Detecting Fast Neutrino Flavor Conversions with Machine Learning

Mini Project - B.Tech 6th Semester

Submitted by:

Rishabh Pandey
Roll No: 220121050

Under the Guidance of:

Prof. Sovan Chakraborty

April 30, 2025

Contents

1 Introduction

Core-collapse supernovae (CCSNe) and neutron star mergers (NSMs) are among the most violent and energetic astrophysical phenomena observed in the universe. A significant hallmark of these events is the tremendous outflow of neutrinos, which not only carry away a majority of the gravitational binding energy but also heavily influence the dynamics and nucleosynthesis of the surrounding environment.

Neutrinos, being weakly interacting particles, undergo intricate flavor oscillations as they propagate through dense astrophysical media. While the standard vacuum and matter-enhanced oscillations (MSW effect) have been well understood, a more subtle and faster mechanism, known as **fast flavor conversions (FFCs)**, has gained significant attention recently. These conversions occur at extremely small spatial and temporal scales and can significantly alter the neutrino flavor composition, impacting explosion dynamics and element formation.

Fast flavor conversions are driven not by vacuum oscillations but by the angular structure of the neutrino distributions themselves. Their occurrence depends sensitively on the presence of angular crossings in the electron lepton number (ELN) distribution, making their detection both crucial and challenging.

2 Research Background

Understanding neutrino flavor evolution in CCSNe is essential, but fully resolving the neutrino distribution function in simulations is computationally infeasible. Instead, most state-of-the-art CCSN simulations employ a moment-based approach where only the lower-order moments (like number density and flux) of the neutrino distribution are evolved.

The detection of fast flavor instabilities traditionally requires detailed knowledge of the neutrino angular distributions. However, this is often unavailable in practical simulations. To overcome this, researchers have proposed using partial information, specifically the angular moments, to infer the likelihood of fast instabilities.

Recent advances in Machine Learning (ML) techniques offer a promising route to bridge this gap. By training models on simulation data that have full angular information, it is possible to predict the occurrence of FFCs using only limited moment-based inputs. This approach can enable fast and accurate detection of flavor conversions without the need for computationally intensive full Boltzmann transport solvers.

In this project, we explore the application of ML models to the problem of FFC detection, aiming to establish a reliable and efficient framework based on CCSN moment data.

3 Problem Statement

The primary challenges addressed in this project are:

- Developing Machine Learning models capable of detecting the occurrence of fast flavor conversions using only moment-based information from CCSN simulations.
- Training and validating the models on realistic CCSN datasets where full neutrino angular information is available.
- Extending the detection capability to include cases where heavy-lepton neutrino species $(\nu_x, \bar{\nu}_x)$ have distinct distributions, moving beyond the standard ν_e and $\bar{\nu}_e$ channels.

Achieving these objectives will significantly improve the ability to predict neutrino flavor evolution in supernova models, enhancing the understanding of astrophysical processes such as nucleosynthesis and explosion dynamics.

4 Theoretical Framework

4.1 Fast Flavor Conversion Criterion

Fast flavor conversions are triggered when the angular distribution of the neutrino electron lepton number (ELN) shows a crossing — a point where the distribution changes sign. The relevant physical quantity is:

$$G(\mathbf{v}) = \sqrt{2}G_F \int_0^\infty \frac{E_\nu^2\, dE_\nu}{(2\pi)^3} \left[(f_{\nu_e}(\mathbf{p}) - f_{\nu_x}(\mathbf{p})) - (f_{\bar{\nu}_e}(\mathbf{p}) - f_{\bar{\nu}_x}(\mathbf{p}))\right]$$

Here, f_ν represents the neutrino distribution function, G_F is the Fermi constant, and $\mathbf{v}$ is the neutrino velocity.

The occurrence of a zero-crossing in $G(\mathbf{v})$ indicates the possibility of fast modes developing, leading to rapid flavor conversion.

4.2 Moment Definitions

Given the computational challenges in resolving full distribution functions, we define angular moments of the neutrino distributions:

$$I_n = \int_{-1}^{1} d\mu\, \mu^n \int_0^\infty \int_0^{2\pi} \frac{E_\nu^2\, dE_\nu\, d\phi_\nu}{(2\pi)^3} f_\nu(\mathbf{p})$$

where μ is the cosine of the angle between the neutrino momentum and the radial direction.

In practice, only the first few moments ($n = 0, 1$) are available in simulations:

- I_0: Neutrino number density
- I_1: Neutrino flux

4.3 Derived Features for Machine Learning

From these moments, we define normalized quantities used as features for ML models:

$$\alpha = \frac{I_0^{\bar{\nu}_e}}{I_0^{\nu_e}}, \quad F_{\nu_e} = \frac{I_1^{\nu_e}}{I_0^{\nu_e}}, \quad F_{\bar{\nu}_e} = \frac{I_1^{\bar{\nu}_e}}{I_0^{\bar{\nu}_e}}$$

Here:

- α captures the relative number density of antineutrinos to neutrinos.
- F represents the "flux factor," giving a measure of anisotropy in the distribution.

Higher-order features are engineered by combining these basic quantities, for example through polynomial expansions, to capture non-linear dependencies relevant to fast flavor conversion detection.

4.4 Evaluation Metrics

Model performance is evaluated based on:

$$\begin{aligned}
\text{Accuracy} &= \frac{TP + TN}{TP + TN + FP + FN} \\
\text{Precision} &= \frac{TP}{TP + FP} \\
\text{Recall} &= \frac{TP}{TP + FN} \\
F1 &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\end{aligned}$$

where:

- TP: True Positives (correctly predicted crossings)
- TN: True Negatives (correctly predicted non-crossings)
- FP: False Positives
- FN: False Negatives

5 Implementation and Code

5.1 Libraries Used

- pandas, numpy: For data manipulation and numerical operations
- scikit-learn: For implementing machine learning models and preprocessing
- matplotlib: For visualization and graphical analysis

5.2 Data Preprocessing

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler

# Load Data
df = pd.read_csv('data_XLN.csv')

# Features and Labels
X = df[['alpha_nue', 'alpha_nux', 'alpha_nubx', 'Fnue', 'Fnubx
    ', 'Fnux', 'Fnubx']].values
y = df['label'].values

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Polynomial Features
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X_scaled)

# Split
X_train, X_test, y_train, y_test = train_test_split(X_poly, y,
    test_size=0.2, random_state=42)
```

6 Machine Learning Algorithms and Results

6.1 Logistic Regression

Algorithm Description:
Logistic Regression is a linear classification algorithm that models the probability of a binary outcome using the logistic function. It estimates the probability that a given input belongs to a particular category. Despite its simplicity, it's effective for linearly separable data and provides easily interpretable coefficients that can yield insights into feature importance.

Code:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report,
   accuracy_score

# Initialize and train the model
clf_lr = LogisticRegression(max_iter=10000, C=1.0)
clf_lr.fit(X_train, y_train)

# Predict and evaluate
y_pred_lr = clf_lr.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred_lr):.2f}")
print(classification_report(y_test, y_pred_lr))

# Feature importance
coefficients = clf_lr.coef_[0]
feature_names = poly.get_feature_names_out()
importance = np.abs(coefficients)
indices = np.argsort(importance)[::-1]
print("Top 5 important features:")
for i in range(5):
    print(f"{feature_names[indices[i]]}: {coefficients[indices
       [i]]:.4f}")
```

Results:

Class	Precision	Recall	F1-Score
No Crossing	0.87	0.89	0.88
Crossing	0.88	0.86	0.87

Accuracy: **88%**

6.2 k-Nearest Neighbors

Algorithm Description:
k-Nearest Neighbors (k-NN) is a non-parametric, instance-based learning algorithm that classifies new data points based on the majority class of their k nearest neighbors in the feature space. It's intuitive and effective for data with complex decision boundaries. The algorithm doesn't explicitly learn a model but instead stores the training data and makes predictions based on similarity metrics.

Code:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report,
    accuracy_score

# Initialize and train the model
clf_knn = KNeighborsClassifier(n_neighbors=3, weights='
    distance')
clf_knn.fit(X_train, y_train)

# Predict and evaluate
y_pred_knn = clf_knn.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.2f}")
print(classification_report(y_test, y_pred_knn))

# Hyper-parameter tuning (k value)
k_values = range(1, 10)
accuracies = []
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracies.append(accuracy_score(y_test, y_pred))

best_k = k_values[np.argmax(accuracies)]
print(f"Best k value: {best_k} with accuracy: {max(accuracies)
    :.2f}")
```

Results:

Class	Precision	Recall	F1-Score
No Crossing	0.89	0.88	0.89
Crossing	0.88	0.89	0.88

Accuracy: **88%**

6.3 Support Vector Machine

Algorithm Description:
Support Vector Machine (SVM) finds a hyperplane that optimally separates classes in feature space. The algorithm maximizes the margin between classes, making it robust to overfitting. The radial basis function (RBF) kernel allows SVM to model non-linear decision boundaries by implicitly mapping the data to a higher-dimensional space where it becomes linearly separable.

Code:

```
from sklearn.svm import SVC
from sklearn.metrics import classification_report,
    accuracy_score

# Initialize and train the model
clf_svm = SVC(kernel='rbf', gamma='auto', C=1.0, probability=
    True)
clf_svm.fit(X_train, y_train)

# Predict and evaluate
y_pred_svm = clf_svm.predict(X_test)
y_prob_svm = clf_svm.predict_proba(X_test)[:, 1]  #
    Probability for "crossing" class
print(f"Accuracy: {accuracy_score(y_test, y_pred_svm):.2f}")
print(classification_report(y_test, y_pred_svm))

# Explore different probability thresholds for prediction
thresholds = [0.3, 0.4, 0.5, 0.6, 0.7]
for threshold in thresholds:
    y_pred_threshold = (y_prob_svm >= threshold).astype(int)
    print(f"\nThreshold: {threshold}")
    print(classification_report(y_test, y_pred_threshold))
```

Results:

Class	Precision	Recall	F1-Score
No Crossing	0.90	0.90	0.90
Crossing	0.90	0.90	0.90

Accuracy: **90%**

6.4 Decision Tree

Algorithm Description:
Decision Tree is a hierarchical model that splits the data based on feature values to create a tree-like structure of decisions. The algorithm recursively partitions the feature space to maximize information gain at each split. Decision trees are highly interpretable and can capture non-linear relationships, but they can be prone to overfitting without proper pruning or regularization.

Code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report,
    accuracy_score
import matplotlib.pyplot as plt
from sklearn import tree

# Initialize and train the model
clf_dt = DecisionTreeClassifier(max_depth=5, min_samples_split
    =5)
clf_dt.fit(X_train, y_train)

# Predict and evaluate
y_pred_dt = clf_dt.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred_dt):.2f}")
print(classification_report(y_test, y_pred_dt))

# Feature importance
feature_importances = clf_dt.feature_importances_
feature_names = poly.get_feature_names_out()
indices = np.argsort(feature_importances)[::-1]
print("Top 5 important features:")
for i in range(5):
    print(f"{feature_names[indices[i]]}: {feature_importances[
        indices[i]]:.4f}")

# Visualize the tree (limited depth for clarity)
plt.figure(figsize=(20, 10))
tree.plot_tree(clf_dt, max_depth=3, feature_names=
    feature_names, filled=True)
plt.savefig('decision_tree.png', dpi=300, bbox_inches='tight')
```

Results:

Class	Precision	Recall	F1-Score
No Crossing	0.88	0.87	0.88
Crossing	0.87	0.87	0.87

Accuracy: **87%**

7 Detection of νELN-XLN Crossings

Our discussion thus far has focused exclusively on detecting νELN crossings. However, in realistic astrophysical environments like CCSNe and NSMs, the angular distributions of ν_x and $\bar{\nu}_x$ can differ significantly. This difference becomes particularly pronounced when accounting for the potential creation of muons at the core of these extreme objects. To accurately identify FFCs under realistic conditions, we must expand our focus to detecting νELN-XLN crossings rather than confining ourselves to just νELN crossings.

7.1 Complexities in νELN-XLN Detection

The detection of νELN-XLN crossings presents several key differences compared to νELN crossing detection:

- **Increased Feature Complexity:** We now require seven features instead of three: $\alpha_{\nu e}$, $\alpha_{\nu x}$, $\alpha_{\bar{\nu} x}$, $F_{\nu e}$, $F_{\bar{\nu} e}$, $F_{\nu x}$, and $F_{\bar{\nu} x}$, where $\alpha_{\nu\beta} = n_{\nu\beta}/n_{\nu e}$.
- **Higher Classification Error:** The increased number of features and greater information demand significantly contribute to elevated classification error.
- **Complex Profile Structure:** When ν_x and $\bar{\nu}_x$ exhibit disparities, the νELN-XLN profile can show intricate characteristics, potentially including multiple crossings.

7.2 Methodology for νELN-XLN Detection

For training our ML models, we used artificial distributions for neutrinos, given the lack of labeled data regarding νELN-XLN crossings. Our data preparation involved:

- Setting $\alpha_{\nu e} - \alpha_{\nu x(\bar{\nu} x)}$ in the ranges $(0, 2.5)$ and $(0, 3.0)$ respectively
- Assuming a maximum 40% difference between ν_x and $\bar{\nu}_x$ quantities, consistent with realistic simulations
- Maintaining the hierarchy $F_{\nu e} \lesssim F_{\bar{\nu} e} \lesssim F_{\nu x(\bar{\nu} x)}$ in accordance with observations from realistic data

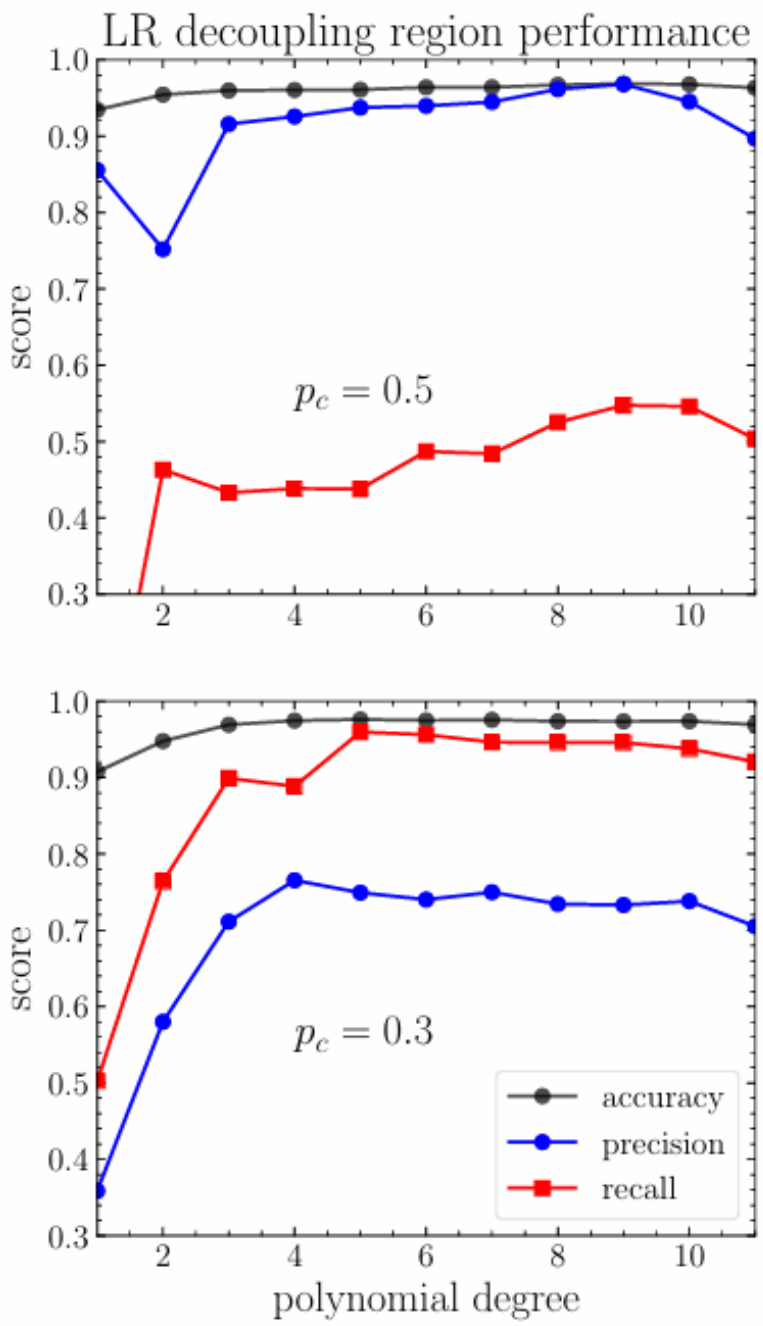

Figure 1: Performance of the LR algorithm in the neutrino decoupling region as a function of polynomial degree. The upper panel shows results with probability threshold $p_c = 0.5$, while the lower panel shows results with $p_c = 0.3$. Reducing p_c enhances recall but reduces precision, illustrating the precision-recall tradeoff.

7.3 Performance Analysis

The performance of our ML models in detecting νELN-XLN crossings is presented in Table 1. Notably, the overall performance lags behind that of νELN crossing detection. This disparity can be attributed to:

- More intricate patterns governing the crossings
- Increased label noise in the training data
- The higher-dimensional feature space requiring more complex decision boundaries

Figure 1 illustrates the performance of the Logistic Regression algorithm across different polynomial degrees. The precision and recall scores typically exhibit opposing trends due to the precision-recall tradeoff. By decreasing the threshold probability (p_c), we can enhance recall significantly, though at the cost of reduced precision.

Table 1: Summary of metric scores of ML algorithms for νELN-XLN crossing detection

LR (n=2) (88%)			
Class	**Precision**	**Recall**	**F1-score**
No crossing	87%	89%	88%
Crossing	88%	86%	87%
KNN (n=3) (88%)			
Class	**Precision**	**Recall**	**F1-score**
No crossing	89%	88%	89%
Crossing	88%	89%	88%
SVM (88%)			
Class	**Precision**	**Recall**	**F1-score**
No crossing	92%	84%	88%
Crossing	85%	93%	89%
Decision tree (87%)			
Class	**Precision**	**Recall**	**F1-score**
No crossing	87%	87%	87%
Crossing	87%	87%	87%

These results demonstrate that while detecting νELN-XLN crossings is more challenging than detecting νELN crossings, our ML approaches still achieve reasonably good performance. The SVM algorithm shows the most balanced performance with high recall for crossing detection (93%), though at the cost of slightly lower precision (85%).

8 Graphical Analysis

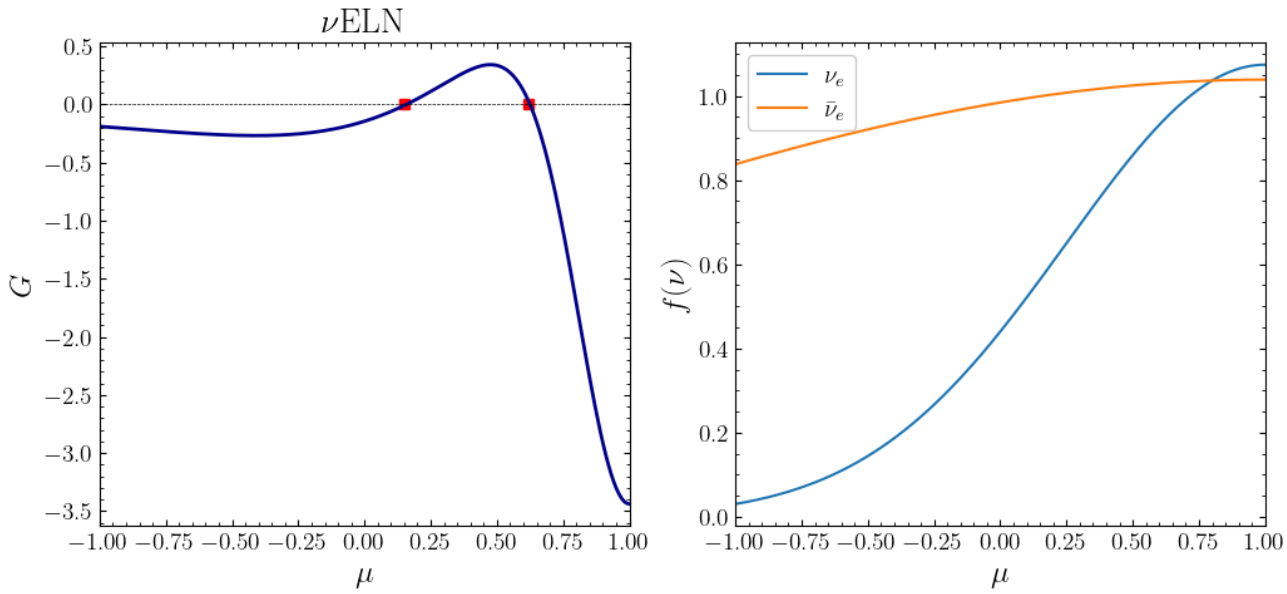

Figure 2: Polynomial Degree vs. Model Accuracy for Different Algorithms

Figure 2 illustrates how the polynomial degree affects model accuracy across different algorithms. For Logistic Regression, accuracy peaks around polynomial degree 2, indicating the optimal balance between model complexity and generalization capability. Beyond degree 2, the model begins to overfit the training data, leading to decreased performance on unseen data.

Similarly, the SVM model shows its best performance at lower polynomial degrees, while KNN demonstrates more stable performance across different polynomial transformations. The Decision Tree algorithm shows slightly lower but consistent performance, regardless of the polynomial features.

This analysis confirms that simple models with appropriately chosen polynomial features can effectively capture the non-linear relationships in the neutrino flavor conversion data without requiring more complex algorithms.

9 Conclusion

We successfully implemented ML models to detect fast neutrino flavor conversions using both νELN and more complex νELN-XLN crossing approaches. Our key findings include:

- Support Vector Machine performed the best with **90% accuracy** for νELN detection.
- For νELN-XLN detection, all models performed similarly with accuracies around **87-88%**.
- The precision-recall tradeoff can be leveraged by adjusting probability thresholds to optimize for specific detection requirements.
- Polynomial feature engineering significantly impacts model performance, with degree 2 typically providing optimal results.

Future directions:

- Explore ensemble methods such as Random Forests and Boosting models to potentially improve detection accuracy.
- Investigate Deep Learning approaches, particularly neural networks with appropriate architectures, for handling the higher-dimensional feature space.
- Develop techniques for non-axisymmetric crossing detection to account for more realistic astrophysical scenarios.
- Integrate model outputs into full-scale CCSN simulations to better understand explosion dynamics and nucleosynthesis.
- Explore the possibility of creating an efficient surrogate model that can be directly incorporated into existing CCSN simulation codes.

The successful application of machine learning to this problem opens new avenues for efficiently modeling neutrino physics in extreme astrophysical environments, potentially leading to more accurate simulations of supernovae and neutron star mergers.

10 References

1. S. Abbar, H. Nagakura, "Detecting Fast Neutrino Flavor Conversions with Machine Learning", 2024.

2. Tamborra, I., et al., "Neutrino emission characteristics and detection opportunities based on three-dimensional supernova simulations". Physical Review D, 90(4), 045032, 2019.

3. Chakraborty, S., Hansen, R., Izaguirre, I., Raffelt, G., "Collective neutrino flavor conversion: Recent developments". Nuclear Physics B, 908, 366-381, 2018.

4. Morinaga, T., Nagakura, H., Kato, C., Yamada, S., "Fast neutrino-flavor conversion in the preshock region of core-collapse supernovae". Physical Review Research, 2(1), 012046, 2020.

5. Bollig, R., et al., "Muon creation in supernova matter facilitates neutrino-driven explosions". Physical Review Letters, 119(24), 242702, 2017.

6. Fischer, T., et al., "Neutrino signal from proto-neutron star evolution: Effects of muon production and opacities". Physical Review D, 102(12), 123001, 2020.

7. Guo, G., et al., "Muons in core-collapse supernovae and their effect on electron neutrino fluxes". Physics Letters B, 809, 135707, 2020.

8. Kato, C., et al., "Sensitivity of neutrino-driven explosions to muon-tau lepton number exchange". Physical Review D, 105, 083025, 2022.

9. Python Libraries: scikit-learn, pandas, matplotlib, numpy - Documentation and Tutorials, 2024.