Developing a Solver for Inelastic Neutrino-Nucleon Scattering in Core-Collapse Supernovae

Connor Hainje
Advisor: Prof. Adam Burrows

The mechanisms driving the energy transfer at the heart of core-collapse supernovae are complex, high-energy nuclear processes involving the interactions of nucleons, electrons, neutrinos, and other particles at very high rates. One such process which has not been very well modeled in supernova simulations is that of inelastic neutrino-nucleon scattering. A recently-developed Kompaneets formalism which describes this process, however, allows for a better understanding of the process. This work develops a numerical solver which implements this formalism to simulate the time evolution of inelastic neutrino-nucleon scattering. In the process, this work considers various numerical methods for interpolation, differentiation, and integration. Each method is analytically described and empirically tested to determine which are the most effective for this solver. Sample codes for these methods are also included.

I. INTRODUCTION

In their death, massive stars explode, becoming what are known as "core-collapse" supernovae. These explosions are high energy processes, resulting in the production and distribution of a rich variety of elements throughout the universe [1]. Understanding these events requires developing an understanding of the nuclear physics at the heart of the explosion and, specifically, the mechanism of energy transfer sparking the explosion. Neutrino-matter interactions in particular are believed to be primarily responsible for the energy deposition necessary for explosion.

The dominant neutrino-matter interactions driving the heating of the matter are absorption processes, which are well understood. Inelastic scattering of neutrinos on electrons and nucleons in the matter, however, heats the matter at a somewhat lower rate, on the order of 10% to 20% [2]. Near explosion, being able to correctly account for these effects can make all the difference in predicting the explosion. Thus far, inelastic neutrino-electron scattering has generally been the only effect incorporated in supernova simulations, despite indications that neutrino-nucleon scattering provides larger energy deposition than neutrino-electron scattering. This is primarily because inelastic neutrino-nucelon scattering has not been very well-described.

Recently, however, Wang and Burrows [2] have developed a novel Kompaneets formalism to describe inelastic neutrino-nucleon scattering in core-collapse supernova simulations. In particular, they derive a Kompaneets formalism for the time evolution of the energy distribution of neutrinos, as well as the means to find the energy deposition rate. We can break this down as follows.

Let the neutrino energy distribution be represented by $f(\varepsilon)$, where ε is the neutrino energy. It is a dimensionless distribution. It will also be useful to define a dimensionless energy variable, $x \equiv \varepsilon\beta$, where $\beta = 1/kT$ is the inverse of the nucleon temperature. The time evolution of this distribution is given by

$$\frac{df}{dt} = \frac{1}{x^2}\frac{dI_\nu}{dx}, \tag{1}$$

for I_ν proportional to the energy flux. In the Kompaneets formalism, I_ν is given by

$$I_\nu = \alpha(V^2 + 5A^2)x^6\left[\frac{df}{dx} + f(x) - f(x)^2\right], \tag{2}$$

where α is a coefficient given by physical parameters of the system, and V and A are the vector and axial-vector coupling constants. These coupling constants differ based on whether the neutrino is interacting with a proton or a neutron. They are given by

$$\begin{gathered} V_p = \tfrac{1}{2} - 2\sin^2\theta_W, \quad A_p = \tfrac{1}{2}g_A \\ V_n = -\tfrac{1}{2}, \quad A_n = -\tfrac{1}{2}g_A, \end{gathered} \tag{3}$$

where $\sin^2\theta_W = 0.23129$ and $g_A = -1.2723$. For a given electron fraction Y_e, we sum the effects of the neutron and proton interactions to compute the total I_ν as

$$\begin{aligned} I_\nu = \left[Y_e(V_p^2 + 5A_p^2) + (1 - Y_e)(V_n^2 + 5A_n^2)\right] \\ \alpha x^6\left[\frac{df}{dx} + f(x) - f(x)^2\right], \end{aligned} \tag{4}$$

For brevity, this term will generally be omitted when we write the equation for I_ν. One can consider it to be absorbed into the α coefficient.

The coefficient α which was first introduced above is given by

$$\alpha \equiv \frac{2G^2 n_N}{3\pi\beta^3 m}, \tag{5}$$

where $G^2 = 1.55 \times 10^{-33}$ cm^3 MeV^{-2} s^{-1}, n_N is the nucleon number density, and $m = 939$ MeV is the nucleon mass in natural units. The quantities of interest are ρ_N,

the nucleon mass density, and β, so we can rewrite the coefficient numerically as

$$\alpha = \left(2.092 \times 10^{-3}\ \mathrm{s}^{-1}\right) \times \left(\frac{\rho_N}{10^{10}\ \mathrm{g\ cm}^{-3}}\right) \times \left(\frac{1\ \mathrm{MeV}}{\beta}\right)^3, \quad (6)$$

where it is manifestly apparent that α has dimension $[\text{time}]^{-1}$.

There exists a higher-order correction to the I_ν formula which is given by

$$\lambda(x) = \frac{kT}{mc^2}\frac{8x(2-\delta_N) - (50 - 22\delta_N)}{3 - \delta_N}, \quad (7)$$

with

$$\delta_N \equiv \frac{V^2 - A^2}{V^2 + 3A^2}. \quad (8)$$

Since this has dependence on V and A, we will have two different values of δ_N for proton and neutron interactions. As such, there will be two corrective factors, λ_p and λ_n. The fully-expanded, corrected form of I_ν is

$$I_\nu = \Big[Y_e(V_p^2 + 5A_p^2)(1-\lambda_p) + (1-Y_e)(V_n^2 + 5A_n^2)(1-\lambda_n)\Big] \alpha x^6 \left[\frac{df}{dx} + f(x) - f(x)^2\right], \quad (9)$$

More compactly, absorbing the $V^2 + 5A^2$ factor into α and making the proton-neutron distinction implicit, it may be written

$$I_\nu = \alpha x^6 \left[\frac{df}{dx} + f(x) - f(x)^2\right](1-\lambda). \quad (10)$$

With these conventions, we can write the full differential equation as

$$\frac{df}{dt} = \frac{\alpha}{x^2}\frac{d}{dx}\left[x^6\left(\frac{df}{dx} + f - f^2\right)(1-\lambda)\right] \quad (11)$$

It is also useful to explore the expected steady state of a distribution function evolved using this formalism. In general, we expect f to converge toward a Fermi-Dirac distribution,

$$\frac{1}{e^{(\varepsilon-\mu)/kT} + 1}, \quad (12)$$

where ε is the neutrino energy, T is the nucleon temperature, and μ is the electrochemical potential. μ is a physical parameter dependent primarily on the nucleon temperature kT and the total neutrino number of the initial distribution.[1] Numerical tests are performed to test this prediction in Section VI. We also note that, due to the x^6 dependence of I_ν, we expect to see significantly smaller updates to the lower-energy bins than we see at high-energy bins. This means that the convergence to the Fermi-Dirac distribution will occur most slowly at small energies.

One feature of this formalism is that it automatically guarantees neutrino conservation. Since we are only considering scattering interactions between neutrinos and nucleons, it is important that there is neutrino number conservation. The differential neutrino number is given by $x^2\, df/dt$. Following Equation 1, this is

$$x^2\frac{df}{dt} = \frac{dI_\nu}{dx}, \quad (13)$$

i.e. I_ν gives the flux in energy space. Integrating this equation yields the total change in particle number,

$$\Delta N = \int_{x_{\min}}^{x_{\max}} \frac{dI_\nu}{dx}\, dx = I_\nu(x_{\max}) - I_\nu(x_{\min}). \quad (14)$$

For sufficiently small $x_{\min}$ and large $x_{\max}$, this should be trivially zero. In our solver, however, we consider a discretized energy spectrum. Thus, the total change in particle number is given by I_ν on the last energy bin boundary minus that on the first energy bin boundary. Manually setting these to zero when we compute I_ν guarantees neutrino number conservation.

Expressions for the spectrum and rate of energy deposition can also be derived, as was done in [2]. We merely quote the results.

$$\begin{aligned} \dot{q}_\nu &= kT\left(I_\nu - \frac{d}{dx}(xI_\nu)\right) \\ \dot{Q}_\nu &= \frac{(kT)^4}{2\pi^2\hbar^3c^3}\int dx\, I_\nu \end{aligned} \quad (15)$$

II. DESCRIPTION OF PIPELINE

There are a few assumptions we make first about the format of the data. First, we assume that we know the various information necessary to compute the coefficient α described previously. Next, we assume that we are given the values of f sampled at various positions in x.

We assume the binning in x to be *logarithmically* spaced, such that $x_j = x_0 e^{jw}$ for bin-width w. If we specify the number of bins to be n and the minimum and maximum values of x to be x_0 and x_{n-1}, we know that $x_{n-1} = x_0 e^{(n-1)w}$, so we may solve for w to find

$$w = \frac{\log x_{n-1} - \log x_0}{n-1}.$$

Note that the binning is arithmetically equispaced in $\log x$, since

$$\log x_{j+1} - \log x_j = \log(x_{j+1}/x_j) = w.$$

[1] We do not provide a derivation for the electrochemical potential from our parameters. We do note that it can be found by evolving a system forward in time until it reaches a steady state and finding the energy at which the distribution has value 0.5 (as this corresponds to $\varepsilon = \mu$).

Further, we let the x-values specify bin *centers*, such that $f_j = f(x_j)$ is assumed to be the value representative of f on the interval $[x_j e^{-w/2}, x_j e^{w/2}]$.

A. Reformulating the ODE

There are two primary parts to our solver pipeline: computing I_ν, and then using it to compute the right-hand side of the differential equation. However, there exists a reformulation of the differential equation that is rather powerful and will motivate a similar reformulation of the equation describing I_ν.

To derive this reformulation, we begin with the form given in Equation 1. By multiplying each side by x^3, we obtain the following

$$x^3 \frac{df}{dt} = \frac{d}{dt}\left(x^3 f(x)\right) = x\frac{dI_\nu}{dx}. \tag{16}$$

In this form, there is no division by x required, which grants a speed increase of its own. More importantly, however, this form will allow us to take advantage of the fact that our binning in x is *logarithmic*.

Many differentiation schemes, including all of the ones that we will consider later, benefit immensely from having equally spaced sampling points. This poses a potential problem for us, since our sampling points are not equispaced. However, they *are* equispaced in $\log x$, so if we could instead differentiate with respect to $\log x$, we would be able to take advantage of this performance boost.

Note also that

$$\frac{d}{dx} = \frac{d(\log x)}{dx}\frac{d}{d(\log x)} = \frac{1}{x}\frac{d}{d(\log x)}. \tag{17}$$

If we were to insert this identity into the x^3 scheme differential equation, we would obtain

$$\frac{d(x^3 f)}{dt} = \frac{dI_\nu}{d(\log x)}. \tag{18}$$

This form is quite powerful, as we have managed to transform the right hand side of the equation into a single differentiation with respect to $\log x$. In this form, we can avoid needless multiplications and divisions by x on the right-hand side *as well as* harness the full accuracy increases obtained by using equispaced sampling points. As such, the x^3 scheme allows for a simplified computation that is more efficient and accurate.

Supposing that we have a method for calculating I_ν—which will be fleshed out in more detail below—the general form of the pipeline is as follows. First, we calculate $x^3 f$ from the given values of x and f. Then, we use $x^3 f$ to calculate I_ν on the bin edges. We can then use our calculated values of I_ν to then calculate $dI_\nu/d(\log x)$, the right-hand side of the differential equation. With this method for calculating the right-hand side, we can make use of an ODE solving method to compute the updated values of $x^3 f$, from which we will finally obtain and return the updated values of $f(x)$.

B. Computing I_ν

Perhaps the most important step of the pipeline, and the one we have not yet discussed, is the computation of $I_\nu(x)$. To recall, our givens are the sampling points x, the sampled values of f at those points, and any physical parameters necessary to compute α. Thus, we can find I_ν from Equation 10, reproduced here:

$$I_\nu(x) = \alpha x^6 \left[\frac{df}{dx} + f(x) - f(x)^2\right](1 - \lambda(x)).$$

As described before, we need to compute the values of I_ν on the bin *edges* between the sampling x values in order to ensure that neutrino number is conserved. For the outermost bin edges, which we will label 0 and n, I_ν must be 0. For the inner bin edges, however, we can invert our bin edge/center terminology to consider the existing x sampling points as defining bin edges, with I_ν needing to be computed on the centers of these bins.

One potential method for computing I_ν, then, is to use some differentiation scheme to compute df/dx on the inner bin edges, to interpolate $f(x)$ to the inner bin edges, and then to simply compute the above.

However, we can make use of the x^3 scheme here, choosing to take $x^3 f$ as a given. As such, let $y(x) = x^3 f(x)$. We can rewrite this calculation in terms of y as follows

$$\begin{aligned} x^6\Big[\frac{df}{dx} + f(x) - f(x)^2\Big] &= x^6\frac{df}{dx} + x^6 f(x) - x^6 f(x)^2 \\ &= x^3\left(\frac{dy}{dx} - 3x^2 f(x)\right) + x^3 y(x) - y(x)^2 \\ &= x^3\frac{dy}{dx} + (x^3 - 3x^2)y(x) - y(x)^2 \end{aligned}$$

We can also once again change the differentiation to be with respect to $\log x$, transforming the $x^3(dy/dx)$ term into $x^2 dy/d(\log x)$. Now we can rewrite I_ν using this formulation

$$I_\nu = \alpha(1-\lambda)\left[x^2\frac{dy}{d(\log x)} + (x^3 - 3x^2)y - y^2\right].$$

In this form, it is simpler and more efficient to compute I_ν directly from $y = x^3 f$. Thus, in our solving method, it makes sense to compute $y = x^3 f$ at the beginning of the routine and perform the whole of the computation using these values, only converting back at the end to write out the final updated values of $f(x)$.

III. INTERPOLATION METHODS

During the computation of I_ν, we have to employ interpolative methods to determine the value of $f(x)$ on the inner bin edges, lying between the sampled bin centers. To consider interpolations of this type, we first clarify the form of the specific problem being considered.

Suppose we have $n+1$ sampling points x_0, x_1, $\cdots$, x_n, and we have sampled some function at these points, yielding $y_0 = f(x_0), y_1 = f(x_1), \cdots, y_n = f(x_n)$. We want to interpolate these points to the center of each $[x_i, x_{i+1}]$ interval. We assume that the x-values are equispaced. In practice, our x values are logarithmically spaced, but these methods may still be used by simply interpolating f as a function of $\log x$.

We also note that $f(x)$ must be strictly positive in our simulation. As such, we can build a positivity constraint into our interpolation method. This is done by interpolating the values of $\log y_0$ to $\log y_n$ at the given x values, and then exponentiating the returned interpolated midpoint values. This method guarantees positivity and, in many cases, improves error performance.

A. Linear Lagrange interpolation

The simplest interpolative method for a subinterval $[x_i, x_{i+1}]$ would simply fit a line to the two points (x_i, y_i) and (x_{i+1}, y_{i+1}), and then return the value of this line at the midpoint. This is *linear interpolation*. The equation of the line describing bin i may be given by

$$f_i(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i, \tag{19}$$

where $f_i(x)$ is the interpolating function describing bin i. We can rewrite this function in the form of a Lagrange interpolating polynomial, however [3]

$$f_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1}. \tag{20}$$

For the specific case we are considering, we want the value of f_i at the center of the bin. Plugging in the bin center for x, we find that the equation can be simplified to

$$f_i(x) = \tfrac{1}{2}\left(y_i + y_{i+1}\right). \tag{21}$$

This method is fast and efficient, making use of only the sampled y values. It is also inaccurate, however, due to its inability to approximate curves and the fact that it makes use of very little of the available data.

In order to better elucidate the accuracy of this method, we will make use of an alternative derivation of the same formula. Instead of using Lagrange interpolation, one can instead use Taylor approximations. Recall that x is at the center of the interval $[x_i, x_{i+1}]$. As such, define h to be $x_{i+1} - x = x - x_i$. Then, the bin edges y_i, y_{i+1} can be described by $f(x \pm h)$. We can expand $f(x \pm h)$ about $f(x)$ as follows

$$f(x \pm h) = f(x) \pm h f'(x) + \tfrac{1}{2} h^2 f''(x) \pm \tfrac{1}{6} h^3 f'''(x) + \mathcal{O}(h^4).$$

Then, if we add $f(x-h)$ and $f(x+h)$, we obtain

$$f(x-h) + f(x+h) = 2f(x) + h^2 f''(x) + \mathcal{O}(h^4).$$

This can be solved for $f(x)$ to find

$$f(x) = \tfrac{1}{2}\left(f(x-h) + f(x+h)\right) + \tfrac{h^2}{2} f''(x) + \mathcal{O}(h^4) \tag{22}$$

This formula is identical to the one derived previously when equating $f(x \pm h)$ with y_i and y_{i+1} and truncating the expansion before the h^2 term, but we have obtained additional information about the nature of the error. In particular, we have learned that the "truncation error," i.e. the error incurred by truncating the Taylor expansion, is $h^2 f''(x)/2$, second order in h.

B. Cubic Lagrange interpolation

We can again use Lagrange interpolation, but with higher order interpolants by instead using cubic interpolants. A cubic interpolation requires four points: let these be (x_j, y_j) for $j \in [0, 1, 2, 3]$. The Lagrange interpolating polynomial for these four points is then [3]

$$\begin{aligned} f(x) \approx{}& \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} y_0 \\ &+ \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} y_1 \\ &+ \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} y_2 \\ &+ \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} y_3 \end{aligned} \tag{23}$$

However, we concern ourselves primarily with the value of this interpolant at the midpoint of a bin. As such, we choose the midpoint of bin i to be x, with the surrounding bin edges being x_{i-1}, x_i, x_{i+1}, and x_{i+2}. If we assume that the bin edges are equispaced with a bin width of w, then we can plug in all of these values and simplify to find a more efficient formula for this computation.

$$f_i(x) \;=\; -\frac{1}{16} y_{i-1} + \frac{9}{16} y_i + \frac{9}{16} y_{i+1} - \frac{1}{16} y_{i+2} \tag{24}$$

Another way to arrive at this result is to instead consider performing two Lagrange interpolations on each interval. Each time, we take three points and compute the interpolating quadratic. First, we do this on the forward interval

$\{i, i+1, i+2\}$. Then, we do this on the backward interval $\{i-1, i, i+1\}$. Finally, we average the two results. It can be shown that when computing the midpoint of the central bin for equispaced bins, these two methods return the same formula.

There is a problem with this cubic interpolation method, however, and that is that we do not immediately have a way to deal with the first and last bins. The first bin will lack access to y_{i-1}, and the last bin has no y_{i+2}. Formulating this method as the average of forward and backward quadratic interpolants, however, yields a very natural solution. For the first bin, one takes only the value given by the forward quadratic interpolant. For the last bin, we take the value from the backward quadratic interpolant. For all other bins, however, we can use the full cubic interpolant.

Similar to the linear interpolation method, there again exists an alternative derivation that makes use of Taylor expansions. We will present this derivation here as well, as it better elucidates the accuracy and convergence of the method. Again, we must make the assumption that the bins are all equispaced. As such, letting x be the center of the bin of interest, we can write the surrounding four points as $x \pm h$ and $x \pm 3h$, defining h as before. Their Taylor expansions can be written out as well

$$f(x \pm h) = f(x) \pm hf'(x) + \tfrac{1}{2}h^2 f''(x) \pm \tfrac{1}{6}h^3 f'''(x) + \mathcal{O}(h^4). \quad (25)$$

$$f(x \pm 3h) = f(x) \pm 3hf'(x) + \tfrac{9}{2}h^2 f''(x) \pm \tfrac{9}{2}h^3 f'''(x) + \mathcal{O}(h^4). \quad (26)$$

Now we can take a weighted sum of these components as follows.

$$9(f(x+h) + f(x-h)) - (f(x+3h) + f(x-3h)) = 16f(x) - 6h^4 f^{(4)}(x) + \mathcal{O}(h^6) \quad (27)$$

If we solve for $f(x)$, one obtains the previously derived formula, along with the truncation error term $-3h^4 f^{(4)}(x)/8$, fourth order in h.

C. Cubic spline interpolation

One shortcoming of the previous piecewise Lagrange interpolation methods is that the resulting interpolant over the entire range of x values is not differentiable, but instead often has cusps on the interval boundaries. A more powerful interpolation method that solves this problem is cubic spline interpolation.

Like the piecewise cubic Lagrange interpolation, we will again find a cubic polynomial in each interval that interpolates the endpoints of the interval. However, in the case of cubic Lagrange interpolation, we required that the interpolant match the values of the nearest points outside the interval. For the spline case, however, we instead impose that the first two derivatives must be continuous everywhere. Note that the following derivation of a cubic spline interpolant follows closely the discussion in [3].

On the interval $[x_j, x_{j+1}]$, take the interpolating cubic to be $S_j(x)$. We require, of course, that this cubic interpolates the endpoints of the interval, so $S_j(x_j) = y_j$ and $S_j(x_{j+1}) = y_{j+1}$. Note that this implies that $S_j(x_{j+1}) = S_{j+1}(x_{j+1})$. We also require that the first two derivatives be continuous, so $S'_j(x_{j+1}) = S'_{j+1}(x_{j+1})$ and $S''_j(x_{j+1}) = S''_{j+1}(x_{j+1})$. With these cubics defined, we can define the entire interpolant as

$$S(x) = S_j(x), \quad x_j \le x \le x_{j+1}, \quad (28)$$

noting that $S(x)$ is a twice-differentiable function.

These conditions alone, however, are not enough to fully specify the interpolant. We require one more condition, $S''(x_0) = S''(x_n) = 0$. This is known as the "natural" boundary condition, making our interpolant a natural spline. There exist other boundary conditions, but these are not relevant to our studies.

To construct the spline, we first write the equation for $S_j(x)$,

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3. \quad (29)$$

Thus, there are $4n$ coefficients that must be set: each of the above coefficients for all $j \in [0, n-1]$. Note that as written, $S_j(x_j) = a_j$. Thus, the a coefficients are easily set as $a_j = y_j$.

For convenience, let's define $h_j = x_{j+1} - x_j$ for $j \in [0, n-1]$. From the requirement that $S_j(x_{j+1}) = S_{j+1}(x_{j+1})$, we can also find that

$$y_{j+1} = y_j + b_j h_j + c_j h_j^2 + d_j h_j^3. \quad (30)$$

Now, notice that

$$S'_j(x) = b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2. \quad (31)$$

Similar to before, we find $S'_j(x_j) = b_j$, so the requirement that $S'_j(x_{j+1}) = S'_{j+1}(x_{j+1})$ yields

$$b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2. \quad (32)$$

We may repeat this process for $c_j = S''_j(x_j)/2$, obtaining

$$c_{j+1} = c_j + 3d_j h_j. \quad (33)$$

This system of three equations may be rewritten as

$$\begin{cases} h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} \\ \qquad = \frac{3}{h_j}(y_{j+1} - y_j) - \frac{3}{h_{j-1}}(y_j - y_{j-1}) \\ b_j = \frac{1}{h_j}(y_{j+1} - y_j) - \frac{h_j}{3}(2c_j + c_{j+1}) \\ d_j = \frac{1}{3h_j}(c_{j+1} - c_j) \end{cases} \quad (34)$$

If we take the first of these three equations for all values of j and use our the natural boundary condition that $c_0 = c_n = 0$, we have a linear system of equations that can be solved for the values of c_j. These values can then be used to derive the values of the b and d coefficients, which fully specifies the cubic interpolant. An algorithm to carry out this computation is given in Appendix A.

There is a constraint on the error of this method, given by [4]. If we assume that $f(x) \in C^4$ on the interval and let $h \equiv \max_i(x_{i+1} - x_i)$, then we have

$$\max |f(x) - S(x)| \leq \tfrac{5}{384} h^4 \max |f(x)|, \tag{35}$$

In other words, the spline converges uniformly to f as the interval size decreases and the number of sampling points increases. In particular, we see that our method is accurate up to cubic order in h.

D. Evaluation

To evaluate the performance of these methods, we have developed a simple test. We begin with a simple Gaussian distribution,

$$f(x) = \frac{1}{2} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right], \tag{36}$$

which has mean μ and width σ. We sample this distribution at $n+1$ sampling points $x_0, \cdots, x_n$ with $x_0 = 1$ and $x_n = 100$, yielding sampled values $y_0 = f(x_0), \cdots, y_n = f(x_n)$. Our goal is to interpolate $f(x)$ to the center of each interval $[x_i, x_{i+1}]$, and then test these interpolated values against the true values of f at these interval midpoints. In order to better reflect the nature of interpolation in our solver pipeline, we use logarithmically spaced sampling points, such that $x_j = x_0 e^{jw}$, where w is the bin width. We then perform our interpolation on the points $(\log(x_i), y_i)$ so that the interpolation methods can rely on equal bin spacing.

First, we simply perform this test for a few different numbers of sampling points and plot the resulting interpolated values against the true distribution $f(x)$. These results can be found in Figure 1.

However, as described previously, we can take advantage of an important quality of our data. Our interpolative methods are always used on distributions which *must* be strictly positive. As such, we can take the logarithm of the sampled values, interpolate, and exponentiate the final output values. This guarantees positivity, and many of the methods also enjoy a significant performance boost from this manipulation. Interpolation performed using these methods is shown in Figure 2.

More rigorously, we would like to quantify the performance of these methods as a function of the number of interpolated sampling points. To do so, we will use the least absolute error (LAE). Let the interpolated values be called f_i, and their corresponding interval centers be $x_{i+1/2}$. The LAE is then defined by

$$\text{LAE} = \sum_{i=0}^{n-1} \left| f_i - f(x_{i+1/2}) \right|.$$

We assume that the LAE will have a power law relationship with n, i.e. LAE $\propto n^k$ for some k. Note that this is well justified by the our Taylor expansion error analysis from before, as bin width $w \propto 1/n$. Taking the logarithm of both sides yields $\log(\text{LAE}) \propto k \log(n)$, so we can find k by finding the slope of a line fit to the log-log data.

In our testing, we begin with a given method and number of sampling points. For this case, we then compute the LAE of the computed derivative using Gaussian test distributions with 30 different means ranging from 5 to 25 (and always a width of 3). This is then repeated for many different numbers of sampling points (from 12 to 200) and for each method. The recorded data is shown in Figure 3. For each method, we have also performed a simple linear regression to determine the slope of the best line fit to the data. This slope is given in the legend. These trials have been performed for the interpolation methods both standardly and with the "forced-positive" method. Forced-positive data is plotted with a dashed line and indicated in the legend by (+).

Notice also that we have performed linear regression on each of the lines and included the slope of the resulting line in the legend. This indicates the order of growth of the LAE with respect to the number of bins. The number of bins, however, scales with the reciprocal of the

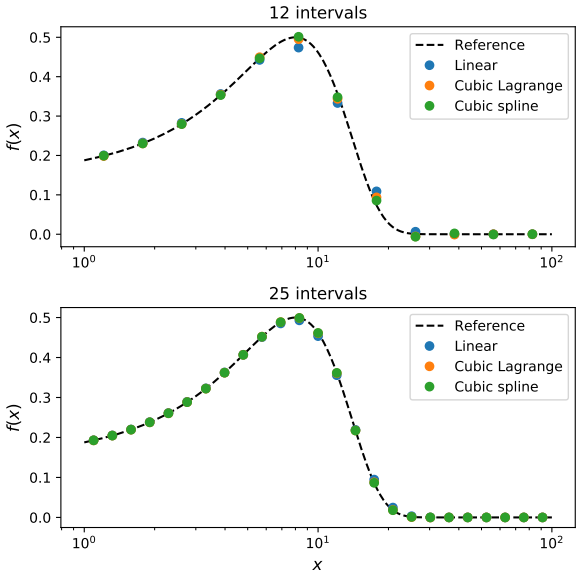

FIG. 1. The results of our interpolation methods for 12 and 25 sampled intervals from a Gaussian of mean $\mu = 8$ and width $\sigma = 5$. Note that the binning is logarithmically spaced, so the x-axis is as well.

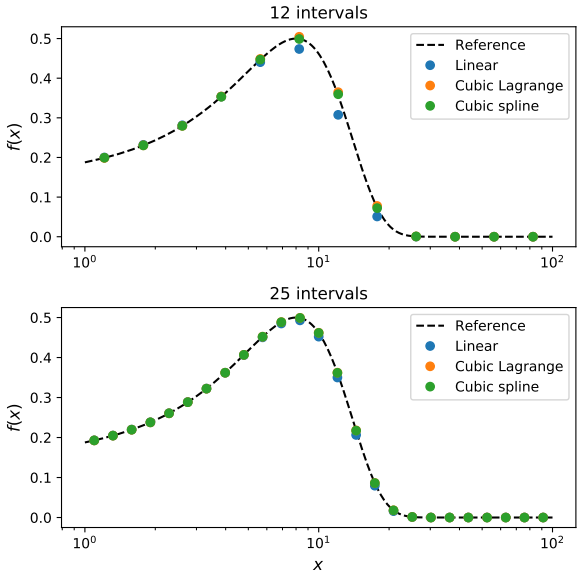

FIG. 2. The results of our interpolation methods for 12 and 25 sampled intervals from a Gaussian of mean $\mu = 8$ and width $\sigma = 5$. In this case, the interpolation methods have been given the logarithm of the sampled values (i.e. y-values equal to $\log y$), and the final outputs have been exponentiated. This guarantees positivity and seems to improve accuracy for points close to zero.

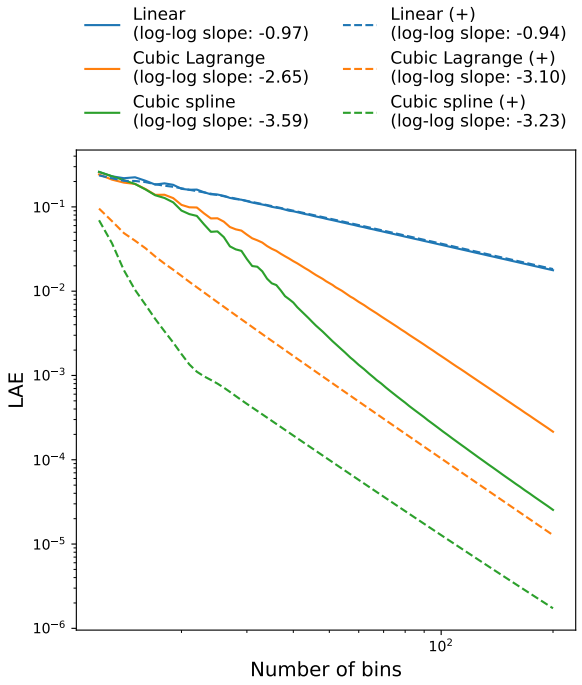

FIG. 3. Error performance of our interpolation methods. We calculate the least absolute error (LAE) for a variety of numbers of sampling bins (from 12 to 200). We then plot $\log_{10}(\text{LAE})$ against $\log_{10}(n)$ as a measure of the order of convergence. A simple linear regression is applied to these log-log data, and the slope of the result is displayed in the legend. This slope is the order of convergence.

step size (the bin width). Thus, a slope of -1 means that LAE $\propto n^{-1} \propto w^1$ (where n is the number of bins and w is the bin width). When considering the forced positive method, then, we can see that the linear method gives order 1 convergence, where the cubic Lagrange and spline methods appear to give approximately order 3 convergence.

Also worth noting is that the data are not very smooth for low numbers of bins. This is a result of an aliasing effect occurring due to the nature of our test distribution. For small numbers of bins, the sampling points are often distributed in such a way that for most Gaussians, important information about the peak of the distribution is lost. This affects different numbers of bins more or less strongly depending on how well the sampling points encode the region of tested Gaussian means.[2] As the number of bins increases, however, the effect is silenced and the curves become much more smooth. The forced-positive method also seems to help reduce this phenomenon.

These plots show that with or without forcing positivity, the linear method performs the worst, with a roughly linear dependence on the sampling interval size. The two cubic methods perform similarly at low numbers of bins. For higher bin counts, the two have appear to have the same convergence behavior, but the cubic spline generally sees higher accuracy. For these methods, the "forced-positive" method also grants a notable increase in accuracy. For small bin counts, the particular target of our studies, either of the cubic methods appears to perform well.

Another important aspect of these methods, however, is their runtime speed. Analytically, we expect of these methods have runtime $\mathcal{O}(n)$. However, we expect cubic spline interpolation to be the slowest, as it must perform several passes over the data points and solve a tridiagonal system of equations, where the other methods are able to finish in a single for loop. To better elucidate these claims, we have also performed empirical timing tests. The data is shown in Figure 4.

The data show a strongly linear dependence on the number of bins. Linear regression was applied to all of the curves; the fit lines can be seen in the legend. These fits have R^2 values of approximately 1. This linear dependence was expectable, as the implementation of all methods relies simply on `for` loops over the number of bins.

In this data, one can see that cubic spline interpolation incurs a very large speed penalty for all regions, with its slope (i.e. the time per bin) a full order of magnitude larger than the linear or cubic Lagrange methods.

[2] Computing the average LAE for many Gaussians of varying means helps to reduce this issue, but it does not alleviate the issue entirely.

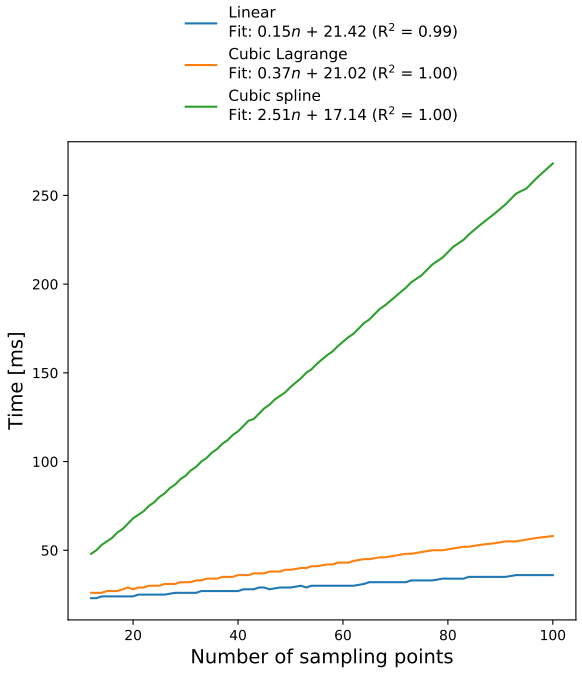

FIG. 4. The runtime performance of our interpolation methods as a function of the number of sampling points. Notice that all methods have a roughly linear dependence on the number of points, elucidated by the linear regression fits with R^2 values very near 1, as shown in the legend.

At low bin counts, this is a rather substantial penalty, especially when compared to the negligible performance gains. For higher bin counts, where the cubic spline enjoyed top performance, its runtime appears prohibitive, especially when our cubic Lagrange methods exhibit the same order of convergence in the limit of high bin counts. As such, we conclude that the best interpolator for our solver is cubic Lagrange interpolation, owing to its near-optimal error and runtime performance, especially for low bin counts.

IV. DIFFERENTIATION METHODS

We also make use of differentiation schemes at two points in our solver pipeline. The first case is when we use the values of $f(x)$ at the zone centers to compute df/dx on the inner zone boundaries, and the second case is when we use the values of $I_\nu(x)$ on the zone boundaries to compute dI_ν/dx on the zone centers. In both cases, we must compute the derivative on the midpoints of the intervals defined by the sampling points, so there is a degree to which interpolation is involved.

For concreteness, we are again considering a case where there are $n+1$ data points, x_0 to x_n and $y_0 = f(x_0)$ to $y_n = f(x_n)$, and we desire the values of df/dx at the midpoint of each $[x_i, x_{i+1}]$ interval. We make the assumption as well that sampling points are equispaced. In our solver, however, the sampling points are logarithmically spaced. This is not much of a problem, though, as all derivatives have been rewritten as derivatives with respect to $\log x$.

More generally, however, this issue can be tackled by making use of the following identity.

$$\frac{dy}{dx} = \frac{1}{x}\frac{dy}{d(\log x)}. \tag{37}$$

As such, one can easily use one of the following differentiation methods with respect to $\log x$, which *is* equispaced, and use the result to obtain the values of dy/dx.

A. Linear method

The simplest possible differentiation scheme to return the derivative at the center of an interval is to simply return the slope of the line connecting the two endpoints of the interval. Note that this is equivalent to performing linear interpolation on the interval and returning the derivative of the interpolant at the center of the interval.

One can write such a derivative as

$$f'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \tag{38}$$

for $x_i \le x \le x_{i+1}$. Similar to the linear interpolation method, however, we can motivate this derivative by making reference to Taylor expansions [3]. Let x be the center of the interval of interest, and let the interval's edges be $x \pm h$. Let's again consider the expansion of $f(x \pm h)$ about $f(x)$.

$$f(x \pm h) = f(x) \pm hf'(x) + \tfrac{1}{2}h^2 f''(x) \pm \tfrac{1}{6}h^3 f'''(x) + \mathcal{O}(h^4).$$

Instead of considering $f(x+h)+f(x-h)$, now we consider the difference, $f(x+h)-f(x-h)$. In this case, the even-degree terms will cancel and the odd-degree terms will sum. We find

$$f(x+h) - f(x-h) = 2hf'(x) + \tfrac{1}{3}h^3 f'''(x) + \mathcal{O}(h^4).$$

This can be solved for $f'(x)$, yielding

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \tfrac{1}{6}h^2 f'''(x) + \mathcal{O}(h^3).$$

Substituting y_i for $f(x-h)$, y_{i+1} for $f(x+h)$, and $x_{i+1} - x_i$ for $2h$, one can see that this formula reproduces the same formula as previously, but with the added information that this method carries truncation error $h^2 f'''(x)/6$, making it second order in h. This method is often also known as "central" or "three-point" finite differencing. We choose to call it a linear method, however, since it may also be easily represented as the differentiation of a linear interpolant.

B. Cubic method

In the previous section, we derived a differentiation method by simply taking the derivative of the linear Lagrange interpolant. Similarly, we may do the same for our cubic Lagrange interpolant. We begin with Equation 23, and differentiate with respect to x. This gives us an estimate for the derivative of $f(x)$ in the interval. The full equation, while a bit unwieldy, is

$$\begin{aligned} f'(x) \approx & \frac{(x-x_1)(x-x_2)+(x-x_1)(x-x_3)+(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} y_0 \\ & + \frac{(x-x_0)(x-x_2)+(x-x_0)(x-x_3)+(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} y_1 \\ & + \frac{(x-x_0)(x-x_1)+(x-x_0)(x-x_3)+(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} y_2 \\ & + \frac{(x-x_0)(x-x_1)+(x-x_0)(x-x_2)+(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} y_3 \end{aligned} \quad (39)$$

As before, we make the assumption that the spacing between all of the points is equal (and let this bin width be w), and let x be the midpoint between x_1 and x_2. Then, we find that the formula may be simplified significantly, yielding

$$f'(x) = \frac{1}{w}\left(\frac{1}{24}y_0 - \frac{9}{8}y_1 + \frac{9}{8}y_2 - \frac{1}{24}y_3\right) \quad (40)$$

Just as with cubic interpolation, we may also derive this formula by making reference to Taylor expansions. Let $h = w/2$, as usual, such that x_1 and x_2 are $x \pm h$ and x_0 and x_3 are $x \pm 3h$. We know the Taylor expansions for $f(x \pm h)$ and $f(x \pm 3h)$ about $f(x)$ (c.f. Equations 25, 26). Again, we can carefully choose a weighted sum of the four Taylor expansions, this time such that the $f'(x)$ terms survive and all the other order terms vanish. We take a cue from our above formula, and consider the following.

$$\begin{aligned} & 27(f(x+h) - f(x-h)) - (f(x+3h) - f(x-3h)) \\ & \quad = 48hf'(x) - \frac{18}{5}h^5 f^{(5)}(x) + \mathcal{O}(h^7) \end{aligned} \quad (41)$$

Solving for $f'(x)$, then, we obtain

$$\begin{aligned} f'(x) = & \frac{9}{16h}(f(x+h) - f(x-h)) \\ & - \frac{1}{48h}(f(x+3h) - f(x-3h)) \\ & + \frac{18}{5}h^4 f^{(5)}(x) + \mathcal{O}(h^6) \end{aligned} \quad (42)$$

Identifying the y_i values with the corresponding quantities and substituting $h = w/2$ gives us the formula we previously derived, with the added information that the truncation error is $18h^4 f^{(5)}(x)/5$, fourth order in h.

As with interpolation, however, this method faces issues at the first and last bins, where we do not have access to a value either before or after the interval. As such, we will again employ three-point quadratic Lagrange interpolation on these intervals and return the value of the derivative of the interpolant at the interval midpoint. The derivative of the quadratic Lagrange interpolant at the center of one of the two subintervals turns out to simply be the linear derivative described previously.

C. Cubic spline method

One can also compute the derivative using cubic spline interpolation. Recall that cubic spline interpolation yields a function $S(x)$ which is twice differentiable. This $S(x)$ is a piecewise combination of cubic polynomials $S_j(x)$, where S_j applies to all x such that $x_j \leq x_{j+1}$. This S_j has equation

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \quad (43)$$

where the coefficients are all fully solvable as described before. We can take the derivative of this, yielding

$$S_j'(x) = b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2, \quad (44)$$

We can simply return the value of this quadratic at the midpoint of the interval.

In the case of differentiation, we again have a constraint on the error. Assume $f \in C^4$ on the interval and let $h = \max_i(x_{i+1} - x_i)$. Then, [4] gives the following constraint

$$\max|f'(x) - S'(x)| \leq \frac{1}{24}h^3 \max|f'(x)|, \quad (45)$$

which implies uniform convergence of order $\mathcal{O}(h^3)$.

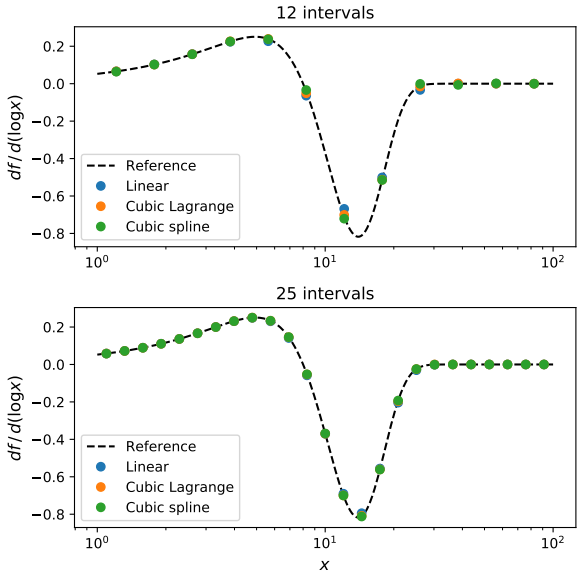

FIG. 5. Performance of our differentiation methods when calculating $df/d(\log x)$ at 12 and 25 interval midpoints for a Gaussian with mean $\mu = 8$ and width $\sigma = 5$.

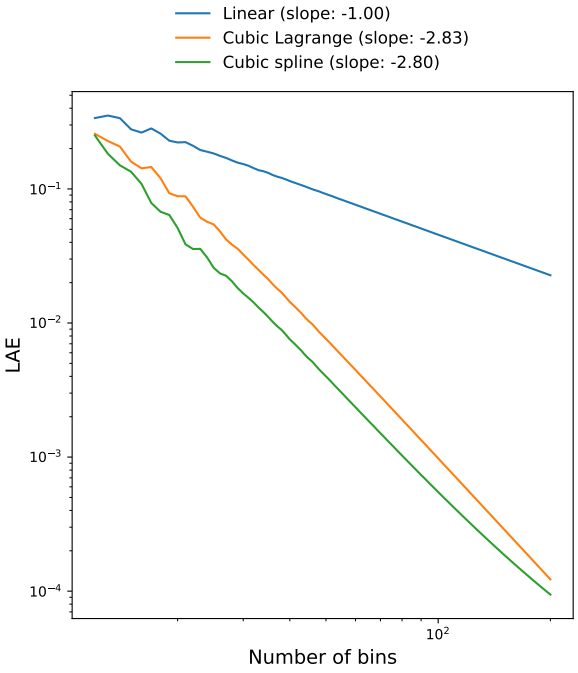

FIG. 6. Empirical error performance for our differentiation methods. The LAE is the average LAE for the method and number of bins when applied to Gaussians with means ranging from 5 to 25 and a width of 5.

D. Evaluation

As a preliminary test of these methods, let's return to our previous example. Consider again the Gaussian from Equation 36. We will sample this function at $n+1$ logarithmically spaced sampling points x_i, giving sampled values y_i. Then, we can use the previously described differentiation methods to determine the derivative of f at the midpoint of each interval. Notice how the solver has been converted into a form that makes use of only derivatives with respect to $\log x$, however. As such, we will feed our differentiators the points $(\log x_i, y_i)$. This ensures parity with our goal application of the differentiation methods *and* allows us to take advantage of the natural preference of these methods for equispaced sampling.

We can write the derivative of the Gaussian as

$$\frac{df}{dx} = -\frac{1}{2}\frac{x-\mu}{\sigma^2}\exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]. \tag{46}$$

From this, we obtain the derivative with respect to $\log x$,

$$\frac{df}{d\log x} = x\frac{df}{dx} = -\frac{x(x-\mu)}{\sigma^2}f(x) \tag{47}$$

We begin by simply performing this test for a few different numbers of sampling points and plotting the resulting interpolated values against the true distribution $df/d(\log x)$. These results can be found in Figure 5.

Next, we employ the least absolute error (LAE) to obtain a quantitative measurement of the accuracy of these methods. Following our method from Section III D, we will plot the logarithm of the error against the logarithm of the numbers of bins. Applying linear regression to the log-log data and taking the slope will give us a measurement of the order of convergence of the method.

In our testing, we choose a method and a set number of bins. We then compute the LAE for the derivative computed by the method for a range of 50 different Gaussians with means spanning linearly from 5 to 25, all with a width of 5. This is then repeated for many different bin counts ranging from 12 to 200 for each method. The resulting data is plotted in Figure IV D, with the computed slopes of the log-log data shown in the legend.

Again, we must note that these methods suffer from some instability for low numbers of bins. This is, as described previously, an issue of aliasing with our test distribution. Small bin numbers can place the sampling points at unfortunate positions, losing significant information about the test distribution and causing a loss of accuracy not seen for neighboring values. Our tests with many different means help to smooth this out, but the effect cannot be wholly erased.

The data suggest that the linear method sees roughly order 1 convergence, where both cubic methods see roughly order 3 convergence. This is in accord with our findings for the similarly-derived interpolation methods. In gen-

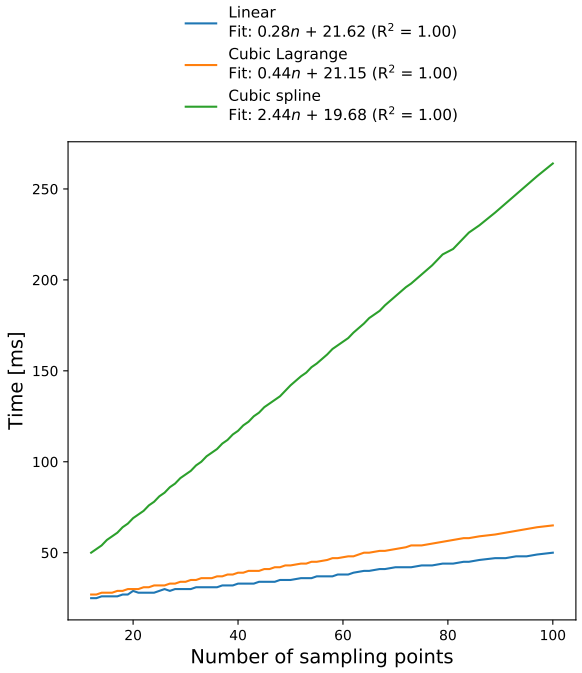

FIG. 7. Empirical timing data for our differentiation methods. These plots are shown with linear scaling (*not* log-log), and demonstrate that all considered methods have roughly linear dependence on the number of sampling points. Linear regressions have been performed; the best fit lines are given in the legend.

eral, it seems that the cubic spline method is the most accurate, but the performance gain over the cubic Lagrange method is small. In fact, for a small enough number of bins, the difference is negligible. As such, the decision as to which method better suits the needs of the solver will largely lie in the runtime performance of these methods.

We thus repeat our timing methods from Section III D for the differentiation methods. As before, we expect each method to have a strong linear dependence on the number of bins, as the implementations of each method are no more complex than `for` loops over the bins. For each number of bins, we make 1000 measurements of the amount of time it takes to compute the derivatives and average these measurements. This is repeated for many different values for the number of sampling points. The data is plotted in Figure 7.

The data show that all of the methods do indeed have a strong linear dependence on the number of bins, as all three methods have $R^2 = 1$ to better than a part in 10^3. Further, we see again the exact trends present in Section III D: the linear and cubic Lagrange methods have comparable performance, but the cubic spline method is nearly an order of magnitude slower. This runtime increase does not warrant the accuracy gains, especially at low numbers of bins where the error performance did not differ substantially to begin with. As such, we conclude that the best differentiator for our solver is cubic Lagrange differentiation as it gives nearly optimal runtime and accuracy, especially for low bin counts.

V. INTEGRATION METHODS

Also of importance our studies are the methods used for integration of the ODE given in Equation 16. For our discussion of these methods, we will instead, however, consider the ODE in a simple form. Suppose we have

$$\frac{dy}{dt} = f(t, y), \tag{48}$$

where f is a function which is known *a priori*. We desire methods which will advance y through a time step h according to this ODE. We typically use an index notation, reflecting that y has been discretized in time. Specifically, we will let $t_n = nh$ and take y_n to be $y(t_n)$. Thus, we desire methods which will advance y_n to y_{n+1}.

A. Euler's method

The simplest ODE integrator is Euler's method [5]. The method can be derived as follows. Consider the Taylor expansion of $y(t_{n+1})$ about t_n, with the time step called h.

$$y(t_{n+1}) = y(t_n) + h \left.\frac{dy}{dt}\right|_{t=t_n} + \mathcal{O}(h^2) \tag{49}$$

Replacing dy/dt with the right hand side (RHS) of the ODE gives

$$y_{n+1} = y_n + hf(t_n, y_n) + \mathcal{O}(h^2). \tag{50}$$

The first order update to y, then, is just the RHS evaluated at time t_n and multiplied by the step-size. This is known as Euler's method.

Euler's method is, however, known to be prone to inaccuracies. It is not symmetrical, as it uses derivative information only at the beginning of the time interval $[t_n, t_n + h]$. The error of this method is only $\mathcal{O}(h^2)$, making it a "first order" method.[3] More generally, though, the method is known to have issues with stability, especially in cases of oscillatory function behavior.

B. Runge-Kutta methods

Runge-Kutta methods do not differ substantially from Euler's method for solving the ODE. They are in some

[3] A method is often called nth order when its local error term is $\mathcal{O}(h^{n+1})$, because it accurate represents the function up to nth order.

ways higher order extensions of Euler's method, which compute trial steps to points along the interval $[t, t+h]$ to improve the accuracy. Of course, this comes with the cost of additional computations, increasing the overall runtime.

We can derive the *second-order* Runge-Kutta method, which we will call RK2, by taking a trial Euler step to the midpoint of the time interval [6]. At the midpoint, we compute dy/dt and then use this derivative information to take the full step through to $t+h$. Mathematically,

$$\begin{aligned} k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 + \mathcal{O}(h^3). \end{aligned} \tag{51}$$

This method "symmetrizes" the time step calculation, as it only uses derivative information from the center of the time interval when computing the final update. In doing this, we have canceled out the $\mathcal{O}(h^2)$ error, hence the name "second-order."

We will also consider the *fourth-order* Runge-Kutta method, RK4. This method computes the derivative four times: once at t_n, twice at $t_n+\frac{1}{2}h$, and once at t_n+h. The information from all four of these derivative calculations is used to cancel out various error orders. Mathematically,

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(h^5). \end{aligned} \tag{52}$$

Since the most important pieces of the pipeline are the interpolation and differentiation methods, we move forward with our evaluation of the solver using Euler's method, since it is the fastest and simplest to implement. However, the error performance and runtime of these methods will be considered in Section VI H.

VI. EVALUATION

In this section, we evaluate the performance of the solver on the evolution of a simple Gaussian distribution (mean 10, width 3), sampled at logarithmically-spaced points ranging from 1 MeV to 100 MeV. Unless otherwise specified, we use the cubic methods for interpolation and differentiation of f, the linear method for differentation of I_ν, and Euler's method for advancing the solver through a time step. Note that due to instabilities caused by very small values of $f(x)$, we impose that the distribution must at all times have a minimum value of 10^{-30}, i.e. for every step, we take $f(x) = \max(f(x), 10^{-30})$.

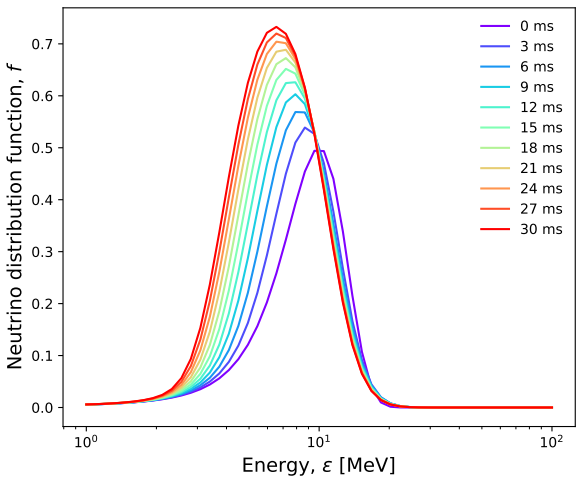

FIG. 8. Evolution for 0.03 seconds with a fixed step size of 10^{-6} s, using parameters $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and $Y_e = 0.2$ and a binning scheme with 50 bins whose zone centers range from 1 to 100 MeV.

Recall as well our discussion in Section I of the steady-state behavior of this system. In general, as we evolve the distribution forward in time, we expect it to converge to a Fermi-Dirac distribution with an electrochemical potential μ which depends on the initial distribution and the nucleon temperature T. The tests we perform in this section will serve as evidence for this convergence, as well as how the steady-state behavior depends on the physical and model parameters. Recall as well that the convergence to the final steady-state is expected to be significantly slower at low energies than at high energies due to the x^6 dependence of I_ν.

A. Fixed versus adaptive time-step evolution

The simplest way to integrate the solver forward in time is to set a fixed step size. We begin with a step size of 1 μs. We also pick some physical values for the kT (nucleon temperature), ρ_N (nucleon mass density), and Y_e (nucleon electron fraction) and integrate the distribution. To begin, we choose $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and $Y_e = 0.2$. We also choose to use an initial neutrino distribution function f of a Gaussian with mean 10 and width 3. We use 50 bins in energy, and we integrate the distribution forward through 30 000 time steps. With a step-size of 1 μs, this is a total integration time of 0.03 seconds. The resulting evolution of the distribution can be seen in Figure 8.

There is an issue with fixed step-size integration, however. A fixed time step can be simply too small for some regions where the derivative does not change much, or too large for regions where the derivative changes quickly. Thus, it would be wise to add some form of step-size control.

As such, we have implemented a rudimentary step-size controller. For each step, we compute the fractional change to f across all the energy bins and find the maximum. We can then compare this to a desired fractional change, ξ, and adjust the step-size for the next iteration. Given the maximum observed fractional change, $\max(\Delta f/f)$, we can write the step-size that *would* have limited the fractional change of the current step to be ξ as $dt' = dt\,\xi/\max(\Delta f/f)$.[4] In principal, we could then integrate the next step forward with a step-size of dt' and continue to adapt the step-size in this way.

Unfortunately, such a method proves to be somewhat unstable because the maximum fractional change also varies with time. This instability can be dampened by introducing a maximum allowed change to dt in a given time step. Call this maximum allowed change dt_m. Then, we clamp dt' to lie in the interval $[dt - dt_m, dt + dt_m]$. The effect of this change is to prevent the step-sizes from changing more quickly than $\max(\Delta f/f)$ and thereby eliminate instabilities. In practice, it was found that $dt_m \sim 10^{-9}$ s helps to ensure the smooth convergence of the step-size to a stable value.

With this step-size controller, we now run the same experiment as before. Instead of specifying a number of steps, we specify a total integration time, computed by simply summing the values of dt that are used as we go. With the same physical parameters as previously, we integrate through 0.03 seconds, setting our desired change $\xi = 0.0001$, i.e. f should not change in any bin by more than 0.01% on any step. It is also instructive to understand the step-sizes used by the solver during this time. As such, for every step we wrote out the step-size which was used and make a plot. The evolution of the distribution and the step-sizes used can be seen in Figure 9.

These time-steps are show that at the very beginning, the solver is computing very large updates to the distribution that must be mediated by very small time steps. As the evolution continues, however, the computed updates very quickly become much smaller, permitting larger step sizes. Overall, though, we can see that a fixed step size on the order of $\sim 10^{-9}$ to 10^{-6} s is generally appropriate, given that the adaptive step size generally stays around this value. Further, even in regions where the adaptive step size is quite small, the general shape of the distribution remains unaffected by using larger steps, indicating that the time derivative does not fluctuate very much. This all indicates that using a fixed step size is generally accurate.

[4] Given that we are using Euler's method for integration, the update to f varies linearly with dt. Thus, it holds that dt' as written would have limited the maximum fractional change to ξ.

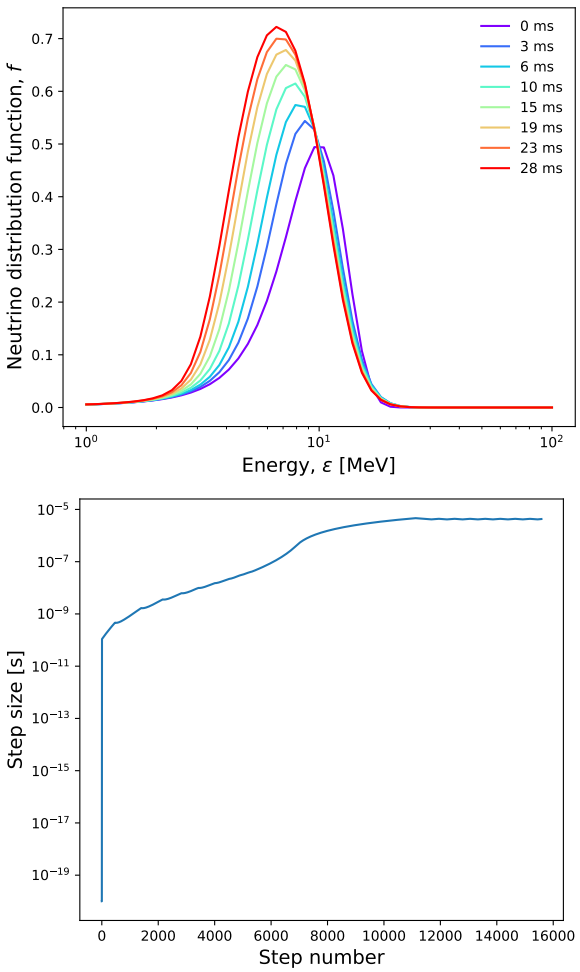

FIG. 9. (Top) Evolution for 0.03 seconds with an adaptive step size, using parameters $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and $Y_e = 0.2$ and a binning scheme with 50 bins whose zone centers range from 1 to 100 MeV. (Bottom) The time step sizes chosen by our adaptive step size scheme, using a desired fractional change of 0.01%.

B. Comparison to past work

As a check on the accuracy of our work, we consider an article by Thompson et al. that presents the expected time evolution of a similar distribution through the mechanism of mu neutrino and neutron scattering [7]. We focus specifically on their Figure 2. In their work, the initial distribution has a mean of 40 MeV, width of approximately 10, and a height of approximately 0.8. We note that our solver by default considers neutron and proton interactions; however, we have introduced a switch in the code that allows one to turn off the contributions of one or the other at will. As such, we turn off protons for this section.

The physical parameters considered in the paper were $kT = 14.51$ MeV, $\rho_N = 10^{13}$ g/cm^3, and $Y_e = 0.202$. Note that these the largest temperatures and densities

to be considered in this work. For our solver, we choose to use 100 bins in energy space spread between 1 and 100 MeV. We evolve our distribution forward using adaptive step-size with a desired fractional change of 0.005% and no maximum change in step-size per timestep. Such a small desired change was found to be sufficient in this case to avoid the instabilities previously described.

The evolution of the previously described distribution over approximately 1 millisecond from [7] is reproduced alongside the same evolution computed by our solver in Figure 10. These plots show that our solver performs exactly as expected given the reference distribution with the general shape, spread, and amplitude of the distribution reproduced quite faithfully by our solver at all time stamps.

C. Impact of energy binning

One specific aspect of the performance of the solver that we need to evaluate is how it relates to the specifics of the energy binning. There are two aspects to this worth considering: the size of the energy bins and the energy range covered by the bins.

Let's begin by considering a relatively standard case of energies ranging from 1 to 100 MeV. All the methods we use guarantee convergence to the true function behavior in the small bin-size (large number of bins) limit, but the method needs to remain accurate even when there are relatively few bins. As such, we can take a set of physical parameters—say $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and $Y_e = 0.2$, as before—and integrate it forward in time for about 0.03 s using various numbers of energy bins between 1 and 100 MeV. The numbers of bins chosen were 12, 25, 50, and 100. We choose to perform this time evolution with a fixed step-size of 1 μs, since we saw in the previous section that it gave qualitatively similar results for this length of integration. It will also allow us to more easily compare the different bin counts after the *exact* same amount of integrated time. In Figure 11, one can see the time evolution of f for each of these numbers of bins. Further, in Figure 12, we plot the evolution of the 100 bin distribution as a solid line and the evolution of the 12 bin distribution with circular markers.

In these plots, one can see that we generally retain a fairly faithful representation of the distributions as the number of bins decreases (and the size of the bins increases). Toward very small numbers of bins, however, we begin to lose accuracy, in large part it seems because of an increasing inability to accurately represent the initial distribution given the sampling points.

We can quantify the agreement between the low and high numbers of bins by considering the least absolute error. For this test, we take the distribution with 100 bins to be the "reference". For each smaller bin count, we compare the distribution values to the interpolated value of

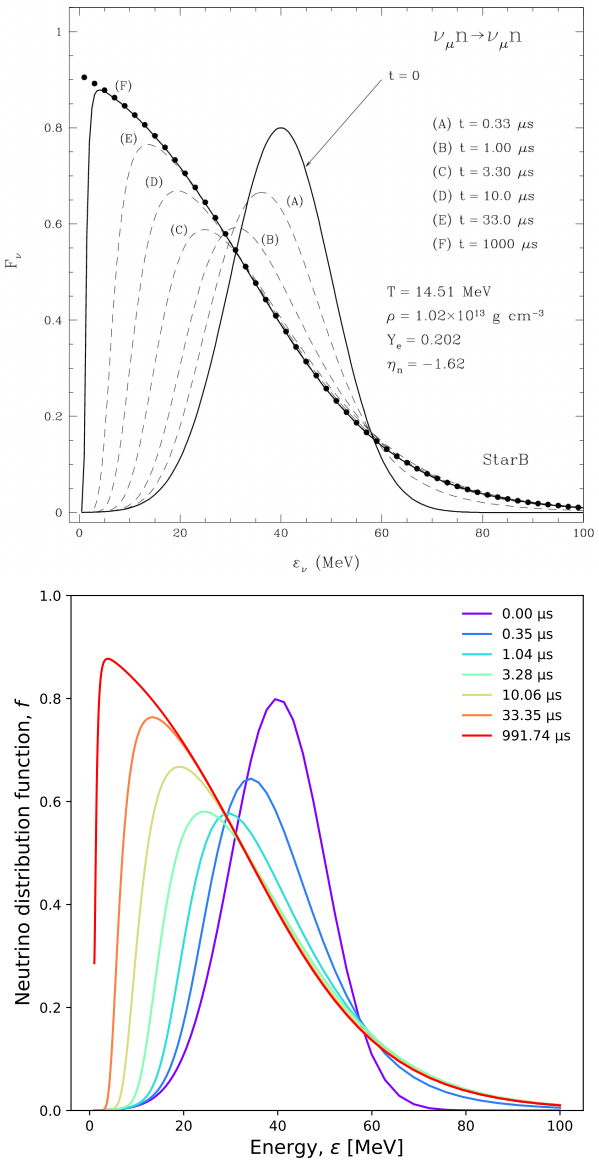

FIG. 10. Time evolution of the neutrino distribution function through neutrino-neutron interactions over 1 millisecond. (Top) Reference evolution reproduced from Figure 2 of [7]. (Bottom) Evolution computed by our solver pipeline. Timestamps for our evolution are approximately the same as those in the reference plot up to around one percent.

the reference distribution at the same time and compute the LAE. We then take the logarithm of both the error and the bin count and plot the log-log data, much as we did in previous sections. The resulting plot is shown in Figure 13.

The plot shows that error increases somewhat as the time evolution continues, though its increase seems to slow. Further, we see a slope of roughly -1 on the log-log plot, indicating roughly linear convergence with increasing bin count. This is consistent with our expectation. The worst error convergence in the solver is approximately linear,

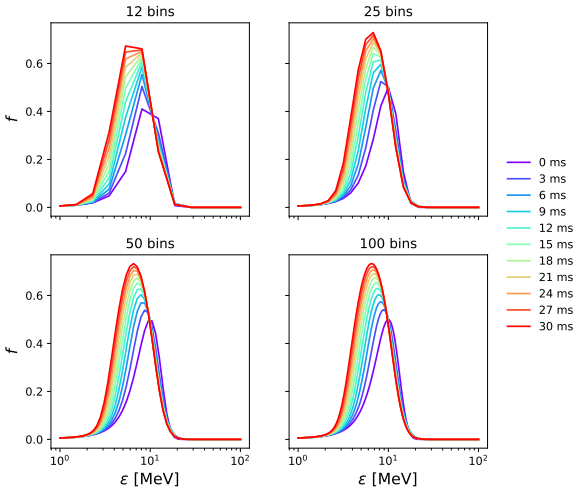

FIG. 11. Evolution of a Gaussian distribution for 0.03 seconds using a fixed stepsize of 10^{-6} seconds and a varying number of bins. Convergence to an ideal function with growing number of bins is evident. Further, we see that the lower bin counts provide a mostly accurate representation of the high-bin count case.

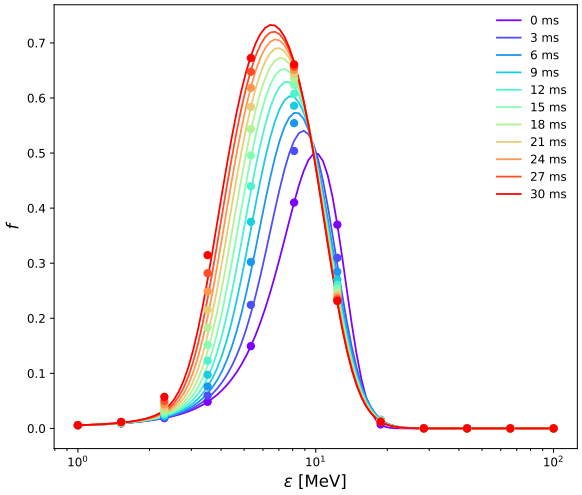

FIG. 12. Evolution of a Gaussian distribution for 0.03 seconds using a fixed stepsize of 10^{-6} seconds and two numbers of bins: 12 and 100. (This is the same data as in Figure 12.) The 100-bin data are plotted as smooth curves, while the 12-bin data are plotted as circular markers. Note that the x-axis is log-scaled in this plot to more clearly show the agreement between the two bin counts at low energies.

since we choose to use the linear differentiator when evaluating $dI_\nu/d(\log x)$ at the end of the solver. We will expand these error considerations in Section VI F.

We can also consider how the solver would perform if we were to extend the size of the energy range upward to around 300 MeV. The primary effect this has on the solver is that all of the energy bins become *even larger* for the same number of energy bins. To test the performance, we keep the same physical parameters from the previous tests and evolve 12 and 100 bins forward in time. We then plot these on the same axes, where the 100 bin run is plotted as a solid line and the 12 bin run is plotted as circular markers. This plot can be seen in Figure 14.

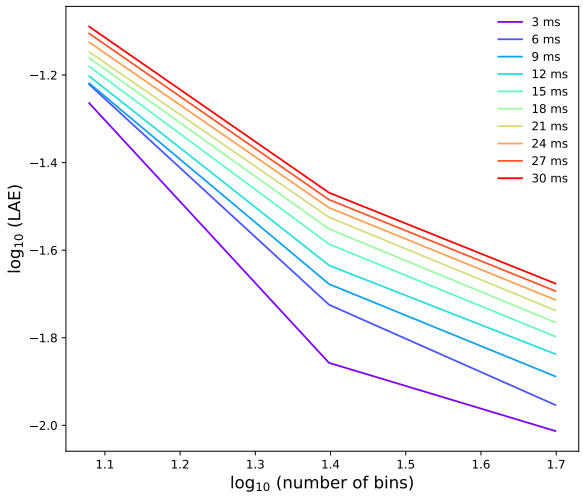

FIG. 13. Least absolute error (LAE) with respect to number of bins. The error is computed against the 100 bin distribution for each time step.

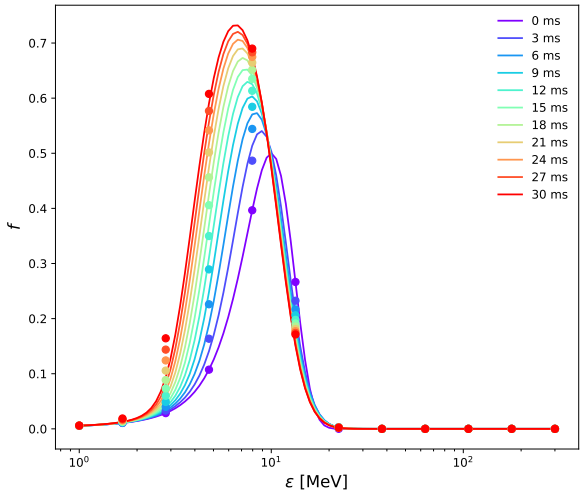

FIG. 14. Evolution of a Gaussian distribution for 0.03 seconds using a fixed stepsize of 10^{-6} seconds and two numbers of bins: 12 and 100, ranging from 1 to 300 MeV. The 100 bin data are plotted as smooth curves, while the 12 bin data are plotted as circular markers. Even though the bins are spaced out more than in previous tests, the small number of bins is still able to faithfully reproduce the shape and behavior of the function over time.

Here again we can see that the 12 bin scheme is able to faithfully reproduce much of the shape and behavior of the distribution over time, with some error introduced in areas where the binning simply does not very well capture

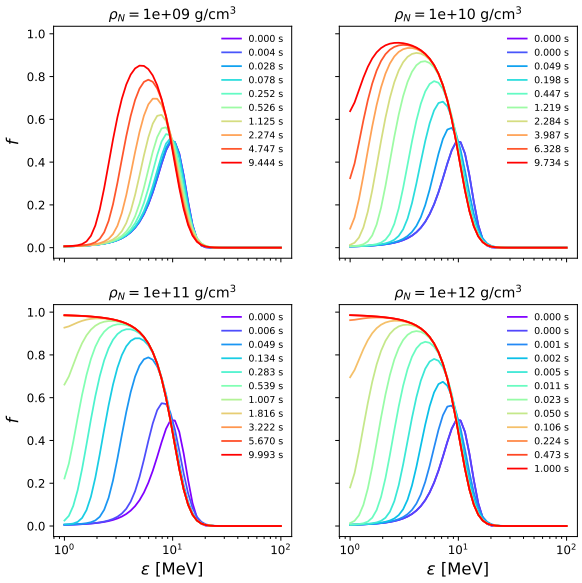

FIG. 15. Evolution for 10 seconds of a Gaussian using 50 energy bins with physical parameters $kT = 2$ MeV and $Y_e = 0.2$, and ρ_N varying with values $\{10^9, 10^{10}, 10^{11}, 10^{12}\}$ g/cm^3. The adaptive stepsize scheme was used.

the initial distribution.

D. Impact of physical parameters

It is also instructive to study the impact of varying physical parameters on the time evolution of the distribution. The first parameter we will look at is the nucleon mass density, ρ_N. This parameter comes into the solver pipeline in the coefficient α of I_ν. We find that updates to f should scale linearly with α, and thus linearly with ρ_N as well. As such, we expect increasing the density to increase the size of the updates to the f distribution, reflecting an increased frequency of neutrino-nucleon interactions. For the adaptive step-size control implemented above, this will amount to smaller steps being taken. In terms of integration time, however, it means faster convergence to the equilibrium distribution.

To test the above hypothesized behavior, we performed evolutions with $kT = 2$ MeV and $Y_e = 0.2$ as before, but with ρ_N taking values $\{10^9, 10^{10}, 10^{11}, 10^{12}\}$ g/cm^3. We also choose to use 50 bins to provide sufficient energy resolution of the distribution. For each value of ρ_N, we then evolved the distribution forward with adaptive step-size for a total integration of approximately 10 seconds. The evolutions can be seen in Figure 15.

From these plots, we can see that the higher densities indeed converge much faster than the small ones. This is particularly easy to see at the lowest energies, where the $\rho_N = 10^9$ g/cm^3 distribution appears to seriously struggle to obtain convergence but the $\rho_N = 10^{11}$ g/cm^3 distribution obtains convergence much more quickly. Notice further that the distribution after 10 seconds with $\rho_N = 10^9$ g/cm^3 is approximately equivalent to the distribution after around 1 second with $\rho_N = 10^{10}$. This phenomenon is notable as well with 10 seconds with $\rho_N = 10^{11}$ g/cm^3 compared to 1 second with $\rho_N = 10^{12}$ g/cm^3. This serves as evidence that the evolution of the neutrino distribution function is indeed linearly dependent on nucleon density.

However, this faster convergence comes at the cost of significantly smaller step-sizes when using the adaptive step-size controller. This is because the update to the distribution goes up by the same factor as the density. The adaptive step-size controller regulates the update size, though, cutting the time step size down by a factor of ten to accomodate. For significantly higher densities, this causes the integration to take much longer. This is the reason why the $\rho_N = 10^{12}$ g/cm^3 run was stopped after only one second.

The next parameter we will look at is the electron fraction, Y_e. It is similarly simple to identify where Y_e comes into the solver pipeline: it is in the coefficient to I_ν again. It acts as a linear switch between the neutron and proton coefficients for $V^2 + 5A^2$, as well as appearing in the calculation of the correction λ. To first order, however, we expect the updates to vary linearly with Y_e through its impact on the $V^2 + 5A^2$ term. Specifically, the $V_p^2 + 5A_p^2 \approx 2.02$ and $V_n^2 + 5A_n^2 \approx 2.27$, and, since larger Y_e increases the proton contribution relative to the neutron contribution, we expect the overall coefficient to decrease slightly with increasing electron fraction. However, given the similarity between the proton and neutron coefficients, the effects should not be very large.

To test this hypothesis, we let $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and then take Y_e values of $\{0.1, 0.2, 0.3, 0.4\}$. For each value, we integrate forward in time for a total integration of several seconds and plot the resulting evolution of the distribution. The plots may be found in Figure 16. The four are remarkably similar, supporting our hypothesis before that varying Y_e would have mostly negligible effects on the evolution of the distribution.

The last parameter we consider is the nucleon temperature kT. This has been kept for last because its interaction in the solver is more complicated. There are two main places where the influence of kT will be felt: the first is through α, which has a cubic dependence on kT through the $1/\beta^3$ factor. The second, however, is in x, the dimensionless energy parameter, as $x = \varepsilon/kT$. Through its presence in α, we predict that larger values of kT will yield larger updates to f. This intuitively makes sense, as raising the temperature of the nucleons should increase the frequency of scattering. Through its presence in x, though, we expect the very shape of the distribution to change. In particular, we expect larger kT values to give larger chemical potentials. In the large kT limit, we expect the distribution to slowly approach

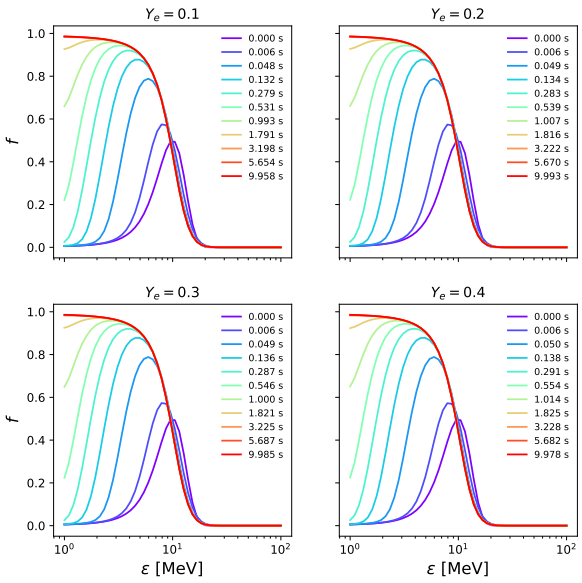

FIG. 16. Evolution for 10 seconds of a Gaussian using 50 bins with physical parameters $kT = 2$ MeV and $\rho_N = 10^{11}$ g/cm^3, and Y_e varying with values $\{0.1, 0.2, 0.3, 0.4\}$. The adaptive stepsize scheme was used.

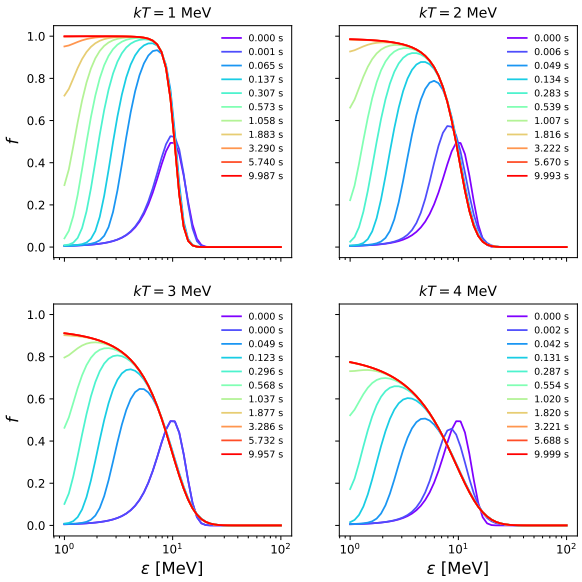

FIG. 17. Evolution until (approximate) steady-state behavior of a distribution using 50 bins with physical parameters $\rho_N = 10^{11}$ g/cm^3, $Y_e = 0.2$, and kT varying with values $\{1, 2, 3, 4\}$ MeV. The adaptive stepsize scheme was used.

a Maxwell-Boltzmann distribution.

To test this, we let $\rho_N = 10^{11}$ g/cm^3, $Y_e = 0.2$, and we consider kT with values $\{1, 2, 3, 4\}$ MeV. For each such value, we integrate forward several seconds using the adaptive step size algorithm. The evolutions can be seen in Figure 17.

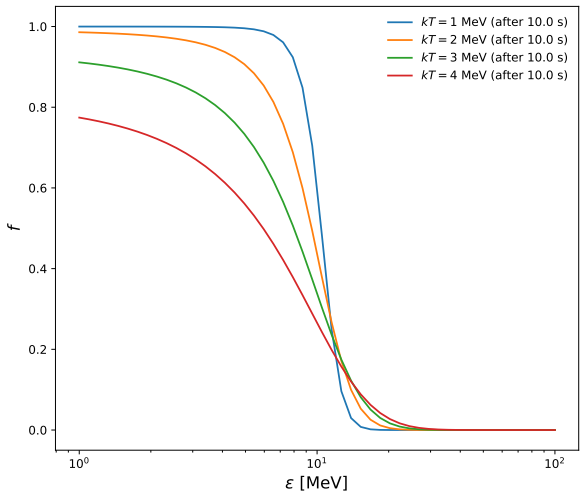

FIG. 18. The last step taken for the distributions from Figure 17. We can directly compare the steepness of the transition for varying nucleon temperature to see that higher temperatures give much less steep transitions.

From these plots, we can see that higher nucleon temperature does indeed yield faster convergence to the final distribution. This is especially clear for the energy bins near 1, where the lower values of kT struggle to adjust the curve upward but higher values of kT see these bins increasing quickly along with the surrounding energy bins. We also see that the general shape of the final result is a bit different, with higher values of kT carrying with them a wider and less-steep slope as the distribution transitions from values near 1 to values near 0. This is the result of a higher electrochemical potential in the Fermi-Dirac distribution.

In the specific case of varying kT, it is interesting to run the integrations until steady-state behavior and look at only the end distribution. Given the different rates of convergence and the corresponding adaptive step sizes, we ran the values for different amounts of total integration time, and note that the distributions shown are only approximately "steady-state". The distribution in the final step of each integration is then plotted on the same axes. This can be seen in Figure 18.

In this plot, the differences in shape are much more evident between the different values of kT. In particular, it is very easy to see that the steepness of the transition from 1 to 0 decreases sharply with increasing nucleon temperature. Further, it is also slowly approaching the shape of a Maxwell-Boltzmann distribution in much the same way as we would expect from Fermi-Dirac statistics at high energy.

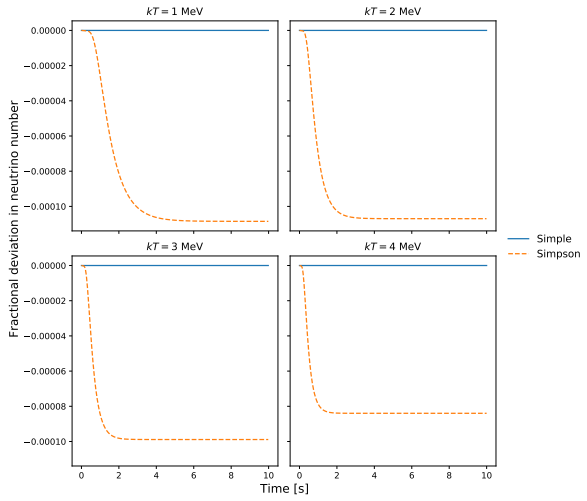

FIG. 19. Fractional deviation from initial neutrino number over time for a ten second integration with 50 energy bins, $\rho_N = 10^{10}$ g/cm^3, and $Y_e = 0.2$. We also vary kT with values $\{1, 2, 3, 4\}$ MeV.

E. Conservation of neutrino number

Another check that is worth performing on our solver is whether the neutrino number is truly being conserved. As discussed previously, I_ν acts as the flux in energy space of neutrino number, so the total change in neutrino number for any given time step will be given by the difference between I_ν on the upper boundary of the largest energy bin and on the lower boundary of the smallest energy bin. Given that both of these are manually set to 0 in our code, we should have mathematically ensured that no neutrinos can escape out of the energy range or be created or destroyed. However, all of this does not mean that the introduction of errors due to floating point precision, rounding errors, or other problems could ruin our neutrino conservation.

Thus, we desire to test whether neutrino number truly is conserved. To do so, we take the integrations that were run with varying kT and ρ_N previously and compute the integral of $x^2 f(x)$ for each time step. The integral is computed using two different methods: "simple", equal to $\sum x^2 f(x) \Delta x$, and "simpson", which uses Simpson's rule.[5] We then plot the fractional deviation from the initial value over time. If we let the neutrino number at time step i be n_i, this is $(n_i - n_0)/n_0$. The resulting plot may be seen in Figure 19.

These plots show that neutrino number is exactly conserved when it is computed using the "simple" scheme. This is as we would expect, since our algorithm was designed in such a way as to automatically conserve this quantity. However, when using the higher-order "simpson" integration scheme, we can see that there is a slight deviation from the initial value. This deviation converges to a value which differs from the initial by only approximately 0.01%, serving as evidence that our solver does, in fact, conserve neutrino number to good precision.

F. Low bin count performance

We would also like to quantify the error of our numerical methods in the context of the full solver pipeline. As such, we begin by choosing $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, and $Y_e = 0.2$ as our physical parameters, and we choose a variety of numbers of bins on which to test our solver. We are most interested in low bin counts, so we will test numbers ranging from around 10 to 25.[6] For each bin count, we integrate a simple Gaussian distribution forward 1 000 steps with a fixed step size of 1 μs. Then, we take the final state of the distribution and compare it to a reference distribution with 100 bins.

We then repeat this experiment for several different combinations of interpolation and differentiation methods. In general, we choose to use the related methods for interpolation and differentiation of f. We found the cubic spline method to be very unstable for low bin counts when used on f, so we only consider the other two methods—linear and cubic Lagrange—for f. Our prior tests showed that the cubic spline method is also horribly slow by comparison with the other methods, so avoiding its double usage (both for interpolation and differentiation of f) in a single pass is advantageous from a runtime perspective as well. For I_ν, however, we consider all three methods. Note that because we are only considering the performance for rather small bin counts, we do *not* expect to see convergence behavior consistent with the large bin limit.

For each set of numerical methods, we compare to a standard reference distribution when computing the least absolute error (LAE). The reference distribution uses 100 bins, cubic Lagrange methods for f, and linear methods for I_ν. This combination was chosen because the timing performance is good, the neutrino conservation is strongly conserved by the use of linear differentiation methods, and the error performance is among the best (as will be seen). The resulting data is shown in Figure 20.

From these plots, we can draw a few conclusions about the low bin count behavior of our numerical methods.

[5] We are simply using the method `scipy.integrate.simps` from SciPy version 1.3.1.

[6] Note that we no longer have a purely analytic description of the true distribution, so our error considerations must be made relative to a reference distribution. This prohibits us from effectively considering the high bin count convergence as was explored for the interpolative and differential methods in previous sections.

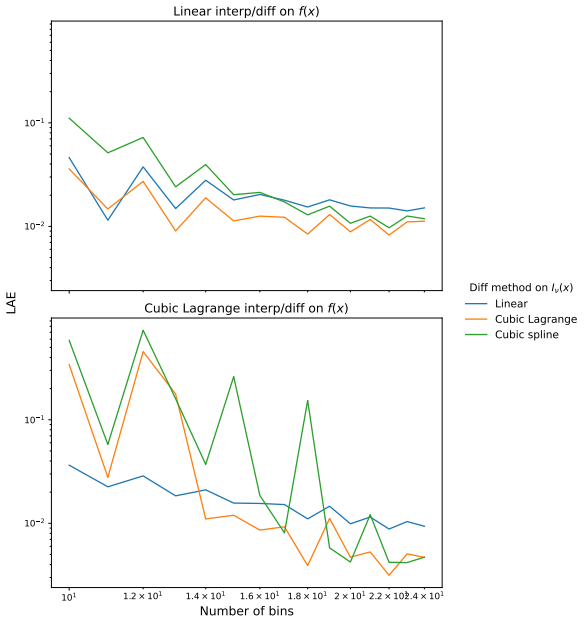

FIG. 20. Least absolute error as a function of number of bins for small bin counts. Errors are calculated by comparison with a 100 bin reference distribution which used cubic Lagrange methods on f and linear methods on I_ν. Base-10 logarithms have been applied ot the data on both axes.

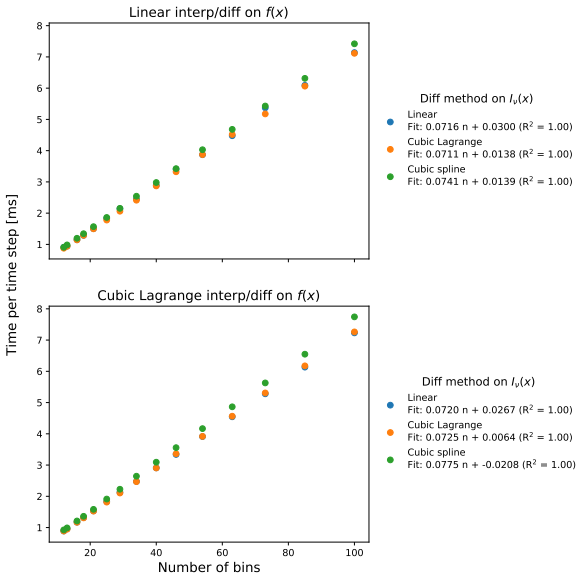

FIG. 21. Solver runtime performance for linear and cubic Lagrange methods for f, all three differentiation methods for I_ν. The runtime is given in milliseconds per step as a function of the number of bins in the energy binning scheme. All the data show a very strongly linear dependence on the number of bins, with best fit lines and R^2 values printed in the legends.

First, the use of higher order methods for both f and I_ν results in less stable behavior. When there are relatively few sampling points, the higher order methods are able to describe the main bell-shape of the Gaussian distribution decently well, but tend to introduce curvature to the shape of the interpolated distribution between points near zero at high energies. The linear method is inherently robust to this phenomenon, and using it at either stage (on f or on I_ν) seems to alleviate this problem.

Another conclusion to draw is that all of the runs with any linear part exhibit similar performance. The errors of all these runs are contained within a similar range, and appear to have roughly equal slopes. Since the slope of data on a log-log plot like this acts as a measure of the convergence order, we can say that all the runs have roughly equal convergence behavior.

Lastly, these plots indicate that the cubic f, linear I_ν method appears to be one of the strongest performers overall in this regime. In addition to having one of the lowest errors for the smallest bin counts, it also shows a general sense of stability unseen in any of the other runs. This is why we have chosen this combinations of methods specifically to serve as the default in the solver and the reference distribution for this section.

G. Runtime performance

We also seek to understand the general runtime performance of the overall pipeline. To do so, we evolve the $kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, $Y_e = 0.2$ distribution forward with fixed $dt = 10^{-6}$ s for a few thousand steps and time how long this task takes to complete. We then divide by the total number of steps integrated to obtain an estimate of the amount of time it takes to integrate a single step. This is repeated for varying numbers of bins, which allows us understand the runtime performance as a function of the binning.

Further, we can also run these tests using the other interpolation and differentiation methods that were considered in the last few sections to test whether the runtime performance found therein holds true for the full solver pipeline. In particular, we adopt the same combinations of methods as were considered in the previous section: linear and cubic Lagrange methods for the interpolation and differentiation of f, and all three differentiation methods for I_ν. The resulting timing data is plotted in Figure 21.

The data show a very strictly linear dependence on number of bins, so linear regression has been applied to the data and the slopes of the best-fit lines are given in the legend. We can see that our choice of cubic methods for f and linear differentiation for I_ν gives one of the fastest runtimes possible (out of the different combinations of considered methods). This is fully in support of our find-

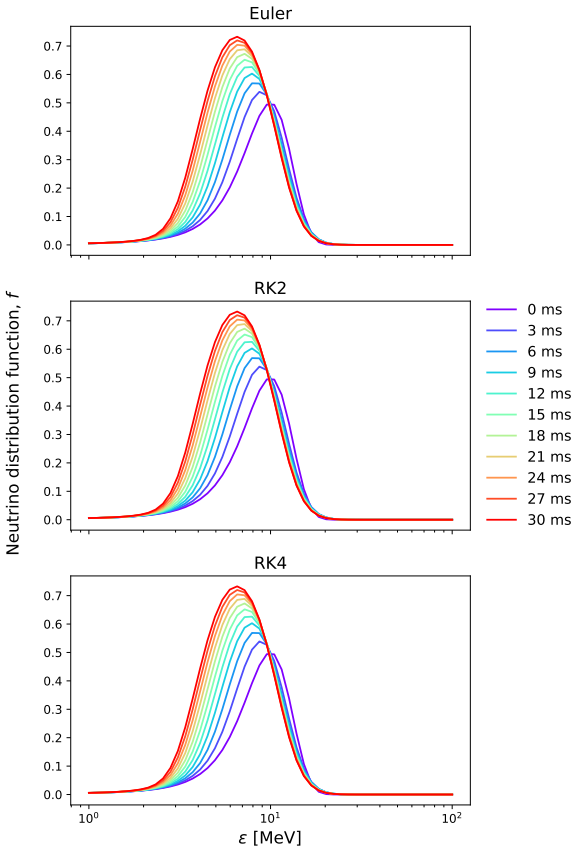

FIG. 22. Time evolution through 30 milliseconds with all three considered ODE stepping methods. The ODE stepper has very little effect on the evolution of the distribution.

ings from the individual method tests from before.

H. ODE steppers

To round off our evaluation of the pipeline, we also swap in the three time-stepping algorithms considered previously: Euler, second order Runge-Kutta (RK2), and fourth order Runge-Kutta (RK4). For all the following runs, we use our standard physical parameters ($kT = 2$ MeV, $\rho_N = 10^{11}$ g/cm^3, $Y_e = 0.2$) with an initial Gaussian distribution with mean 5 and width 3.

We first evolve the distribution forward for 0.03 seconds with 1 μs fixed time steps and 50 bins. The resulting time evolution of the distribution is shown in Figure 22.

We find that the ODE stepping method appears to have very little effect on the time evolution of the distribution, especially when considering relatively good energy resolution as we have here with 50 bins. This leads us to consider the performance of these methods for different bin counts. As such, we perform this experiment again, but with a variable number of bins.

We choose 100 bins to again serve as a reference distribution. For each ODE stepper and each number of bins, we compute the LAE with respect to the 100 bin distribution of *the same stepper*. The resulting data are plotted in Figure 23.

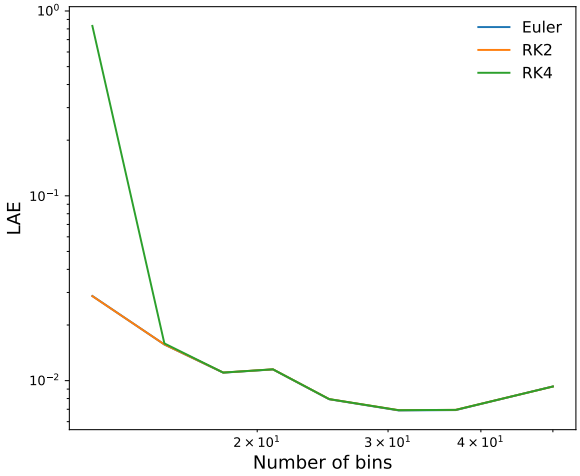

FIG. 23. Least absolute error for the distribution after 1 000 one-microsecond steps with n bins compared against a 100 bin reference. Note that the Euler and RK2 data overlap.

We see very similar error performance for all bin counts except 12, where the RK4 method incurs a significantly higher error than the other methods. Note that the Euler and RK2 data overlap signficantly. These findings imply that the ODE stepper does indeed have very small impact on the evolution of the distribution, except for the specific case of 12 bins with fourth-order Runge-Kutta.

We can also consider the runtime of the different ODE methods. Since the performance of all the methods is so similar, whichever is fastest is likely the best for our case. The average time per step, computed over 1 000 trial steps, is plotted in Figure 24. We see that all methods are strongly linear in the number of bins. Also, from the slopes of the lines, we can see that the Euler method is the fastest method, with RK2 the next fastest and RK4 the slowest. This makes sense analytically, since the Euler method does the fewest computations and RK4 does the most.

The last consideration we would like to make regarding ODE methods is the impact of the stepper on the adaptive step-size. To carry out this test, we evolve the same starting distribution forward for 0.01 seconds with 50 bins, a desired fractional change of 0.5%, and a maximum allowed change in step-size of 1 ns. For each time-step, we record the new step-size and plot it over time. The data can be found in Figure 25.

We see that the data for all three steppers strongly overlaps. As such, we instead consider the fractional difference between each pair of steppers. In other words, for each step taken, we compute $(dt_{RK2} - dt_E)/dt_E$ (and the

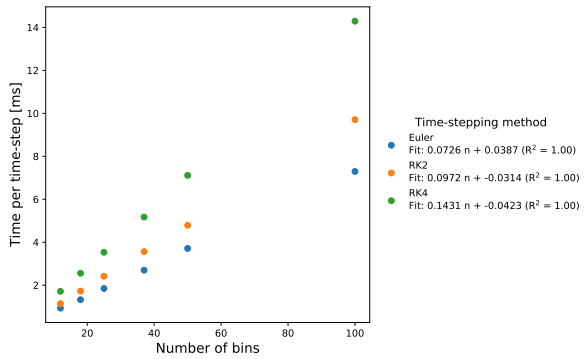

FIG. 24. Runtime per time-step for each ODE method as a function of number of bins. All steppers show a strongly linear dependence on the number of bins, with R^2 values equal to 1 to within a part in 1 000.

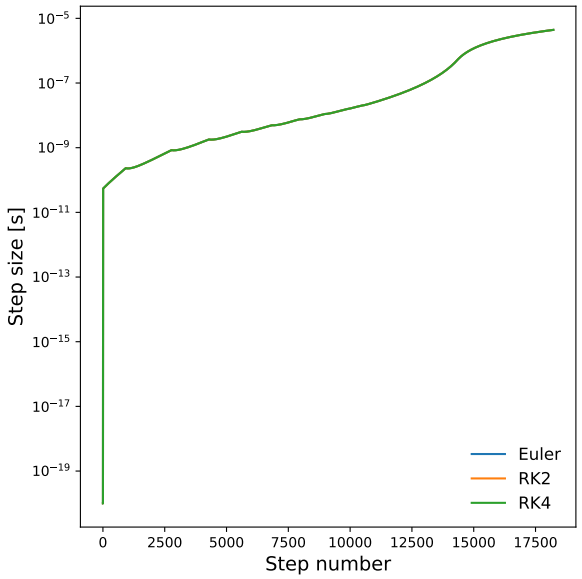

FIG. 25. Time step-size versus step number of a 0.01 second integration with the three considered ODE steppers. The data for all three steppers strongly overlap.

same for (dt_{RK4}, dt_E) and (dt_{RK4}, dt_{RK2})). These data are shown in Figure 26.

There are strange downward "spikes" shown in the first few thousand steps which are quite evident in all these fractional difference plots. These appear to correspond exactly to some very slight discontinuities that are visible in Figure 25. The presence of these spikes appears to be indicative that a rather sudden change to the step-size was needed by solver and that this change exacerbated the magnitude of the fractional differences between the step-sizes of the different steppers.

Overall, however, these data show that the adaptive time step-sizes computed for each ODE stepper agree to within 1% across all time steps considered. Each Runge-

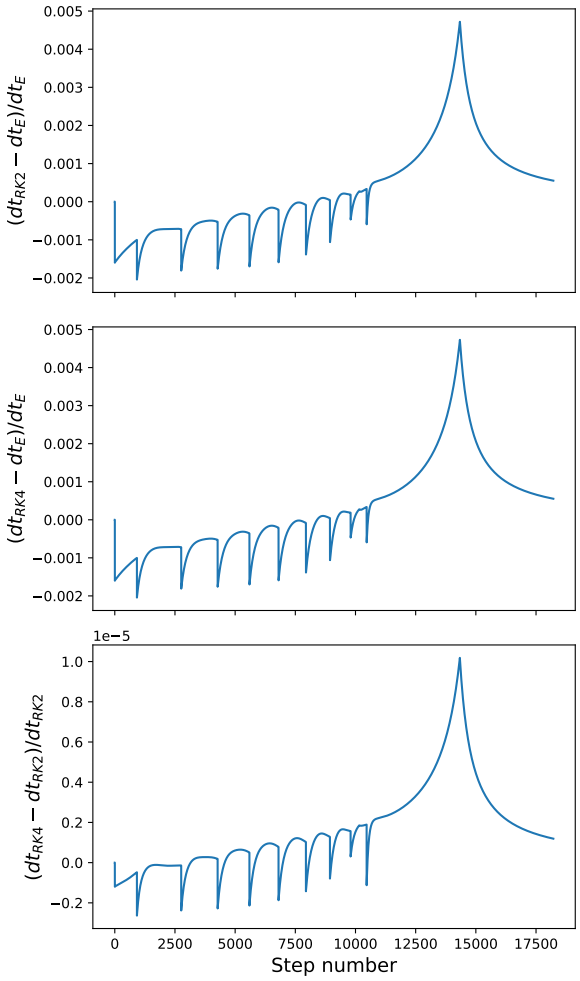

FIG. 26. The fractional difference in step-size between each pair of ODE steppers. These data show that each Runge-Kutta method has computed adaptive step-sizes that agree with the Euler method to within 0.5%. Further, the Runge-Kutta methods themselves agree to roughly a part in 10^5.

Kutta method has time-steps which lie within 0.5% of the Euler method time-steps, and the Runge-Kutta methods agree with each other to roughly a part in 10^5. This serves a further evidence that the ODE stepper used has quite little impact on the evolution of the neutrino distribution function. As such, we conclude that our choice to use Euler steps was justified.

VII. CONCLUSIONS

In this work, we have presented the Kompaneets formalism for describing neutrino-nucleon scattering from [2] and have developed a numerical solver that implements this formalism for use in supernova simulations. In the process, we have considered several interpolation methods, differentiation methods, and ODE-solving integration methods as regards their speed and accuracy,

especially in the case of a low number of sampling points.

In particular, we have found that using cubic Lagrange interpolating polynomials on our sampling intervals generally obtains the best results for interpolation and differentiation. The order of convergence of these methods rivals the more robust cubic spline methods while being significantly faster and equally accurate for low numbers of sampling points. Further, we have found that, for our purposes at least, Euler's method is sufficient for the description of our system, obtaining comparable results to higher order Runge-Kutta methods. Moreover, in the context of the full solver, we find that using higher order interpolation and differentiation methods throughout can lead to poor performance at very low numbers of energy bins.

We have also tested the solver and found that it gives robust performance consistent with analytic predictions and past works.

REFERENCES

[1] A. Burrows and T. A. Thompson, in *Stellar Collapse*, edited by C. L. Fryer (Springer Netherlands, Dordrecht, 2004), vol. 302 of *Astrophysics and Space Science Library*, ISBN 978-90-481-6567-4 978-0-306-48599-2, URL `http://link.springer.com/10.1007/978-0-306-48599-2`.

[2] T. Wang and A. Burrows, Phys. Rev. D **102**, 023017 (2020), ISSN 2470-0010, 2470-0029, arXiv: 2006.12240, URL `http://arxiv.org/abs/2006.12240`.

[3] R. L. Burden and J. D. Faires, *Numerical Analysis* (Cengage Learning, 2010), ninth edition ed.

[4] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics*, no. 37 in Texts in Applied Mathematics (Springer, 2007), second edition ed.

[5] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, no. 12 in Texts in Applied Mathematics (Springer, 2002), third edition ed.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C* (Cambridge: Cambridge University Press, 1992), second edition ed.

[7] T. A. Thompson, A. Burrows, and J. E. Horvath, Phys. Rev. C **62**, 035802 (2000), ISSN 0556-2813, 1089-490X, arXiv: astro-ph/0003054, URL `http://arxiv.org/abs/astro-ph/0003054`.

Appendix A: Codes

In this section, we provide sample codes in C that implement the various numerical methods described throughout the work.

1. Interpolation methods

For our interpolation methods, we define the following API.

```c
/**
 * The API for an interpolation method.
 * @param out   The array to be filled with output values [length: n].
 * @param x     The sampling points [length: n+1].
 * @param y     The sampled values of the function [length: n+1].
 * @param n     The number of intervals.
 */
void interpolation(float out[], float x[], float y[], int n);
```

The first interpolation method that we consider is simple linear interpolation. Abstractly, one can understand this method as constructing a linear interpolating polynomial in each $[x_j, x_{j+1}]$ interval and returning the value of the interpolant at the midpoint of the interval. This process is equivalent, however, to simply returning the average of the sampled values on the endpoints of each interval.

```c
void linear_itp(float out[], float x[], float y[], int n) {
    int i;
    for (i = 0; i < n; i++)
        out[i] = 0.5 * (y[i] + y[i+1]);
}
```

The next interpolation method we will consider is the cubic Lagrange method. For each interval $[x_j, x_{j+1}]$, this method forms a cubic interpolating polynomial by Lagrange interpolation on the points $\{j-1, j, j+1, j+2\}$, and then returns the value of this interpolant at the midpoint of the interval. In the first and last bins, the computation instead forms a quadratic Lagrange interpolant using the three points (of the previously described set of four) that are available.

```c
void cubic_itp(float out[], float x[], float y[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        // Forward quadratic
        if (i == 0)
            out[i] = (3.0 * y[i] + 6.0 * y[i+1] - y[i+2]) / 8.0;

        // Backward quadratic
        else if (i == n - 1)
            out[i] = (-y[i-1] + 6.0 * y[i] + 3.0 * y[i+1]) / 8.0;

        // Full cubic
        else
            out[i] = (9.0 * (y[i+1] + y[i]) - (y[i+2] + y[i-1])) / 16.0;
    }
}
```

The last interpolation method we consider is the cubic spline method. The algorithm for computing this follows closely the one presented in [3]. Again, we form a cubic spline interpolant, then return its value at the midpoint of each interval.

```c
void spline_itp(float out[], float x[], float y[], int n) {
```

```
    int i;
    float a[n+1], b[n+1], c[n+1], d[n+1], h[n], alpha[n], l[n+1], mu[n+1], z[n+1];

    // Find the coefficients a, b, c, d
    for (i = 0; i < n + 1; i++)
        a[i] = y[i];
    for (i = 0; i < n; i++)
        h[i] = x[i+1] - x[i];
    for (i = 1; i < n; i++)
        alpha[i] = 3 * (y[i+1] - y[i]) / h[i] - 3 * (y[i] - y[i-1]) / h[i-1];

    l[0] = 1;
    mu[0] = 0;
    z[0] = 0;

    for (i = 1; i < n; i++) {
        l[i] = 2 * (x[i+1] - x[i-1]) - h[i-1] * mu[i-1];
        mu[i] = h[i] / l[i];
        z[i] = (alpha[i] - h[i-1] * z[i-1]) / l[i];
    }

    l[n] = 1;
    z[n] = 0;
    c[n] = 0;

    for (i = n - 1; i >= 0; i--) {
        c[i] = z[i] - mu[i] * c[i+1];
        b[i] = (a[i+1] - a[i]) / h[i] - h[i] * (c[i+1] + 2 * c[i]) / 3;
        d[i] = (c[i+1] - c[i]) / (3 * h[i]);
    }

    // Write the contents of `out`
    float x;
    for (i = 0; i < n; i++) {
        x = 0.5 * (x[i+1] - x[i]); // midpoint of bin i - x[i]
        out[i] = a[i] + b[i] * x + c[i] * x*x + d[i] * x*x*x;
    }
}
```

2. Differentiation methods

For our differentiation methods, we define the following API.

```
/**
 * The API for a differentiation method.
 * @param out   The array to be filled with output values [length: n].
 * @param x     The sampling points [length: n+1].
 * @param y     The sampled values of the function [length: n+1].
 * @param n     The number of intervals.
 */
void differentiation(float out[], float x[], float y[], int n);
```

We begin again with the method that we have called “linear”. This method is also known as “centered” or “three-point” finite differencing. It approximates the derivative at the center of each interval by the slope of the line connecting the two endpoints.

```
void linear_diff(float out[], float x[], float y[], int n) {
    int i;
    for (i = 0; i < n; i++)
```

```
        out[i] = (y[i+1] - y[i]) / (x[i+1] - x[i]);
}
```

The next method we consider is derived from our cubic Lagrange interpolation method. In this method, we effectively perform the interpolation method described before, but instead of returning the value of the interpolant at the midpoint of the interval, we return the derivative of the interpolant at the midpoint. The derivative of the quadratic interpolant that is fit to the first and last bins reduces to the simple "linear" derivative method for the case of equispaced bins.

```
void cubic_diff(float out[], float x[], float y[], int n)
{
    int i;
    for (i = 0; i < n; i++) {
        if (i == 0 || i == n - 1)
            out[i] = (y[i+1] - y[i]) / (x[i+1] - x[i]);
        else
            out[i] = (27.0 * (y[i+1] - y[i]) - (y[i+2] - y[i-1])) / (24.0 * (x[i+1] - x[i]));
    }
}
```

The last method we consider is the cubic spline method. Like the other two methods, this is derived from the interpolation method of the same name, but instead of returning the value of the interpolant, we return its derivative.

```
void spline_diff(float out[], float x[], float y[], int n)
{
    int i;
    float a[n+1], b[n+1], c[n+1], d[n+1], h[n], alpha[n], l[n+1], mu[n+1], z[n+1];

    // Find the coefficients a, b, c, d
    for (i = 0; i < n + 1; i++)
        a[i] = y[i];
    for (i = 0; i < n; i++)
        h[i] = x[i+1] - x[i];
    for (i = 1; i < n; i++)
        alpha[i] = 3 * (y[i+1] - y[i]) / h[i] - 3 * (y[i] - y[i-1]) / h[i-1];

    l[0] = 1;
    mu[0] = 0;
    z[0] = 0;

    for (i = 1; i < n; i++) {
        l[i] = 2 * (x[i+1] - x[i-1]) - h[i-1] * mu[i-1];
        mu[i] = h[i] / l[i];
        z[i] = (alpha[i] - h[i-1] * z[i-1]) / l[i];
    }

    l[n] = 1;
    z[n] = 0;
    c[n] = 0;

    for (i = n - 1; i >= 0; i--) {
        c[i] = z[i] - mu[i] * c[i+1];
        b[i] = (a[i+1] - a[i]) / h[i] - h[i] * (c[i+1] + 2 * c[i]) / 3;
        d[i] = (c[i+1] - c[i]) / (3 * h[i]);
    }
    // Write the contents of `out`
    long double x;
    for (i = 0; i < n; i++) {
        x = 0.5 * (x[i+1] - x[i]); // midpoint of bin i - x[i]
        out[i] = b[i] + 2.0 * c[i] * x + 3.0 * d[i] * x*x;
    }
```

```
}
```

3. Integration methods

For our integration methods, we define the following API.

```
/**
 * The API for an integration method.
 * @param yout  The array to be filled with updated y values.
 * @param y     The array containing the current y values.
 * @param n     The length of the y array.
 * @param t     The current time.
 * @param h     The step size.
 * @param rhs   A function which computes the right-hand side of the ODE. Takes
 arguments
                float out[] The array to be filled with the current RHS of the ODE.
                float y[]   The input y values.
                float t     The current time.
                int n       The length of the y array.
 */
 void integration(float yout[], float y[], int n, float t, float h,
                  void (*rhs)(float[], float[], float, int));
```

We begin with Euler's method. This method is really quite simple, but it shows rather easily how all of the arguments are required to perform the integration.

```
void euler(float yout[], float y[], int n, float t, float h,
           void (*rhs)(float[], float[], float, int)) {
    int i;
    float dydt[n];

    (*rhs)(dydt, y, t, n);
    for (i = 0; i < n; i++)
        yout[i] = y[i] + h * dydt[i];
}
```

Next, we implement the second-order Runge-Kutta method. This method requires two calls to the `rhs` function.

```
void rk2(float yout[], float y[], int n, float t, float h,
         void (*rhs)(float[], float[], float, int)) {
    int i;
    float dydt[n], k1[n], k2[n];
    float yt[n]; // temporary y values

    // Compute k1
    (*rhs)(dydt, y, t, n);
    for (i = 0; i < n; i++)
        k1[i] = h * dydt[i];

    // Compute k2
    for (i = 0; i < n; i++)
        yt[i] = y[i] + 0.5 * k1[i];
    (*rhs)(dydt, yt, t + 0.5 * h, n);
    for (i = 0; i < n; i++)
        k2[i] = h * dydt[i];

    // Compute the updated y values
    for (i = 0; i < n; i++)
```

```
        yout[i] = y[i] + k2[i];
}
```

Finally, we implement the fourth-order Runge-Kutta method.

```
void rk4(float yout[], float y[], int n, float t, float h,
         void (*rhs)(float[], float[], float, int)) {
    int i;
    float dydt[n], k1[n], k2[n], k3[n], k4[n];
    float yt[n]; // temporary y values

    // Compute k1
    (*rhs)(dydt, y, t, n);
    for (i = 0; i < n; i++)
        k1[i] = h * dydt[i];

    // Compute k2
    for (i = 0; i < n; i++)
        yt[i] = y[i] + 0.5 * k1[i];
    (*rhs)(dydt, yt, t + 0.5 * h, n);
    for (i = 0; i < n; i++)
        k2[i] = h * dydt[i];

    // Compute k3
    for (i = 0; i < n; i++)
        yt[i] = y[i] + 0.5 * k2[i];
    (*rhs)(dydt, yt, t + 0.5 * h, p);
    for (i = 0; i < n; i++)
        k3[i] = h * dydt[i];

    // Compute k4
    for (i = 0; i < n; i++)
        yt[i] = y[i] + k3[i];
    (*rhs)(dydt, yt, t + h, p);
    for (i = 0; i < n; i++)
        k4[i] = h * dydt[i];

    // Compute the updated y values
    for (i = 0; i < n; i++)
        yout[i] = y[i] + (k1[i] + 2.0 * (k2[i] + k3[i]) + k4[i]) / 6.0;
}
```