NuFast-Earth User Guide

Peter B. Denton[1, *] and Stephen J. Parke[2, †]

[1] *High Energy Theory Group, Physics Department Brookhaven National Laboratory, Upton, NY 11973, USA*

[2] *Theoretical Physics Department, Fermi National Accelerator Laboratory, Batavia, IL 60510, USA*

v1.0.2

CONTENTS

I. INTRODUCTION

This guide explains the usage of the NuFast-Earth code which computes neutrino oscillation probabilities through the Earth and is useful for atmospheric, solar, and supernova neutrinos. The accompanying paper that describes the physics is [1] and is based, in part, on the NuFast-LBL code[1] and the associated paper [2] which is optimized for long-baseline (LBL) neutrino oscillations from accelerators or reactors. While the code presented here can be used for LBL calculations we recommend using the NuFast-LBL code for that problem as there is less computational overhead.

The general flow of the code for atmospheric neutrinos is shown as a flow-chart in figure 1. Here is a simple version of the flow for atmospheric neutrinos:

* pdenton@bnl.gov; 0000-0002-5209-872X
† parke@fnal.gov; 0000-0003-2028-6782

[1] See github.com/PeterDenton/NuFast-LBL

1. Create a `Probability_Engine` object.

2. Create an `Earth_Density` object.

3. Set the oscillation parameters (`Set_Oscillation_Parameters`), energy and zenith angle spectra (`Set_Spectra`), and the Earth details (`Set_Earth`) in the probability engine (in any order).

4. Ask the probability engine to get the probabilities (`Get_Probabilities`), at which point all the computations are done.

5. The user can then change individual components and then ask to get the probabilities again, at which point *some* but not *all* calculations will be done again, as necessary.

We first provide two simple minimal working examples for atmospheric and solar neutrinos to be explained in detail below.

```
// Atmospheric neutrino minimal example
Probability_Engine probability_engine;
probability_engine.Set_Oscillation_Parameters(0.307, 0.02195, 0.561, 177 * M_PI / 180, 7.49e-5, 2.534e-3, true);
std::vector<double> energies = {1, 2, 3, 4, 5}; // GeV
std::vector<double> coszs = {-1, -0.5, 0, 1}; // core-crossing to horizontal to down-going
probability_engine.Set_Spectra(energies, coszs);
PREM_NDiscontinuityLayer earth_density(2, 10, 10, 5);
probability_engine.Set_Earth(2, &earth_density); // detector depth in km
std::vector<std::vector<Matrix3r>> probabilities = probability_engine.Get_Probabilities();
```

```
// Solar neutrino minimal example
Probability_Engine probability_engine;
probability_engine.Set_Oscillation_Parameters(0.307, 0.02195, 0.561, 177 * M_PI / 180, 7.49e-5, 2.534e-3, true);
std::vector<double> energies = {1, 2, 3, 4, 5}; // GeV
std::vector<double> coszs = {-1, -0.5, 0, 1}; // core-crossing to horizontal to down-going
probability_engine.Set_Spectra(energies, coszs);
PREM_NDiscontinuityLayer earth_density(2, 10, 10, 5);
probability_engine.Set_Earth(2, &earth_density); // detector depth in km
probability_engine.Set_rhoYe_Sun(100 * (2. / 3)); // g/cm^3
std::vector<std::vector<Matrix3r>> probabilities = probability_engine.Get_Solar_Night_Probabilities();
```

The details of the code are described in the following sections.

II. USAGE RECOMMENDATIONS

Before getting in to the details, we provide our key recommendations on how to use the code within its own choices and how to best implement it in a larger framework. Our most important recommendations, based on our testing, are:

1. Put loops over θ_{23}, δ, and the production height on the inside of the code. This is an absolute must as it takes advantage of the caching feature and can lead to orders of magnitude of speed up.

2. `Set_Eigenvalue_Precision(1)`. This provides many orders of magnitude more precision than is needed for any future experiment while still providing noticeable speed gains over the exact eigenvalues.

3. `PREM_NDiscontinuityLayer earth_density(2, 10, 10, 5)`. This provides a good modeling of the Earth with all the primary features clearly present. If additional detail is needed, these numbers should be proportionally scaled up.

Another recommendation is to prioritize zenith angle points over energy points when defining the array. That is, 100 zenith angles and 50 energies is about 10%-25% faster than 50 energies and 100 zenith angles. This is because varying the energy is more expensive than varying the zenith which does not require recomputing the eigenvalues and eigenvectors (except in the `PREM_Full` and `PREM_Four` Earth models, although those are not recommended for precision reasons).

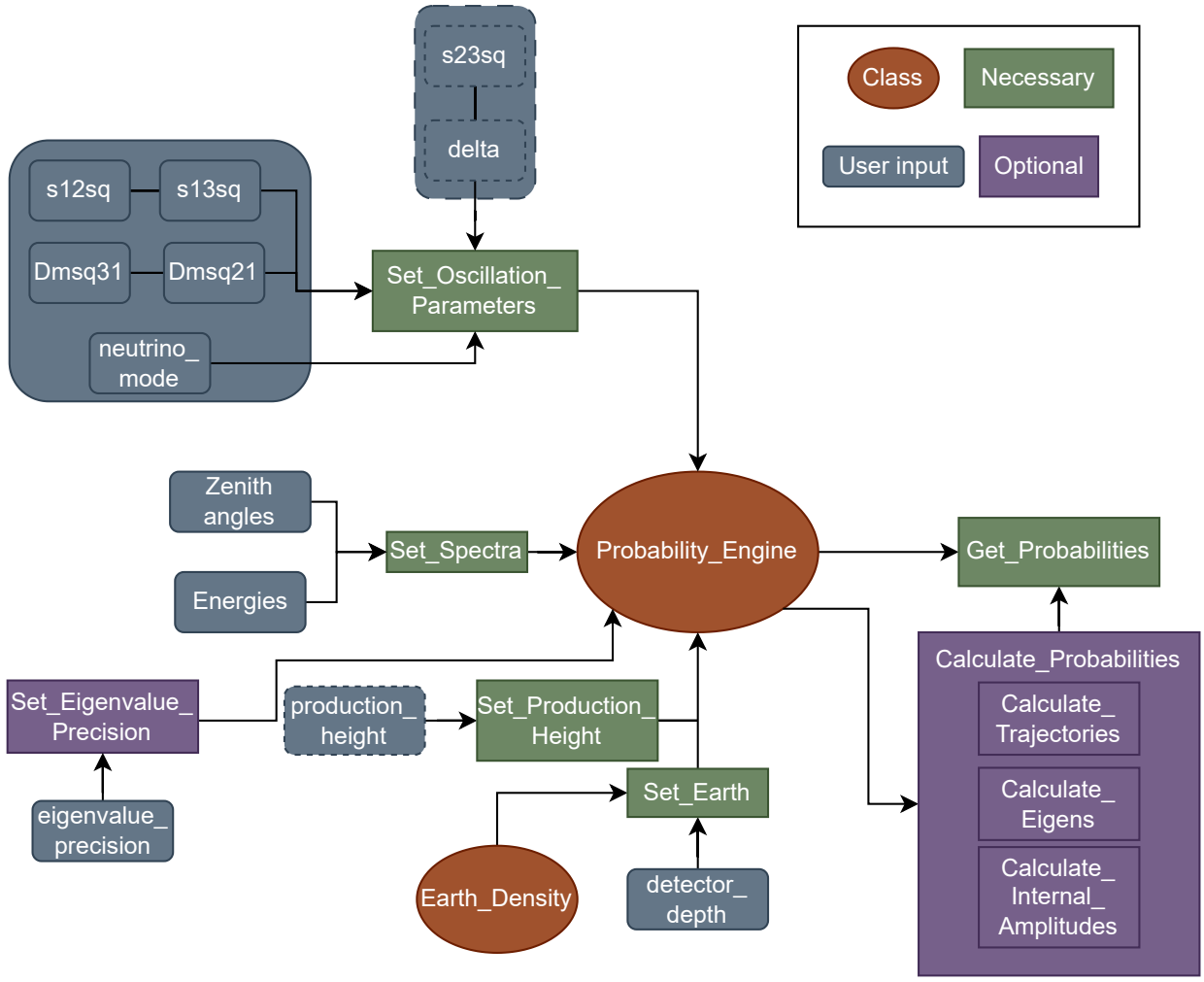

FIG. 1. The typical flow of the code for atmospheric calculations. The user provides inputs in the blue rounded boxes. The dashed inputs (`s23sq`, `delta`, and `production_height`) indicate that not all calculations need to be repeated when they are changed. The user instantiates the brown ellipses. The user initializes the `Probability_Engine` object with the green rectangles. The user then calls `Get_Probabilities` which calls the various `Calculate` functions in purple rectangles as needed.

III. PROBABILITY ENGINE

The core object used to compute probabilities is the `Probability_Engine` class. This class is responsible for keeping track of the set up and remembering things that have already been calculated. It is designed with a medium amount of flexibility. Any usage of the program should begin with creating an instance of `Probability_Engine` which takes no arguments. Next, several parameters *must* be set before any oscillation probabilities may be calculated.

A. Oscillation Parameters

One must set the oscillation physics details with `Set_Oscillation_Parameters` which takes six numbers and one boolean:

```
void Set_Oscillation_Parameters(double s12sq, double s13sq, double s23sq, double delta, double Dmsq21, double Dmsq31,
    bool neutrino_mode);
```

The first three parameters are $\sin^2\theta_{ij}$ followed by δ in radians. The next two parameters are Δm^2_{j1} in eV^2. The inverted mass ordering is automatically handled for `Dmsq31`< 0. The final parameter is a boolean which is true for neutrinos and false for antineutrinos. The `Set_Oscillation_Parameters` function must be called before oscillation probabilities can be calculated.

After `Set_Oscillation_Parameters` has been called, the user may want to change an individual parameter. This can be done via the functions:

```
void Set_s12sq(double s12sq);
void Set_s13sq(double s13sq);
void Set_s23sq(double s23sq); // fast
void Set_delta(double delta); // fast
void Set_Dmsq21(double Dmsq21);
void Set_Dmsq31(double Dmsq31);
void Set_neutrino_mode(bool neutrino_mode);
```

where the inputs are in the same form as above. Note that if one calls these seven functions but not `Set_Oscillation_Parameters`, the probability engine will not know that it has been properly set up and will throw an error. Also note that some of the inputs are labeled as "fast". This is because if the probability has already been calculated and then θ_{23} or δ is changed, many of the computations can be reused. So if one loops over changes in the oscillation parameters, we encourage putting the changes to `s23sq` and `delta` in the innermost loops as appropriate.

There are also getter functions for the above:

```
double Get_s12sq();
double Get_s13sq();
double Get_s23sq();
double Get_delta();
double Get_Dmsq21();
double Get_Dmsq31();
int Get_neutrino_mode_sign();
```

where the last one returns a $+1$ for neutrino mode and a -1 for antineutrino mode.

One can optionally set a parameter that determines the precision of the eigenvalues:

```
void Set_Eigenvalue_Precision(int eigenvalue_precision);
```

This takes an integer and is set to `1` by default. Any value < 0 computes the eigenvalues using the Cardano formula [3] and is exact to machine precision. Values ≥ 0 use an approximate form described in [2]; larger numbers are more precise. A value of `0` is fairly precise and a value of `1` provides significantly more precision than is necessary for any next generation oscillation experiment. That is, a value of `1` is more precise than is needed for DUNE, HK, or JUNO. A value of `2` is equal to the Cardano formula at around the double precision limit for typical oscillation parameters and is faster than the Cardano formula.

An example code through this stage is shown here:

```
// Initialize the engine
Probability_Engine probability_engine;

// Required: Set the oscillation parameters to nu-fit 6 best fit values
probability_engine.Set_Oscillation_Parameters(0.307, 0.02195, 0.561, 177 * M_PI / 180, 7.49e-5, 2.534e-3, true);

// Optional: Set the eigenvalue precision
probability_engine.Set_Eigenvalue_Precision(1);
```

In order to use things like `M_PI`, one needs to include `cmath` via `#include <cmath>`. At this point, nothing has been computed.

B. Energy and Zenith Angle Distributions

One can compute the oscillation probability for a single energy and a single zenith angle, but the real power is when one wants to compute over an array of energies and zenith angles. The user should create vectors of energies and zenith angles and load them into the probability engine using the function `Set_Spectra`:

```
void Set_Spectra(std::vector<double> Es, std::vector<double> coszs); // GeV
```

where all energies are in GeV, whether the context is atmospheric neutrinos, solar neutrinos, or supernova neutrinos. Zenith angles are defined such that `cosz=-1` is core crossing and `cosz=+1` is down going. An example code of this is

```
std::vector<double> energies = {1, 2, 3, 4, 5}; // GeV
std::vector<double> coszs = {-1, -0.5, 0, 1}; // core-crossing to horizontal to down-going
// Required: Set energy and zenith angle arrays
probability_engine.Set_Spectra(energies, coszs);
```

Note that the energy and zenith angle arrays need not be the same size and the vectors can be initialized in a variety of ways. We also note that one needs to include the standard vector header: `#include <vector>`.

IV. EARTH MODELS

Separate from the `Probability_Engine`, the Earth model must be initialized, which is its own class. This describes the spherically symmetric geometry of the Earth. Several example classes are provided, each of which are derived from the abstract class `Earth_Density`. This allows the user to create more complex classes that will still work with the rest of the code. Each class sets the number of sharp discontinuities `n_discontinuities`. Each class stores the radii (in km) of the discontinuities in a vector `discontinuities`. Each class sets an important boolean: whether the shells between the discontinuities are to be treated as constant or varying: `constant_shells`. Finally, each class needs to have a function `double rhoYe(double r)` that returns the density times electron fraction ρY_e in g/cm^3 as a function of radius in km.

In order to create a new class to describe an Earth density model, three main code blocks are needed; we show a simple example motivated by the `prob3++` implementation of PREM [4]. The first piece of code is that the Earth model needs to be mentioned in a header file:

```
// Earth.h
class PREM_Prob3 : public Earth_Density
{
    public:
        PREM_Prob3();
        double rhoYe(double r);
};
```

Note that it is possible to make the initialization take one or more arguments if desired which could change the number or position of the layers or the densities.

Next, in a source file, the initialization function must be defined followed by the density function.

```
// Earth.cpp
PREM_Prob3::PREM_Prob3()
{
    n_discontinuities = 4;
    discontinuities.reserve(n_discontinuities);
    discontinuities[0] = 1220.;
    discontinuities[1] = 3480.;
    discontinuities[2] = 5701.;
    discontinuities[3] = 6371.;
    Ye = 0.5;
    constant_shells = true;
}
double PREM_Prob3::rhoYe(double r) // g/cm^3
{
    assert(r >= 0);
    if (r > 6371.) return 0.;
    if (r > 5701.) return Ye * 3.3;
    if (r > 3480.) return Ye * 5.;
    if (r > 1220.) return Ye * 11.3;
    return Ye * 13.;
}
```

The `rhoYe` function can be any arbitrary function that is non-negative. As a convenience, the full PREM function has been defined in `Earth.cpp` as `double PREM_Full_rho(double r)` which can then be called from within a class. We recommend starting sequential `if` statements from large radii as the `rhoYe` function is likely to be called more times for large radii than for small radii. In order to use `assert`, one needs to include `cassert` via `#include <cassert>`.

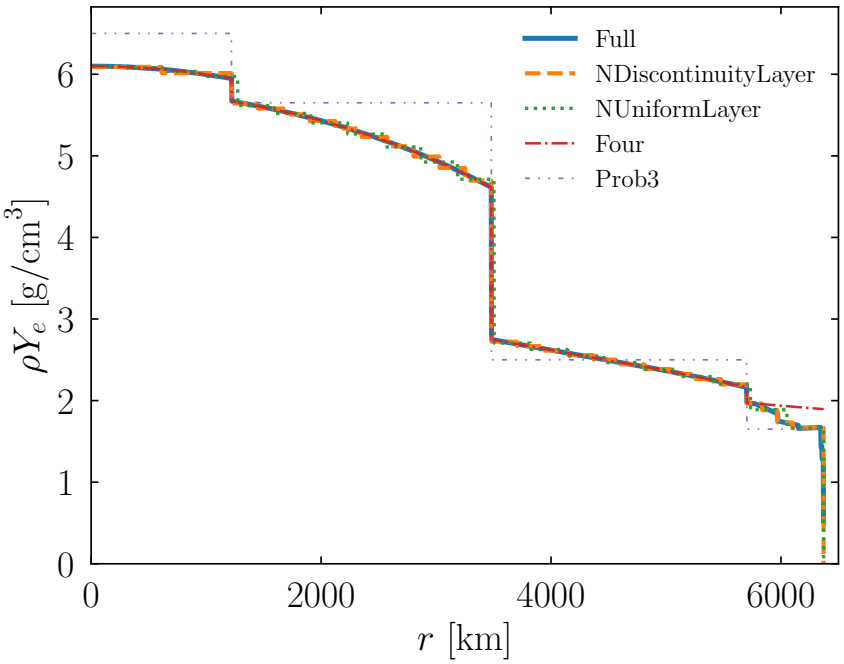

FIG. 2. The density ρ times electron fraction Y_e for various different models. For `PREM_NDiscontinuityLayer` we used $(2, 10, 10, 5)$ for the four regions and for `PREM_NUniformLayer` we used 20 total layers.

Various Earth models are coded up. We briefly list them and their features.

1. `PREM_NDiscontinuityLayer` takes either four or one integer input. With four inputs the number of constant layers in each of the four main parts of the Earth: inner core, outer core, inner mantle, and outer mantle, can be set. With one input, the number of constant layers in each of the four main parts of the Earth are the same. The densities in each layer are taken at the midpoint in the layer from the full PREM model.

2. `PREM_NUniformLayer` follows the same strategy as `PREM_NDiscontinuityLayer` but with layers distributed uniformly across the Earth. For a large number of layers this will generally be quite similar to the equivalent of the `PREM_NDiscontinuityLayer`, but for fewer layers the boundaries between e.g. the core and the mantle may not be correctly realized.

3. `PREM_Full` uses the full varying PREM model with varying shells.

4. `PREM_Four` is the same as `PREM_Full` but uses only four layers ignoring the small changes near the Earth's surface. It includes only the inner/outer core and inner/outer mantle which dominate the oscillation effects.

5. `PREM_Prob3` uses four constant shells taken from the `Prob3++` code, to provide a comparison.

The different Earth density models are plotted in fig. 2.

Once the Earth density model is created, it must be loaded into the probability engine, along with the detector depth (in km) which should be non-negative.

```
void Set_Earth(double detector_depth, Earth_Density *earth_density);
```

The detector depth accounts for the fact that most detectors are inside the Earth.

Optionally, one can set the production height in the atmosphere (set to zero by default). As with the detector depth, it is in km and is non-negative.

```
void Set_Production_Height(double production_height); // km, fast
```

Changing the production height after the probabilities have been calculated is computationally extremely cheap, which is why it is labeled as "fast". Thus loops over production height should be inner loops.

An example code through this stage, to add to the previous example code above, is

```
// Create Earth model instance
PREM_NDiscontinuityLayer earth_density(2, 10, 10, 5); // inner core, outer core, inner mantle, outer mantle
// Required: Set Earth details
probability_engine.Set_Earth(2, &earth_density); // detector depth in km
// Optional: Set production height
probability_enginge.Set_Production_Height(10); // km, fast
```

Still no computations have been performed up to this point.

V. COMPUTATIONS

Once the oscillation parameters, Earth density model, and spectra are set (the order in which these are set does not matter), one can get the probabilities with the function `Get_Probabilities` which returns at 2D array of matrices; the first dimension refers to the energy array and the second dimension to the zenith angle array. If the probabilities have already been calculated and nothing has changed, this simply returns the stored probabilities again without doing any computations, hence using "Get" in the name. If the probabilities have not been calculated since something has changed, then they will be calculated. The `Get_Probabilities` function is of the form:

```
std::vector<std::vector<Matrix3r>> Get_Probabilities();
```

The `Matrix3r` is a custom lightweight class for 3×3 real matrices which requires the Matrix header via `#include "Matrix.h"`.

So an example code building on the above code to complete the calculations and print out the probabilities is

```
std::vector<std::vector<Matrix3r>> probabilities;
probabilities = probability_engine.Get_Probabilities();
for (int i = 0; i <= n; i++)
{
    for (int j = 0; j <= n; j++)
    {
        printf("E=%g, cosz=%g\n", energies[i], coszs[j]);
        probabilities[i][j].Print(1); // Print the matrix with one extra empty line afterwards
    } // j, n, cosz
} // i, n, energy
```

Accessing an individual probability is done by accessing the elements in the matrix stored as the `arr` object; to access the $\nu_\mu \to \nu_e$ probability for the first energy and the first zenith angle in the above examples, one calls `probabilities[0][0].arr[1][0]` where the first two indices refer to the energy and zenith angle bin, and the last two to the flavor.

Calling `Get_Probabilities` will automatically call the function `Calculate_Probabilities` if they have not already been calculated since the last change. Once the probabilities are calculated, the 2D array of matrices are stored in the engine. The function `Calculate_Probabilities` in turn calls the functions `Calculate_Trajectories`, `Calculate_Eigens`, `Calculate_Internal_Amplitudes`, and `Calculate_Eigen_Vac` as needed, see below for more information on the behavior of these functions. It is also possible to call these functions one by one with identical results which is useful for testing purposes.

As mentioned above, we recommend looping over the oscillation parameters in an intelligent way. For example, if one wants to vary Δm^2_{31} and $\sin^2\theta_{23}$, we recommend putting the $\sin^2\theta_{23}$ loop on the inside and then using the `Set_Dmsq31` and `Set_s23sq` commands instead of setting all the oscillation parameters. Then, after `Set_s23sq` is called, one only needs to recompute the probabilities, while calling `Set_Dmsq31`, for example, requires also recomputing the eigenvalues and the internal eigenvectors as well. Changing the Earth density model or the production height/detector depth will require recomputing everything from scratch.

A. Trajectories

Once the zenith angles are set in the probability engine with `Set_Spectra` and the Earth density model, production height, and detector depth are set with `Set_Earth`, one can call `Compute_Trajectories`. Note that this is automatically called, as needed, when one calls `Get_Probabilities`.

This function computes pairs of densities (g/cm^3) and distances (km) and how they are structured depends on the nature of the Earth density model. The trajectories are contained in two separate 2D arrays of pairs. The first

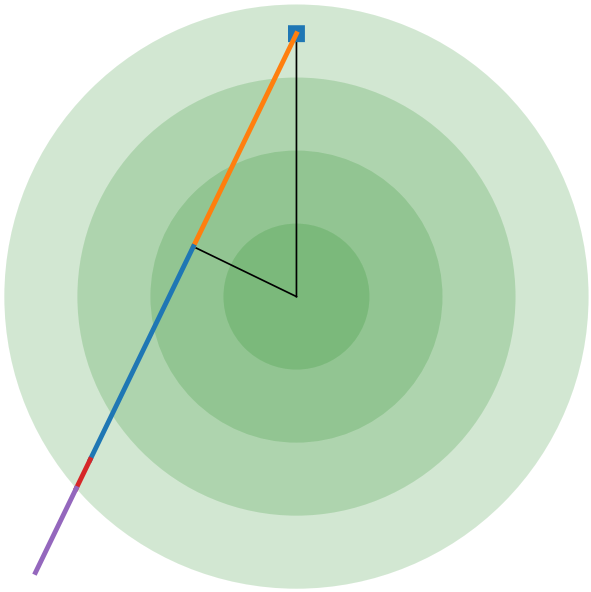

FIG. 3. A schematic of the geometry of the trajectory within the Earth. The trajectory for the first set of $(L, \rho Y_e)$ pairs are shown in red from the atmosphere into the Earth to a depth of the detector depth. The trajectory for the second set is shown in green from the detector depth to the deepest point in the Earth. The amplitude along the final trajectory to the detector (blue square), shown in orange, is the same as the green trajectory, up to a transpose, and thus does not need to be separately computed.

dimension is for the zenith angle and the second is for the layers between the discontinuities. The two sets of trajectory arrays are for the production point to the detector depth and the second is for the detector depth to the deepest point in the Earth, see fig. 3.

This approach of splitting up the trajectories yields a reduction of a factor of nearly 2 in the number of computations through the Earth for Earth models with many layers as the final segment in orange – from the deepest point to the detector – is essentially the same as the previous segment in green from the detector depth to the deepest point.

The densities are calculated in one of two different ways. If the Earth density model has indicated that the densities are to be treated as constant via `constant_shells=true` in the Earth density model class, then the density in each shell is taken as simply the value in the middle of the shell. If the Earth density model has indicated that the densities are to be treated as variable via `constant_shells=false`, then the density in each shell is integrated along the trajectory. This integration incurs some computational cost, especially if there are a large number of zenith angles, but only needs to be run once independent of the number of energies and changes to the oscillation parameters. The integration approach handles cases where the trajectory goes only slightly into a shell in a region where the density is less (and thus one may be able to use fewer total shells for a desired level of precision), but it also means that the density in each shell is different for each zenith angle requiring the computation of more eigenvalues.

B. Eigenvalues

The method with which the eigenvalues are calculated is based on the (optional) `Set_Eigenvalue_Precision` call; by default this parameter is set to `1`. Any negative number will use the exact Cardano formula for the third eigenvalue while non-negative numbers will iteratively approximate it with larger numbers more precise. After the third eigenvalue is calculated, the other two are determined by applying a certain choice of the characteristic equation that tends .

The eigenvalues that need to be calculated differ depending on whether or not the shells in the Earth model are treated as varying or constant. If they are constant, then the code calculates the eigenvalues for each shell and for each energy and stores them. In principle in some cases (such as solar or galactic neutrinos) some of the innermost shells may not be needed, but as this function is not called for every zenith angle, the additional computational cost

is quite marginal. If the shells are treated as varying then the eigenvalues are calculated for each shell, each energy, and each zenith angle and the probability engine stores them all.

Note that changing θ_{23} or δ does not require recomputing the eigenvalues.

C. Eigenvectors

Once the eigenvalues are computed as needed, the eigenvectors are then computed, using the eigenvalues. The eigenvectors are calculated for the entire trajectory including the distance for each component of the trajectory into the amplitude. We refer to this amplitude as an *inner amplitude* as it contains all the matrix multiplication of the complex amplitudes along the entire trajectory, but does not include the outer multiplication by the matrix containing θ_{23} and δ or the norm squareds to compute the probabilities. The code computes and stores a 2D array of 3×3 complex matrices of amplitudes with one dimension corresponding to the neutrino energy and the second dimension corresponding to the zenith angle. To compute the eigenvectors we use the eigenvector-eigenvalue identity [5] applied to neutrino physics [6] as well as the adjugate expressions given in [7] in a way that minimizes the computations [1, 2]. This step depends on the number of layers, the number of energy bins, and the number of zenith angles and is thus fairly expensive. Thus, as with the eigenvalues, the internal eigenvectors also do not need to be recalculated as θ_{23} and δ are changed.

D. Probabilities

The final step is to take the inner amplitudes and convert these to probabilities. This is a relatively straightforward step as it multiples the inner amplitudes by the θ_{23}, δ matrices and computes the norm squared. The code computes the minimum of these necessary and computes the rest via unitarity. The results are then stored as a 2D array of 3×3 real matrices of probabilities where again the first dimension is over the neutrino energy and the second is over zenith angle.

VI. SOLAR

The solar neutrino version of the code follows much of the same approach as the atmospheric structure described above, but involves some key changes. First, one needs to set the density in the Sun at the production point of neutrinos in the `Probability_Engine` class via the command `Set_rhoYe_Sun` which takes a density times electron fraction ρY_e in g/cm^3,

```
void Set_rhoYe_Sun(double rhoYe_Sun);
```

An example of this function called is

```
// Required: Set the density in the Sun at the production point
probability_engine.Set_rhoYe_Sun(100 * (2. / 3)); // g/cm^3
```

One also needs to set the oscillation parameters with `Set_Oscillation_Parameters` (see section III A), the energy spectra with `Set_Spectra` (see section III B), and the Earth density model with `Set_Earth` (see section IV). Note that the production height must be set to zero and the zenith angle distribution is not used for day time calculations. One can get the solar density if it has been set via

```
double Get_rhoYe_Sun();
```

A. Day Time Solar

The day time solar neutrino probability is calculated assuming that the detector is on the Earth's surface[2]. Note that the zenith angles and Earth models are not used in the day time solar neutrino calculations.

[2] If one wants to include the matter effect in the Earth down to an underground detector, one should use the night code described in the night time solar section and include $\cos\theta_z > 0$ in the spectra.

The probabilities are calculated from each flavor to each flavor (and for neutrinos or antineutrinos as set via the `neutrino_mode`parameters. This handles the standard case as well as exotic scenarios via e.g. dark matter annihilation in the Sun. The probabilities are accessed via the `Get_Solar_Day_Probabilities` function which is of the form

```
std::vector<Matrix3r> Get_Solar_Day_Probabilities();
```

which returns a 1D array of 3×3 real matrices of probabilities. The one dimension is over neutrino energy.

An example code to compute daytime solar neutrino probabilities, once the density in the Sun, oscillation parameters, and the spectra are all set is

```
std::vector<Matrix3r> probabilities;
probabilities = probability_engine.Get_Solar_Day_Probabilities();
for (int i = 0; i <= n; i++)
{
    printf("E=%g\n", energies[i]);
    probabilities[i][j].Print(1); // Print the matrix with one extra empty line afterwards
} // i, n, energy
```

Calling this computes the eigenvectors in the Sun (which in turn computes the eigenvalues in the Sun) and then the matrix to project the mass eigenstates back to the flavor basis. In particular, it calls `Calculate_Vsolar` which computes the eigenvectors inside the Sun; this depends on energy. It also calls `Calculate_Solar_Day_Earth` which computes the matrix $|U_{\alpha i}|^2$ which does not depend on the energy. Finally, it multiplies the two matrices together for each energy and returns the probabilities. Note that the `Calculate_Vsolar` computation is reused for the nighttime neutrinos below.

B. Night Time Solar

The code follows the same approach as the daytime solar code and computes the amplitudes through the Earth using the same inner calculation as for atmospheric neutrinos. As with the daytime solar neutrino computation `Set_rhoYe_Sun` must be called and the zenith angle distribution in `Set_Spectra` is now used. One key detail is that when the Earth model is set, the production height must be set to zero. Then the probabilities are computed by calling `Get_Solar_Night_Probabilities` which is of the form

```
std::vector<std::vector<Matrix3r>> Get_Solar_Night_Probabilities();
```

where the first dimension is of energy and the second is of cosine of the zenith angles. An example code using this is

```
std::vector<std::vector<Matrix3r>> probabilities;
probabilities = probability_engine.Get_Solar_Night_Probabilities();
for (int i = 0; i <= n; i++)
{
    for (int j = 0; j <= n; j++)
    {
        printf("E=%g, cosz=%g\n", energies[i], coszs[j]);
        probabilities[i][j].Print(1); // Print the matrix with one extra empty line afterwards
    } // j, n, cosz
} // i, n, energy
```

Calling `Get_Solar_Night_Probabilities` calls `Get_Vsolar` as with daytime neutrinos, and also `Calculate_Solar_Night_In_Earth`. The function `Calculate_Solar_Night_In_Earth` follows the same approach as for atmospherics by calculating the trajectories, eigenvalues along the trajectories, and the inner eigenvectors along the trajectories. Only the final step, due to the different initial basis, differs from the atmospheric code.

Note that calling `Set_rhoYe_Sun` a second time after the probabilities have already been calculated, does not require recalculating the probabilities through the Earth, only the eigenvectors (and therefore also the eigenvalues) inside the Sun.

VII. GALACTIC SUPERNOVA

The neutrino signal coming from a galactic supernova may well travel through the Earth before detection which will modify the signal somewhat. The same code as for solar neutrinos can also be used for a galactic supernova. If one

sets the production point in the Sun to a high density, e.g. `Set_rhoYe_Sun(1e6)`, then the neutrinos will experience the relevant MSW effect in the Sun via the `Calculate_Vsolar` calculation and the rest of the effects in the Earth will also be calculated correctly when using the solar calls described in section VI.

VIII. COMPILING

Compiling is done by running `make` in the folder with the `Makefile`. Cleaning out the temporary object files and the executable for a fresh build is done via `make clean`. There are several optional flags to consider. Optimization levels can be varied by changing `-O3` to `-Ofast` which tells the compiler to make more optimizations, but may clash with some other software, or to `-O0` for the simplest execution that may miss many straightforward optimizations.

Appendix A: Code Changelog

v1.0.0 The initial release.

v1.0.1 Improved compatibility for some Macs.

v1.0.2 Fixed an error in vacuum propagation for antineutrinos and another when recomputing trajectories. Added a test of detector depth.

[1] P. B. Denton and S. J. Parke, `NuFast-Earth`: Efficient Atmospheric, Solar, and Supernova Neutrino Propagation Through the Earth, (2025), arXiv:2511.04735 [hep-ph].
[2] P. B. Denton and S. J. Parke, Fast and accurate algorithm for calculating long-baseline neutrino oscillation probabilities with matter effects, Phys. Rev. D **110**, 073005 (2024), arXiv:2405.02400 [hep-ph].
[3] G. Cardano, *Ars Magna* (1545).
[4] A. M. Dziewonski and D. L. Anderson, Preliminary reference earth model, Phys. Earth Planet. Interiors **25**, 297 (1981).
[5] P. B. Denton, S. J. Parke, T. Tao, and X. Zhang, Eigenvectors from Eigenvalues: a survey of a basic identity in linear algebra, Bull. Am. Math. Soc. **59**, 31 (2022), arXiv:1908.03795 [math.RA].
[6] P. B. Denton, S. J. Parke, and X. Zhang, Neutrino oscillations in matter via eigenvalues, Phys. Rev. D **101**, 093001 (2020), arXiv:1907.02534 [hep-ph].
[7] A. M. Abdullahi and S. J. Parke, Neutrino oscillations in matter using the adjugate of the Hamiltonian, Eur. Phys. J. C **84**, 707 (2024), arXiv:2212.12565 [hep-ph].