norms)
    unnormalized_data = unnormalize(data, data_col, data_norms)
    fig = plt.figure()
    plt.suptitle(f'{material} {name} Plot')
    ax1 = plt.subplot(1, 2, 1, projection='3d')
    ax1.plot_trisurf(angles, energies, unnormalized_data)
    plt.title(f'Data')
```

```
    plt.xlabel('Angles of Incidence (deg)')
    plt.ylabel('Ion Energies (eV)')
    ax1.set_zlabel(f'{name}')
    ax2 = plt.subplot(1, 2, 2, projection='3d')
    ax2.plot_trisurf(angles, energies, preds)
    plt.title(f'Predictions')
    plt.xlabel('Angles of Incidence (deg)')
    plt.ylabel('Ion Energies (eV)')
    ax2.set_zlabel(f'{name}')
    plt.show()

def normalize(data, data_name=None, data_units=None, print_stats=True):
    '''
    Returns the same data with mean set to 0 and variance set to 1.
    Also prints the mean and standard deviation that were used to normalize the data
    '''
    data_mean = np.mean(data)
    data_std = np.std(data)
    if print_stats:
        if data_name and data_units:
            print(f'The mean {data_name} was {data_mean} {data_units}')
            print(f'The standard deviation of {data_name} was {data_std} {data_units}')
        elif data_name:
            print(f'The mean {data_name} was {data_mean}')
            print(f'The standard deviation of {data_name} was {data_std}')
        else:
            print(f'The mean was {data_mean}')
            print(f'The standard deviation was {data_std}')
    normalized_data = (data-data_mean)/data_std
    return normalized_data
```

Data Exploration

The form of the data is an array with [incident_angle, incident_ion_energy, backscat_coef, backscat_energy, backscat_cosx, backscat_cosy, backscat_cosz, sput_yield, sput_energy, sput_cosx, sput_cosy, sput_cosz]

In [ ]:
```
cols_to_plot = range(12)
normalized = False
plots_to_generate = ('C', 'W') # ('C'), ('W'), or ('C', 'W')

if 'W' in plots_to_generate:
    # Plot a surface of the Tungsten sputtering yield

    with open('tungsten.pickle', 'rb') as f:
        w_data = pickle.load(f)

    if normalized:
        for col in cols_to_plot:
            w_data[:, col] = normalize(w_data[:, col], print_stats=False)
            plot_data(w_data, col, 'Tungsten')
    else:
        for col in cols_to_plot:
            plot_data(w_data, col, 'Tungsten')

if 'C' in plots_to_generate:
    # Plot a surface of the Carbon sputtering yield

    with open('carbon.pickle', 'rb') as f:
        c_data = pickle.load(f)

    if normalized:
        for col in cols_to_plot:
            c_data[:, col] = normalize(c_data[:, col], print_stats=False)
            plot_data(c_data, col, 'Carbon')
    else:
        for col in cols_to_plot:
            plot_data(c_data, col, 'Carbon')
```

Importing and Processing Data

In [ ]:
```
model_to_train = 'carbon' # 'carbon' or 'tungsten'

batch_size = 32
split = (0.7, 0.2, 0.1) # (train, test, validation) split

if model_to_train == 'tungsten':
    with open('tungsten.pickle', 'rb') as f:
        data = pickle.load(f)
elif model_to_train == 'carbon':
    with open('carbon.pickle', 'rb') as f:
        data = pickle.load(f)
else:
    raise ValueError("Please set model_to_train to either 'tungsten' or 'carbon'")

# Data Augmentation

# We normalize the data to have mean 0 and variance 1
# We output the mean and variance so we can scale future inputs
# by the same factors and de-scale the output

data[:, 0] = normalize(data[:, 0], 'ion angle', 'degrees')
data[:, 1] = normalize(data[:, 1], 'ion energy', 'eV')
data[:, 2] = normalize(data[:, 2], 'backscattering coefficient')
data[:, 3] = normalize(data[:, 3], 'backscattered energy', 'eV')
data[:, 4] = normalize(data[:, 4], 'backscattered x-directional cosine')
data[:, 5] = normalize(data[:, 5], 'backscattered y-directional cosine')
data[:, 6] = normalize(data[:, 6], 'backscatterd z-directional cosine')
data[:, 7] = normalize(data[:, 7], 'sputtering yield')
data[:, 8] = normalize(data[:, 8], 'sputtered energy', 'eV')
data[:, 9] = normalize(data[:, 9], 'sputtered x-directional cosine')
data[:, 10] = normalize(data[:, 10], 'sputtered y-directional cosine')
data[:, 11] = normalize(data[:, 11], 'sputtered z-directional cosine')

features = data[:, 0:2] # [incident_angle, incident_ion_energy]
```

```
labels = data[:, 2:12] # [backscat_coef, backscat_energy, backscat_cosx, backscat_cosy, backscat_cosz, sput_yield, sput_energy, sput_cosx, sput_cosy, sput_cosz]

cardinality = len(features)

AUTOTUNE = tf.data.AUTOTUNE

num_test_examples = int(cardinality*split[0])
num_train_examples = int(cardinality*split[1])
num_val_examples = int(cardinality*split[2])

ds = tf.data.Dataset.from_tensor_slices((features, labels))
ds = ds.shuffle(cardinality, reshuffle_each_iteration=False)

train_ds = ds.take(num_train_examples)
test_ds = ds.skip(num_train_examples)
val_ds = test_ds.skip(num_val_examples)
test_ds = test_ds.take(num_test_examples)

test_ds = test_ds.batch(batch_size)
test_ds = test_ds.prefetch(AUTOTUNE)

train_ds = train_ds.batch(batch_size)
train_ds = train_ds.prefetch(AUTOTUNE)

val_ds = val_ds.batch(batch_size)
val_ds = val_ds.prefetch(AUTOTUNE)
```

Defining the values output above

```
In [ ]: # The form of this is [mean, std] with rows corresponding to the columns of the data
carbon_data_norms = [
    [22.5, 13.275918047351754], # incident angle
    [3050, 1731.8102282486573], # incident energy
    [0.0637374650709028, 0.059122076487240244], # backscattering coefficient
    [763.2909999604867, 405.42439533698723], # backscattered energy
    [-0.6720263283561208, 0.030189154307014226], # backscattered x-directional cosine
    [0.10108687661261459, 0.08543859178735679], # backscattered y-directional cosine
    [-0.0003458549795388853, 0.06206727404039759], # backscattered z-directional cosine
    [0.013625399771014608, 0.007278795054l131985], # sputtering yield
    [52.61701362624996, 38.93298225740309], # sputtered energy
    [-0.880096628438116, 0.056375343939504094], # sputtered x-directional cosine
    [0.001456343846170728, 0.0772584678868553], # sputtered y-directional cosine
    [0.0011856608447239076, 0.07991989284716235] # sputtered z-directional cosine
]
tungsten_data_norms = [
    [22.5, 13.275918047351754], # incident angle
    [3050, 1731.8102282486573], # incident energy
    [0.39911483465407466, 0.07198504463507843], # backscattering coefficient
    [1513.2032466996836, 801.1452931415333], # backscattered energy
    [-0.6830923809737315, 0.010854350746685008], # backscattered x-directional cosine
    [0.04131061349486775, 0.03401559337626312], # backscattered y-directional cosine
    [-3.292192014994721e-05, 0.017218019987532664], # backscattered z-directional cosine
    [0.0005017969454254625, 0.0010674496442855464], # sputtering yield
    [9.470058622811253, 18.168424385324865], # sputtered energy
    [-0.19088631264203892, 0.34131562415961886], # sputtered x-directional cosine
    [0.003961399387121083, 0.10428122401228], # sputtered y-directional cosine
    [0.0003989868887320404, 0.10726489670744432] # sputtered z-directional cosine
]
```

Creating the Model

```
In [ ]: early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    mode='min',
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

# Best single output model for unnormalized data
# model = tf.keras.Sequential([
#     tf.keras.layers.Dense(2048, activation='sigmoid', input_shape=(2,)),
#     tf.keras.layers.Dense(2048, activation='relu'),
#     tf.keras.layers.Dense(1)
# ])

# Best single output model for normalized data
# learning_rate = 2e-4
# model = tf.keras.Sequential([
#     tf.keras.layers.Dense(2048, activation='relu', input_shape=(2,)),
#     tf.keras.layers.Dense(2048, activation='relu'),
#     tf.keras.layers.Dense(1)
# ])

# The single output models above were trained on just the sputtering yield

# Best multi-output model for carbon and tungsten with normalized data
# learning_rate = 1e-4
# model = tf.keras.Sequential([
#     tf.keras.layers.Dense(2048, activation='relu', input_shape=(2,)),
#     tf.keras.layers.Dense(1024, activation='relu'),
#     tf.keras.layers.Dense(10),
# ])


learning_rate = 1e-4
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2048, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(10),
])
model.compile(
    loss=tf.keras.losses.MeanSquaredError(),
    metrics=[tf.keras.metrics.MeanAbsoluteError()],
    optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate)
)
model.summary()
```

Training the Model

```
history = model.fit(
    train_ds,
    callbacks=[early_stopping_callback],
    epochs=1000,
    validation_data=val_ds
)
```

Plotting Training Metrics

```
# Plotting Training Metrics

plt.figure()
plt.plot(history.history['mean_absolute_error'], label='Training MAE')
plt.plot(history.history['val_mean_absolute_error'], label='Validation MAE')
plt.title('Mean Absolute Error')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.legend()
plt.show()
plt.figure()
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss')
plt.legend()
plt.show()
```

Visualizing Model Predictions

In this section we should keep in mind that the majority of particles have angles of incidence very close to 0 so accuracy there is more important than elsewhere

```
compare_all = True

if model_to_train == 'carbon':
    data_norms = carbon_data_norms
elif model_to_train == 'tungsten':
    data_norms = tungsten_data_norms
else:
    raise ValueError('Please set model_to_train to either "carbon" or "tungsten"')

if compare_all:
    for data_col in range(2, 12):
        compare_data_and_prediction(data, data_col, data_norms, model, model_to_train.capitalize())
else:
    data_cols = [7]
    for data_col in data_cols:
        compare_data_and_prediction(data, data_col, data_norms, model, model_to_train.capitalize())
```

Checking Model Accuracy on the Test Set

Remember not to overuse this, the test set should ideally only be used once so as to not become a de facto part of the validation set (used to dictate training)

```
metrics = model.evaluate(test_ds, return_dict=True)
print(f'The loss of the model on the test set was {metrics["loss"]}')
print(f'The mean absolute error of the model on the test set was {metrics["mean_absolute_error"]}')
```

Saving the Model

```
model.save(f'{model_to_train}_model')
```

Loading the Model

This section exists for generating plots or checking metrics with the saved model using the above code blocks

```
model_to_load = 'carbon' # 'carbon' or 'tungsten'

if model_to_load == 'carbon' or 'tungsten':
    model = tf.keras.models.load_model(f'{model_to_load}_model')
else:
    raise ValueError('Please set model_to_load to "carbon" or "tungsten"')
```

Timing a single model prediction

```
start = time.perf_counter()
result = model.predict(data[0:1, 0:2])
end = time.perf_counter()
print(f'The calculation took {end-start} seconds')
```