Karlsruhe Institute of Technology

ETP-KA/2023-14
FAKULTÄT FÜR PHYSIK
Institut für Experimentelle Teilchenphysik (ETP)

Workflow- und Performanzoptimierung für schnelle NNLO pQCD-Berechnungen

(Workflow and performance optimization for fast NNLO pQCD Calculations)

Masterarbeit
von

Johannes Gäßler

am Institut für Experimentelle Teilchenphysik (ETP)

Reviewer:	P. D. Dr. Klaus Rabbertz
Second Reviewer:	Prof. Dr. Günter Quast

Bearbeitungszeit: 21.12.2022 – 21.12.2023

Erklärung zur Selbstständigkeit

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung vom 24.05.2018 beachtet habe.

Karlsruhe, den 20.12.23, _____

Johannes Gäßler

Als Prüfungsexemplar genehmigt von

Karlsruhe, den 20.12.23, _____

P. D. Dr. Klaus Rabbertz

Contents

1. Introduction **1**

2. fastNLO: Fast pQCD Calculations for a posteriori PDF choices **3**
2.1. Scale Nodes 4
2.2. Software Architecture and Context 5

3. Parton Distribution Function Interpolation Techniques **7**
3.1. Node selection 15
3.2. Monte Carlo Integration 16

4. `NodeDensity`: A New Method for x Node Spacing **19**
4.1. Scale nodes 23

5. Results **25**
5.1. Performance optimization 25
5.2. Node Density Efficiency Benchmarks 29

6. Conclusion **39**
6.1. Outlook 39

7. ROOT Routines **3**

Appendix **41**

A. Implementation Details **43**
A. Configuration 43
B. Workflow Integration 43
C. Table Filling 45
D. Table Merging 52
E. Testing 53

B. NNLOJET Runtime Percentages **55**

Bibliography **59**

List of Figures

2.1. NLOJet++ interface 5
2.2. NNLOJET interface 6

3.1. Nearest-neighbor interpolation 8
3.2. Linear interpolation 8
3.3. Lagrange interpolation 10
3.4. Lagrange spline interpolation 11
3.5. Catmull-Rom spline interpolation 12
3.6. Interpolation error comparison 12
3.7. Rel. interpolation error comparison 13
3.8. Interpolation cancellation comparison 14
3.9. Lagrange interpolation four nodes 15

4.1. NodesPerBin vs. NodeDensity 20
4.2. NodesPerBin vs. NodeDensity 21
4.3. Node weight spread 22

5.1. Interpolation efficiency dijet all 30
5.2. Interpolation efficiency dijet 1 31
5.3. Interpolation efficiency dijet 2 31
5.4. Interpolation efficiency dijet 3 32
5.5. Interpolation efficiency dijet 4 32
5.6. Interpolation efficiency dijet 5 33
5.7. Z+jet rapidity bins 34
5.8. Interpolation efficiency Z+jet LO all 35
5.9. Interpolation efficiency Z+jet NLO all 36
5.10. Interpolation efficiency Z+jet NLO all 98% 36
5.11. Interpolation efficiency Z+jet NLO all 95% 37
5.12. Interpolation efficiency Drell-Yan LO all 38

A.1. Warmup run logic 44
C.2. Original Table Filling 46
C.3. New Table Filling 48
C.4. Table Filling Logic 49
C.5. fastNLOCreate class diagram 50

List of Tables

3.1. Lagrange polynomials 9
3.2. Transfer functions 16

5.1. NNLOJET rev5918 profiling by contribution 26
5.2. NNLOJET rev6591 LC profiling by contribution 27
5.3. NNLOJET rev6591 FC profiling by contribution 28
5.4. NNLOJET rev6591 FC MC profiling by contribution 28

.1. Comparison of RRa profiling results for select NNLOJET methods. 56
.2. NNLOJET rev6591 RRa method runtime 57
.3. NNLOJET rev6591 RV method runtime 57

1. Introduction

The Large Hadron Collider at CERN - as the name suggests - collides hadrons, namely protons. Compared to the previous Large Electron-Positron Collider the advantage of the LHC is its much higher center-of-mass energy of 13.6 TeV compared to the 209 GeV achieved with LEP. This is in part because protons have a mass of 938 MeV compared to only 0.511 MeV for electrons/positrons. For this reason they lose a significantly lower fraction of their energy to Bremsstrahlung in a circular collider which in turn increases the maximum energy to which particles can be accelerated. However, the disadvantage of the LHC is that the use of protons also makes the use of its data for analyses more challenging. According to the standard model electrons and positrons are elementary particles. Protons on the other hand consist of multiple partons, three valence quarks (2 up, 1 down) as well as virtual sea quarks and gluons. Theory predictions (e.g. the cross sections of various physical processes) then of course depend on the exact proton contents.

At high enough energies quantum chromodynamics becomes accessible to perturbation theory (perturbative quantum chromodynamics). So-called *parton distribution functions* (PDFs) can then be used to describe the probability density for finding a given parton with momentum fraction x at energy scale μ. Unfortunately these PDFs are not known from theory. They instead need to be estimated from experimental data and therefore introduce a corresponding uncertainty to analyses using LHC data. The two primary methods for estimating these uncertainties are to either derive them from the method of maximum likelihood when fitting a PDF to experimental data (e.g. [23]) or from an ensemble of PDFs derived from subsets of the experimental data (oftentimes using artificial neural networks, e.g. [8]). Both of these approaches have in common that the PDFs have free parameters that need to be optimized numerically in order to match the experimental data. In practical terms, this means that the cost function for determining how closely the theory predictions match experimental data needs to be evaluated hundreds if not thousands of times for different sets of free PDF parameters. And for each evaluation of the cost function with a new set of PDF parameters in turn new theory predictions need to be calculated. To make such analyses feasible the computational costs of theory predictions therefore need to be sufficiently low.

fastNLO[19][10] is a project which - as the name implies - was created to facilitate the calculation of fast next-to-leading-order theory predictions. The concept is to calculate so-called "coefficient tables" that separate the PDFs from the rest of the calculation by effectively interpolating them. Given a coefficient table, theory predictions can then be calculated quickly for an arbitrary choice of PDFs, the strong coupling constant α_s (and potentially energy scales). As of writing the project has evolved to support calculations up to next-to-next-to-leading order for deep inelastic scattering, proton-proton collisions, and proton-antiproton collisions.

First, in chapter 2 the theoretical aspects for pQCD calculations as well as the concept behind the fastNLO project will be explained. Chapter 3 details the intricacies of PDF

interpolation in further detail. Chapter 4 describes the concept behind the implementation of `NodeDensity`, a new technique for x node interpolation that allows for a simpler workflow; the corresponding implementation details can be found in appendix A. Chapter 5 will list the results of the code changes made for this thesis and benchmarks against the preexisting implementations in terms of memory/disk space use. And Finally chapter 6 will recapitulate the contents of this thesis and provide insight into possible further areas of improvement.

2. fastNLO: Fast pQCD Calculations for a posteriori PDF choices

As first discovered by David Gross, Frank Wilczek, and David Politzer, the strength of the *strong coupling constant* α_s depends on the energy scale of the interaction[16][22]. At low energies/long distances α_s is large while it decreases for high energies/small distances. As a consequence, in a theoretical infinite momentum frame the quarks and gluons inside a hadron can be treated as a cloud of free partons that do not interact via the strong force (asymptotic freedom). While a particle obviously cannot be given infinite momentum (and therefore infinite energy) in an actual physics experiment, the *factorization* of a hadron into its parton is still a valid assumption at high enough energies. It is then possible to define *parton distribution functions* $f(x, \mu_f)$ that describe the likelihood of finding a specific parton with momentum fraction x of the hadron at factorization scale μ_f. *Perturbative quantum chromodynamics*[14] then allows for the calculation of a proton-proton cross section σ given an additional renormalization scale μ_r and perturbative coefficients $c_{a,b,n}(x_1, x_2, \mu_r, \mu_f)$:

$$\sigma = \sum_{a,b,n} \int_0^1 dx_1 \int_0^1 dx_2 \; \alpha_s^n(\mu_r) \; c_{a,b,n}(x_1, x_2, \mu_r, \mu_f) \; f_a(x_1, \mu_f) \; f_b(x_2, \mu_f), \tag{2.1}$$

where the indices a and b indicate the respective types of the ingoing partons and n indicates the order of the strong coupling constant α_s. Unfortunately calculating the above integral is computationally expensive. Due to the large number of coupled degrees of freedom the only feasible method for numerical integration is Monte Carlo integration where the uncertainty on the result only decreases with the square root of the number of generated events. One possibility to keep the computational costs low is to only calculate the cross section at leading order (LO). However, because α_s is on the order of 10^{-1} (compared to e.g. the fine-structure constant with $\alpha \approx \frac{1}{137}$) the difference between LO and higher orders can be comparatively large.

The core idea behind fastNLO is to separate the dependency of the cross section on α_s and the parton PDFs from the Monte Carlo integration. That way the Monte Carlo integration has to be performed only once and α_s and the PDFs can be chosen a posteriori to calculate the cross section. To explain how this works, consider a decomposition $f(x, \mu_f) = \sum_i a_i \; g_i(x, \mu_f)$ of the PDF used above where a_i are simple scalars and $g_i(x, \mu_f)$ are functions. The cross section calculation can then be rearranged as such:

$$\begin{aligned}
\sigma(\mu_r, \mu_f) &= \sum_{a,b,n} \int_0^1 dx_1 \int_0^1 dx_2 \; \alpha_s^n(\mu_r) \; c_{a,b,n}(x_1, x_2, \mu_r, \mu_f) \sum_{i,j} a_i \; g_i(x, \mu_f) \; a_j \; g_j(x, \mu_f) \\
&= \sum_{i,j,n} a_i \; a_j \; \alpha_s^n(\mu_r) \sum_{a,b} \int_0^1 dx_1 \int_0^1 dx_2 \; c_{a,b,n}(x_1, x_2, \mu_r, \mu_f) \; g_i(x, \mu_f) \; g_j(x, \mu_f) \\
&=: \sum_{i,j,n} a_i \; a_j \; \alpha_s^n(\mu_r) \; \tilde{\sigma}_{ijn}(\mu_r, \mu_f).
\end{aligned} \tag{2.2}$$

Notably the cross section can now be calculated as a simple sum given the pre-computed coefficients $\tilde{\sigma}_{ijn}$. However, in order to actually reduce the amount of computation necessary for e.g. PDF fits these coefficients must be made reusable for different PDF choices. This can be achieved by choosing the PDF composition in a way that *interpolates* the PDF given some set of *nodes* x_i via $a_i = f(x_i, \mu_f)$ and $g_i(x_j, \mu_f) = \delta_{ij}$:

$$f(x, \mu_f) = \sum_i a_i \; g_i(x, \mu_f) \approx \sum_i f(x_i, \mu_f) \; g_i(x, \mu_f). \tag{2.3}$$

Plugging this into equation 2.2 yields:

$$\sigma(\mu_r, \mu_f) \approx \sum_{i,j,n} f(x_i, \mu_f) \; f(x_j, \mu_f) \; \alpha_s^n(\mu_r) \; \tilde{\sigma}_{ijn}(\mu_r, \mu_f). \tag{2.4}$$

With this decomposition choice the coefficients $\tilde{\sigma}_{ijn}(\mu_r, \mu_f)$ become entirely independent of the PDF $f(x, \mu_f)$ and depend only on the nodes x_i. In the context of the fastNLO project the set of pre-computed coefficients $\tilde{\sigma}_{ijn}(\mu_r, \mu_f)$ is also referred to as a "coefficient table". The coefficients are computed only once and then reused to calculate a cross section for an arbitrary PDF and α_s choice via the sum in eq. 2.4. This is only an approximation but according to the *Weierstrass approximation theorem*[27] polynomials can be used to interpolate any continuous function to arbitrary precision.

This then only leaves the question of how to choose the nodes x_i. One approach is to do a "warmup" run in which some events are generated for the sole purpose of determining good x_i values for a "production" run where the actual coefficients are calculated. With this method the x nodes are set at the beginning of the production run and not changed during the run. The primary goal of this thesis is to implement and test methods for choosing x_i values *without* a prior warmup run (see chapter 4).

2.1. Scale Nodes

So far, the cross section was simply treated as a function of the scales μ_r, μ_f. However, for an actual physics analysis using LHC data the number of events is so large that binning the events is effectively mandatory. Typically one of the histogram dimensions is some sort of energy scale, for example the dijet mass m_{12} in the case of dijet analyses. In order for the theory predictions produced by fastNLO to match those bins the cross section needs to be integrated across the bin width. The additional integration dimension can be treated in the same way as the integration over the momentum fraction: the PDF is interpolated in the scale dimension to absorb the scale dependence into the coefficients $\tilde{\sigma}_{ijkn}$ where k is the newly added scale node index. Assuming $\mu_r = \mu_f = \mu$ and given scale nodes μ_k the cross section becomes:

$$\int \sigma(\mu) d\mu \approx \sum_{i,j,k,n} f(x_i, \mu_k) \; f(x_j, \mu_k) \; \alpha_s^n(\mu_k) \; \tilde{\sigma}_{ijkn}. \tag{2.5}$$

Given a decomposition

$$f(x, \mu) \approx \sum_{i,k} f(x_i, \mu_k) \; g_i(x) \; h_k(\mu) \tag{2.6}$$

the coefficients $\tilde{\sigma}_{ijkn}$ are defined as:

$$\tilde{\sigma}_{ijkn} = \sum_{a,b} \int d\mu \int_0^1 dx_1 \int_0^1 dx_2 \; c_{a,b,n}(x_1, x_2, \mu) \; g_i(x) \; g_j(x) \; h_k(\mu). \tag{2.7}$$

The scale variation is handled by explicitly calculating and saving coefficients for multiple scale variations. In fastNLO this type of coefficient table is referred to as a "fixed-scale table". Alternatively the coefficient table can be saved as the scale-independent part plus factors of $\log \mu_r$, $\log \mu_f$, $\log^2 \mu_r$, $\log \mu_r \log \mu_f$, and $\log^2 \mu_f$. This way in addition to the PDF and α_s the energy scale can also be chosen freely when calculating the cross section. In fastNLO this type of coefficient table is referred to as a "flexible scale tables".

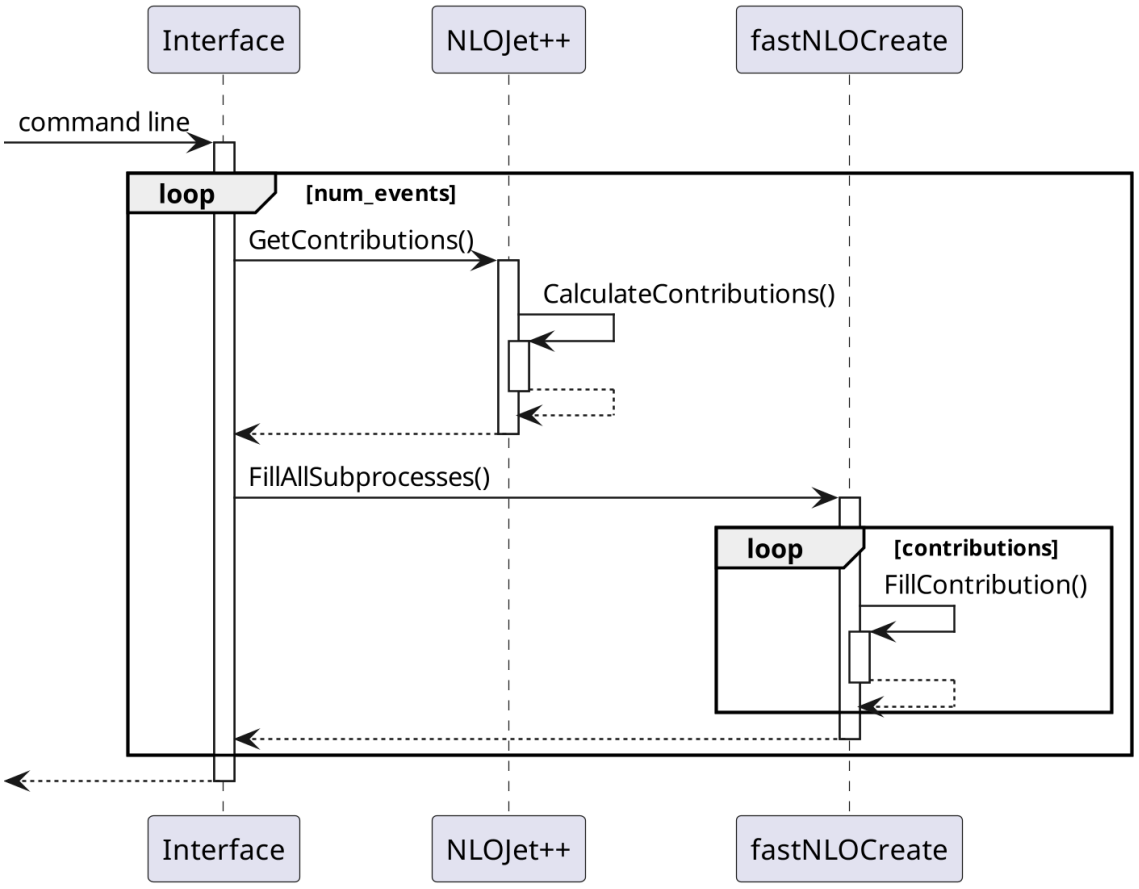

Figure 2.1.: UML sequence diagram showing schematically how NLOJet++ can be used to fill fastNLO coefficient tables in user code.

2.2. Software Architecture and Context

fastNLO by itself is not sufficient to generate coefficient tables. Instead it is part of a wider QCD software ecosystem. In particular, an external event generator is required. As of writing, Sherpa[9], NLOJet++[21], and NNLOJET[15] are supported. Alternatives to fastNLO include *APPLGrid*[11] and *PineAPPL*[12].

Due to the differences in software architecture between event generators the corresponding fastNLO interfaces also need to differ. Figure 2.1 shows how NLOJet++ can be used to fill fastNLO coefficient tables. Both NLOJet++ and fastNLO are used as software libraries by user code: the user code first generates events using NLOJet++, then passes those events to fastNLO. By contrast, figure 2.2 shows the equivalent process for NNLOJET. Instead of user code, it is the NNLOJET binary itself that (given the correct compilation and configuration options) fills the fastNLO coefficient tables. The coefficient tables are created via an adapter named *NNLO Bridge* (part of NNLOJET). This adapter provides a unified interface to both fastNLO and APPLgrid to ensure that both libraries can be used interchangeably.

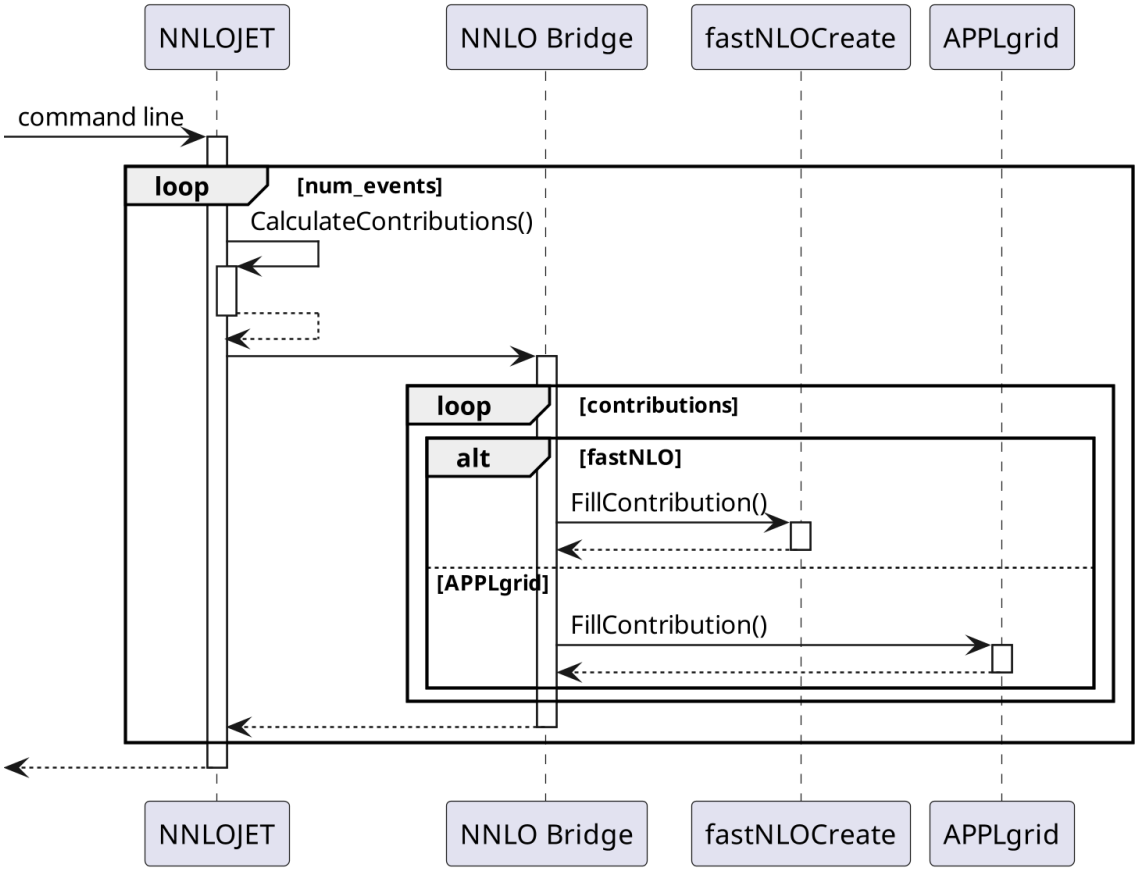

Figure 2.2.: UML sequence diagram showing schematically how the NNLOJET binary can be used to fill fastNLO coefficient tables.

3. Parton Distribution Function Interpolation Techniques

For the purposes of this thesis, an interpolation $G(x)$ of a function $f(x)$ is defined as a decomposition of $f(x)$ into functions $g_i(x)$ using n nodes x_i:

$$f(x) \approx G(x) = \sum_{i=0..n-1} f(x_i)\ g_i(x), \quad g_i(x_j) = \delta_{ij}. \tag{3.1}$$

The Kronecker delta ensures that modifying the function value $f(x_i)$ for one of the nodes does not affect the value of any other nodes. For the following discussion the *interpolation error* δ is defined as the maximum absolute difference between a function $f(x)$ and its approximation $G(x)$ on an interval $[a, b]$:

$$\delta = \max \left| f(x) - G(x) \right|, \quad x \in [a, b]. \tag{3.2}$$

The definition in eq. 3.2 assumes a one-dimensional interval $[a, b]$ but it can be trivially generalized to a volume of higher dimension.

A very simple way to interpolate a function would be to construct an equidistant *grid* of nodes across the volume of interest and to evaluate the function at each grid point. When evaluating the interpolated function $G(x)$ the x vector is then rounded to the nearest grid point and the function value at that grid point is assumed for the interpolated function (nearest-neighbor interpolation). The functions $g_i(x)$ are then either 1 if x_i is the nearest neighbor and 0 otherwise. A visualization of the technique is shown in figure 3.1. For simplicity an interpolation of a simple one-dimensional function $f(x) = x(1-x)\exp(-3x)$ is shown (but the same principles apply for higher dimensions). The previously mentioned function is chosen because it vaguely resembles a parton distribution function: it satisfies $f(0) = 0$ and $f(1) = 0$ with a bias towards lower x values. With seven nodes nearest-neighbor interpolation achieves an interpolation error of $\delta = 5.9 \cdot 10^{-2}$ for the toy problem. For an infinitely large grid (with infinitely dense nodes due to the finite volume) this technique could approximate any function to arbitrary precision. But an infinitely large grid would imply infinite CPU time and memory - and since the whole point is to reduce computing costs that would not be a good solution.

A still relatively simple improvement upon the previous approach would be to *linearly interpolate* the grid values between the points instead of just rounding to the nearest point. A visualization of linear interpolation is shown in figure 3.2. The $g_i(x)$ are defined as follows:

$$g_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if } 1 \leq i \leq n-1 \wedge x_{i-1} \leq x \leq x_i, \\ \frac{x - x_{i+1}}{x_i - x_{i+1}} & \text{if } 0 \leq i \leq n-2 \wedge x_i \leq x \leq x_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

If the x value exactly matches a grid point x_i then the interpolated function is equal to the value of the grid point $f(x_i)$. With seven nodes linear interpolation has an interpolation

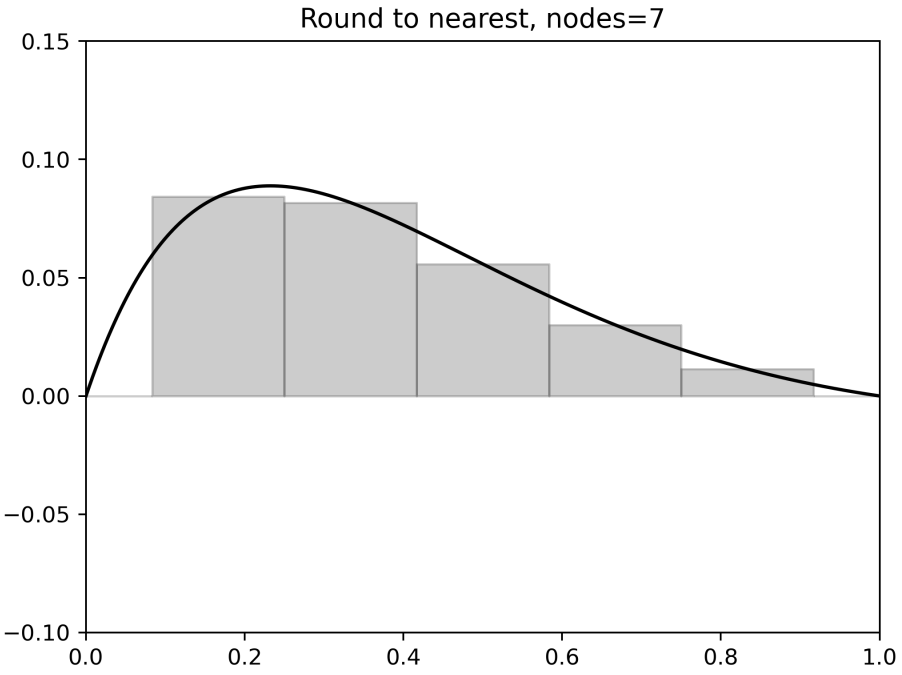

Figure 3.1.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using nearest-neighbor interpolation. The solid black line indicates the function $f(x)$. The gray area indicates the interpolated function $G(x)$.

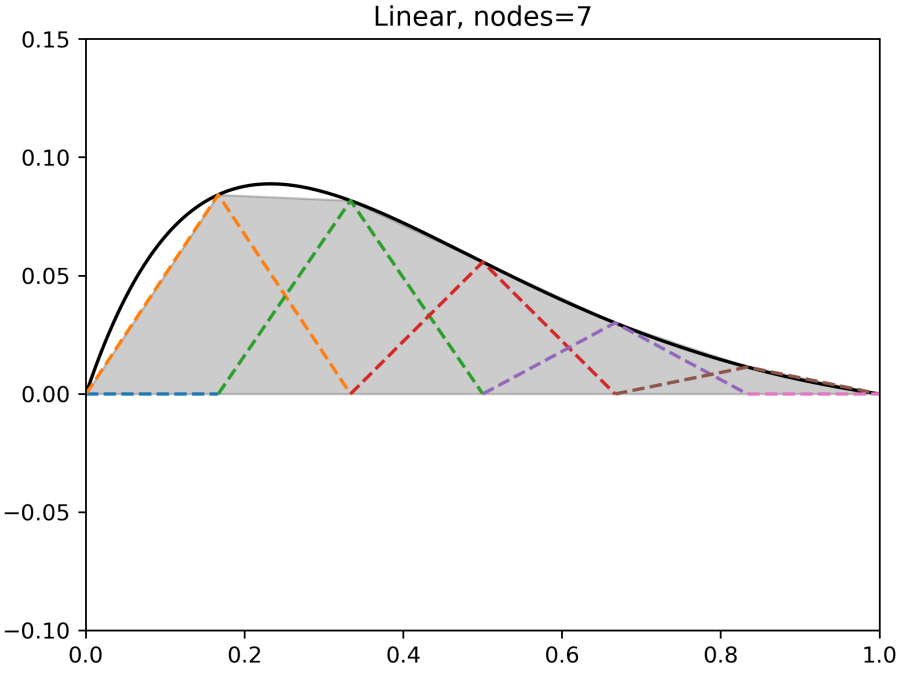

Figure 3.2.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using linear interpolation. The solid black line indicates the function $f(x)$. The gray area indicates the interpolated function $G(x)$. The colored, dashed lines indicate the nonzero $f(x_i)\ g_i(x)$ contributions of individual grid points to the interpolated function $G(x)$.

Table 3.1.: Lagrange polynomials up to $n = 4$ with nodes $x_i = i = 0, 1..n$. Note that the degree of the polynomials is $n - 1$.

n	i	$g_{n,k}(x) = \prod_{0 \leq j \leq n-1,\ j \neq i} \frac{x-j}{i-j}$.
1	0	1
2	0	$1 - x$
2	1	x
3	0	$\frac{1}{2}x^2 - \frac{3}{2}x + 1$
3	1	$2x - x^2$
3	2	$-1x + x^2$
4	0	$1 - \frac{11}{6}x + x^2 - \frac{1}{6}x^3$
4	1	$3x - \frac{5}{2}x^2 + \frac{1}{2}x^3$
4	2	$-\frac{3}{2}x + 2x^2 - \frac{1}{2}x^3$
4	3	$\frac{1}{3}x - \frac{1}{2}x^2 + \frac{1}{6}x^3$

error of $\delta = 1.7 \cdot 10^{-2}$ for the toy problem. Linear interpolation is superior to rounding because it matches the actual function very closely as long as the function's second derivative is small on the scale of the distance between two grid nodes. For Monte Carlo integrations in particular linear interpolation also has the advantage that for an integration volume with n dimensions each node will receive contributions from 2^n more events compared to nearest neighbor interpolation and thus the result will converge much faster.

The improvement from linear interpolation compared to nearest-neighbor interpolation comes from the fact that it takes the first derivative of the function into account. Higher derivatives can be taken into account by utilizing polynomials of higher degrees. This leads us to the next improvement: the use of *Lagrange polynomials*[20]. For example, Lagrange polynomials that interpolate the function $f(x)$ can be constructed as follows:

$$g_i(x) = \prod_{\substack{0 \leq j \leq n-1 \\ j \neq i}} \frac{x - x_j}{x_i - x_j}. \tag{3.4}$$

Example Lagrange polynomials up to third degree are listed in table 3.1. The interpolation toy problem is also visualized in figure 3.3. The contributions from the nodes at $x = 0$ and $x = 1$ are not shown because they would be equal to 0.

With seven nodes Lagrange interpolation has an interpolation error of $\delta = 2.0 \cdot 10^{-4}$ for the toy problem, almost two orders of magnitude less than with linear interpolation. However, defining Lagrange polynomials across the entire grid comes with a significant disadvantage: when using equidistant nodes, Lagrange polynomials have a tendency towards oscillations between nodes, particularly towards the edges of the interval. This is known as *Runge's phenomenon*[26]. As a consequence there are cases, e.g. when interpolating the so-called Runge function $f(x) = \frac{1}{1+25x^2}$ with equidistant nodes on the interval $[-1, 1]$, in which the interpolation error actually *increases* as the number of equidistant nodes is increased.

Even if the interpolation error does not diverge as the number of nodes is increased the oscillations still pose problems for actually calculating the interpolated function. The interpolated function $G(x)$ receives contributions $f(x_i)\ g_i(x)$ from all grid points. And due to the oscillations in $g_i(x)$ these contributions can be many times larger than the actual function value $f(x)$ between grid points. While these contributions will theoretically cancel each other out when including all grid points they do in practice introduce significant rounding errors due to floating point arithmetic. This is because floating point values can

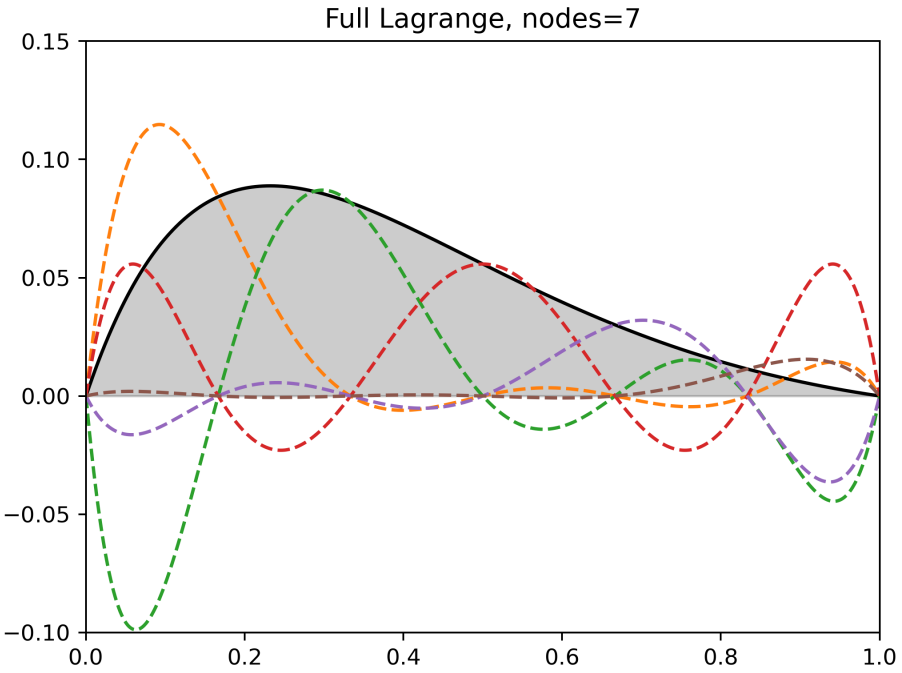

Figure 3.3.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using Lagrange polynomial interpolation. The solid black line indicates the function $f(x)$. The gray area indicates the interpolated function $G(x)$. The colored, dashed lines indicate the nonzero $f(x_i)\ g_i(x)$ contributions of individual grid points to the interpolated function $G(x)$.

only store a limited precision and thus exhibit a behavior known as *cancellation*[1]: if the difference of two floating point numbers is calculated the rounding error of the individual numbers is amplified if the absolute value of the difference is small compared to the absolute values of the original numbers. In extreme cases the resulting rounding error can be orders of magnitudes larger than the difference to be calculated.

There are multiple ways to mitigate Runge's phenomenon. One option would be to choose better nodes than simply equidistant ones since the *Weierstrass approximation theorem*[27] states that on a closed interval for any continuous function a polynomial approximation with arbitrarily small interpolation error must exist. However, in practice it is difficult to actually determine the correct choice of node values, particularly for parton density functions which are not known from theory. Therefore, in fastNLO Runge's phenomenon is mitigated by using *Lagrange splines* (local, piecewise Lagrange polynomials) instead of the full Lagrange polynomials spanning the entire grid. In practical terms this means that the Lagrange polynomials for interpolation are defined using only a subset of all grid points. Because the number of nodes actually used for interpolation is now constant, adding more nodes to the grid cannot result in a divergent interpolation error. The linear interpolation that we discussed before is in fact a Lagrange spline interpolation using first-degree Lagrangian polynomials. In that case each event modifies the weights of two grid points (assuming a single dimension). The next (symmetrical) candidate for spline interpolation would be to modify the weights of four grid points. This results in third degree Lagrangian polynomials. These are the polynomials employed by fastNLO's "Lagrange" kernels. At the edges of the grid there is only a single node on one side of the event in one of the dimensions. In that case that single node as well as three nodes (instead of two) on the other side are used for the interpolation. The Lagrange splines used by fastNLO are visualized in figure 3.4. With seven nodes third degree Lagrange spline interpolation has

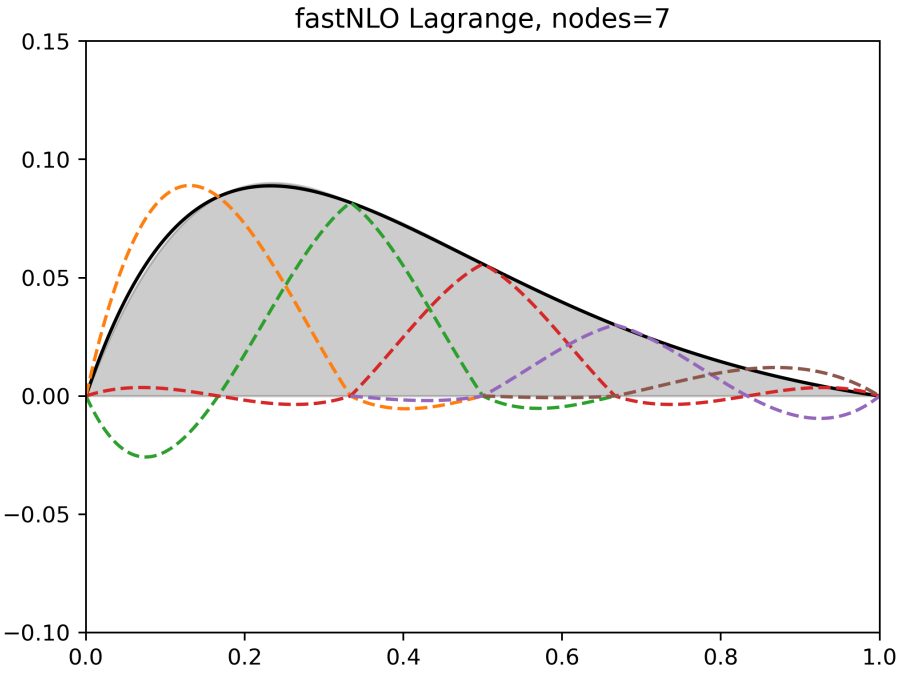

Figure 3.4.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using third degree Lagrange spline interpolation. The solid black line indicates the function $f(x)$. The gray area indicates the interpolated function $G(x)$. The colored, dashed lines indicate the nonzero $f(x_i)\ g_i(x)$ contributions of individual grid points to the interpolated function $G(x)$.

an interpolation error of $\delta = 2.8 \cdot 10^{-3}$ for the toy problem. The interpolation error thus roughly sits between linear interpolation and full Lagrange interpolation. But the ratio of the individual contributions $f(x_i)\ g_i(x)$ to the total interpolated function value $G(x)$ is much better (cancellation quantified further down).

An alternative way to define polynomial splines are so-called *cubic Hermite splines*[2]. As the name suggests, they are derived from cubic polynomials in their Hermite form, i.e. third degree polynomials defined by their values p_k and first-degree derivatives m_k at the boundaries. The way in which the derivatives m_k are chosen differentiates the subtypes of cubic Hermite splines. For the so-called *Catmull-Rom splines*[13] the derivatives are chosen as

$$m_k = \frac{p_{k+1} - p_{k-1}}{2}, \tag{3.5}$$

where the nodes are assumed to be equidistant with a distance of 1 between nodes. Naturally this requires special treatment at the edges. There the interpolation is instead done in the same way as with Lagrange polynomials, meaning the polynomials are defined by the node values only. The Catmull-Rom splines implemented by fastNLO are visualized in figure 3.5. With seven nodes Catmull-Rom spline interpolation has an interpolation error of $\delta = 1.1 \cdot 10^{-2}$ for the toy problem. This is comparable to the interpolation error of $\delta = 1.7 \cdot 10^{-2}$ obtained for linear interpolation. However, compared to the interpolations based on Lagrange polynomials the overshoot (and therefore the numerical cancellation) is greatly reduced.

The interpolation errors listed so far have always been for a total of seven nodes. However, the interpolation error of all of the described techniques can be reduced by increasing the number of nodes. Furthermore the relative magnitudes of the corresponding interpolation error δ may vary depending on the number of nodes. Figure 3.6 shows the interpolation

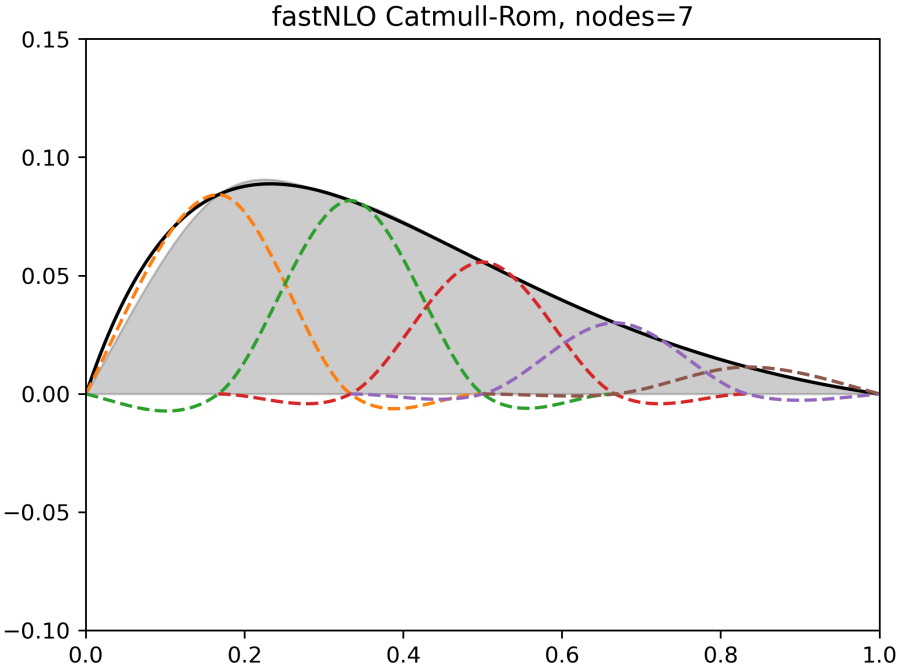

Figure 3.5.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using Catmull-Rom spline interpolation. The solid black line indicates the function $f(x)$. The gray area indicates the interpolated function $G(x)$. The colored, dashed lines indicate the nonzero $f(x_i)\ g_i(x)$ contributions of individual grid points to the interpolated function $G(x)$.

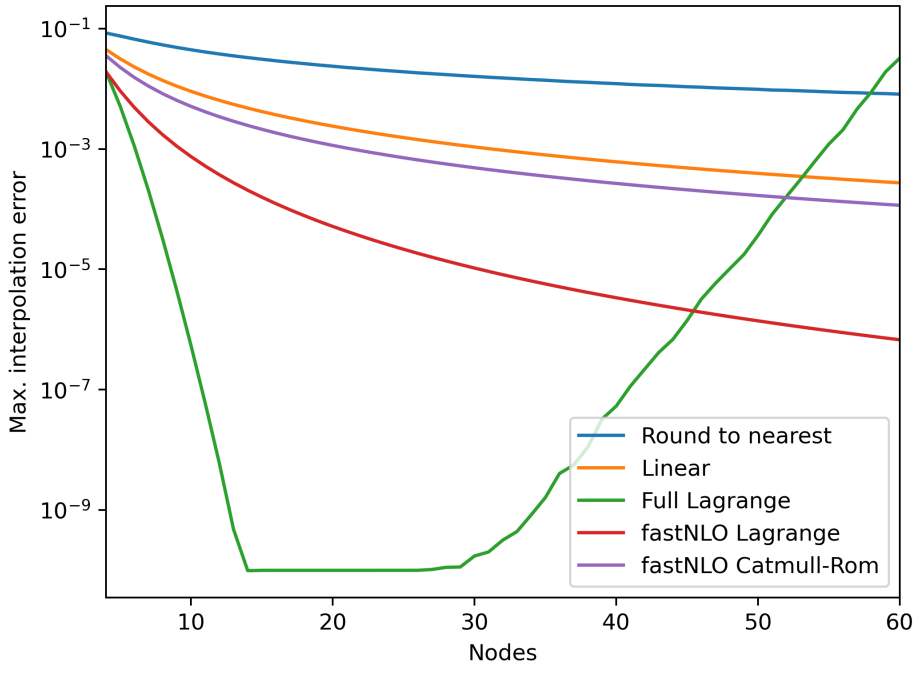

Figure 3.6.: Comparison of interpolation techniques in terms of interpolation error.

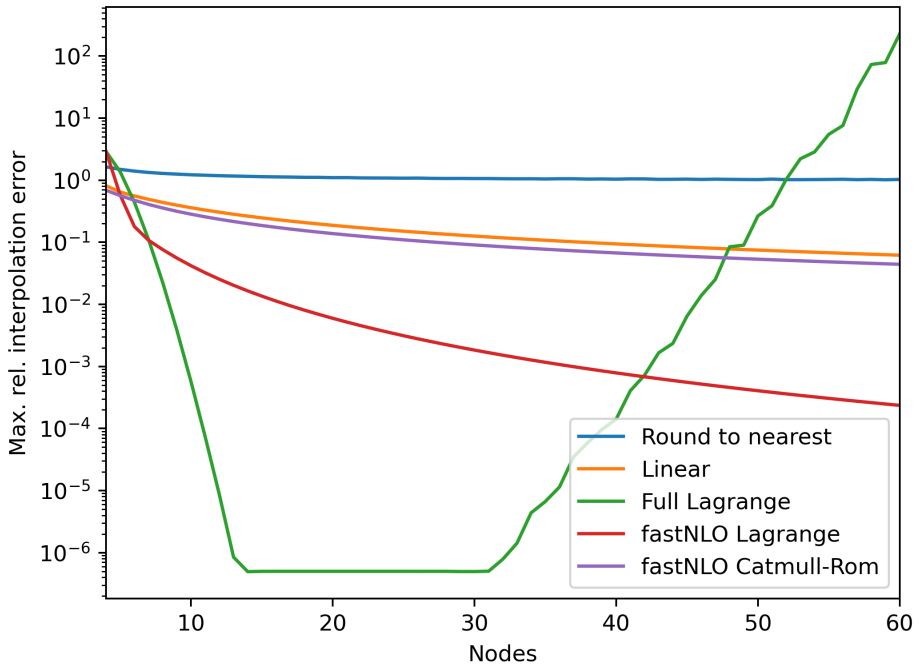

Figure 3.7.: Comparison of interpolation techniques in terms of relative interpolation error.

errors of the previously introduced interpolation techniques for the toy problem as a function of the number of nodes (equidistant spacing). When the number of nodes is low the interpolation errors differ by roughly one order of magnitude but this difference grows with the number of nodes up to nine orders of magnitude. With the exception of full Lagrange interpolation, the ranking by interpolation error does not change as the number of nodes increases: the same techniques that perform well for a small number of nodes are the same techniques that perform well for a large number of nodes. The interpolation error for the full Lagrange interpolation at first decreases the fastest, eventually plateauing at roughly 10^{-10} (likely due to numerical limitations). However, at some point the interpolation error *increases* again due to Runge's phenomenon. The other techniques show consistent improvement as the number of nodes is increased.

The previously used definition of interpolation error only considers the absolute differences between the original function and its interpolation. However, this metric may only be reflective of the parts of the function with a high absolute function value. The parts with a low absolute value may have a high interpolation error relative to their absolute function value without affecting the interpolation error value which only considers the maximum difference. It is therefore important to also inspect the *relative interpolation error* δ_{rel} defined here as

$$\delta_{\text{rel}} = \max \frac{|f(x) - G(x)|}{|f(x)|}, x \in [a, b]. \tag{3.6}$$

Figure 3.7 shows the relative interpolation errors of the previously introduced interpolation techniques for the toy problem as a function of the number of nodes. The behavior for a low number of nodes is slightly different. But the asymptotic behavior of the relative interpolation error as the number of nodes increases is essentially the same as with the absolute interpolation error. This suggests that the absolute interpolation error which only considers the maximum is a good proxy for the behavior in regions with small absolute

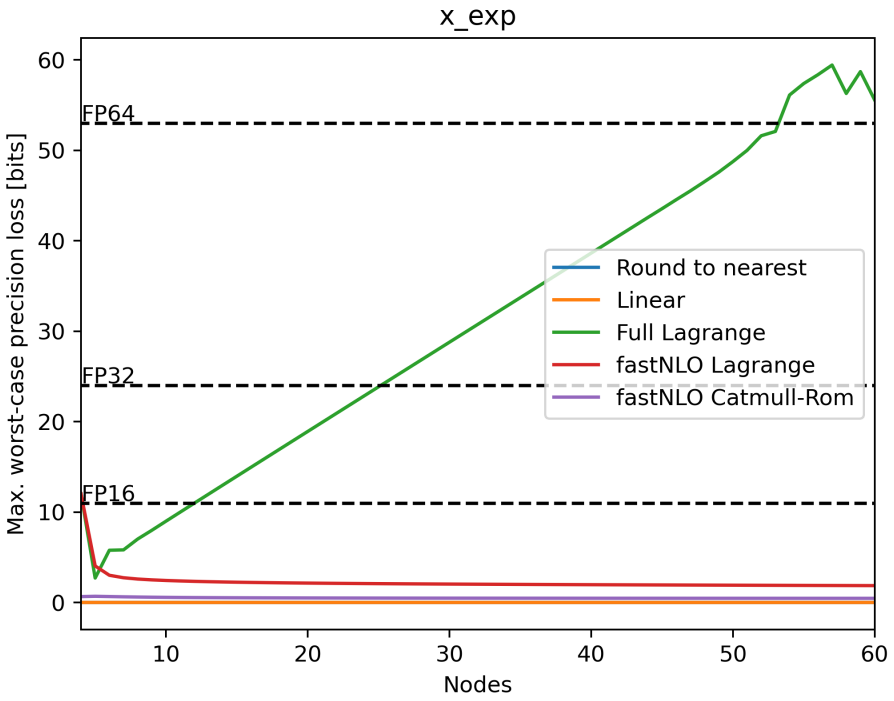

Figure 3.8.: Comparison of interpolation techniques in terms of maximum rounding error from cancellation given the worst-case order of summation. The numerical precision of common IEEE 754 floating point formats has been added for reference. FP16 (half precision) has a precision of 11 bits, FP32 (single precision) has a precision of 24 bits, and FP64 (double precision) has a precision of 53 bits.

value.

Another important aspect to consider is the degree to which numerical cancellation occurs. Because the numerical precision of a floating point number is approximately proportionate to its absolute value the precision loss in bits L_b when adding numbers a and b can be approximated as follows:

$$L_\mathrm{b} \approx \max\left(\log_2|a|, \log_2|b|\right) - \log_2\left(|a+b|\right). \tag{3.7}$$

The interpretation is as follows: After the floating point addition approximately L_b bits of the significand of $a+b$ will be affected by cancellation and contain no useful information regarding the result that would have been obtained without rounding error. The cancellation will be highest for $a \approx -b$ which is equivalent to the scenario in which a and b are subtracted and $|a-b|$ is small compared to $|a|$ and $|b|$. For reference, single-precision floating point numbers following the IEEE 754 format[7] have a precision of 24 bits while double-precision numbers have a precision of 53 bits.

The worst-case precision loss when summing up more than two floating point numbers occurs when first all positive and negative numbers are summed up as partial sums and the final result is then calculated as the sum of those two partial sums. While it is unlikely that this exact pattern would naturally arise in code the same precision loss would occur if the sum were to instead be calculated by summing up a one-dimensional array with a simple for loop if said array is sorted by sign. Figure 3.8 shows the maximum worst-case precision loss of the previously introduced techniques as a function of the number of nodes. Rounding to the nearest node introduces no precision loss because the technique does not involve summation. Linear interpolation only sums up non-negative terms for the

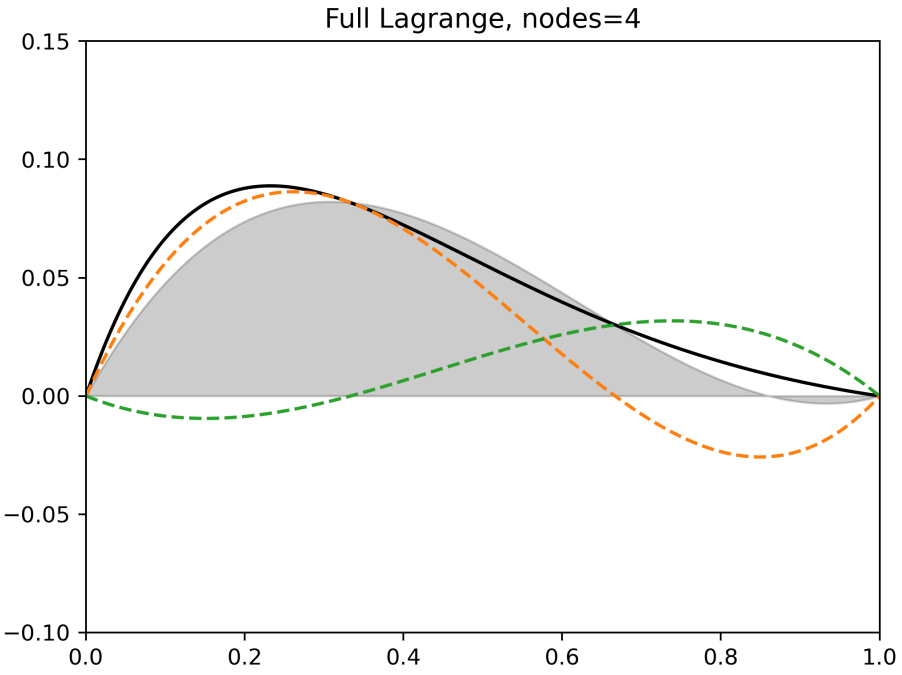

Figure 3.9.: Interpolation of $f(x) = x(1-x)\exp(-3x)$ using full Lagrange interpolation (and only four nodes).

toy example because the function is non-negative on the interval $[0, 1]$. The sum of the negative terms is therefore always 0 which then implies $L_\mathrm{b} = 0$. The interpolations based on Lagrange polynomials or Catmull-Rom splines introduce cancellation because their weights can become negative for non-negative functions. The precision loss for Catmull-Rom and Lagrange splines is roughly constant with 1 bit and 2.5 bits respectively. The precision loss of full Lagrange interpolation measured in bits increases linearly as the number of nodes increases. However, because the number of values that can be represented by a floating point number increases exponentially with the number of bits this actually means that the rounding error of the final result increases exponentially with the number of nodes. The full Lagrange precision loss seems to stagnate shortly after passing the double precision threshold; presumably this is an artifact of the limited numerical precision of the machine on which the calculation was done.

The precision loss for the interpolation techniques based on Lagrange polynomials is noticeably worse when using only four nodes. This is because in that case the interpolation is bad enough to cause the interpolated function to partially become negative (see Figure 3.9). As a result the sum of the individual terms can become very small which amplifies the cancellation. This also means that if $f(x)$ has zero crossings then the precision loss near those crossings will be comparatively higher.

3.1. Node selection

As previously hinted at with the discussion of Runge's phenomenon, choosing good node values for the interpolation is nontrivial. In practice there are three challenges:

1. Determining the interval on which the nodes should be distributed,
2. determining how the nodes should be distributed on the interval,

Table 3.2.: Transfer functions implemented in fastNLO. x refers to the non-transformed variable while $\hat{x}$ refers to the transformed variable.

Name	Transfer function	Inverse transfer function
`linear`	x	$\hat{x}$
`loglog`	$\ln\left[\ln\left(x\right)\right]$	$\exp\left[\exp\left(\hat{x}\right)\right]$
`loglog025`	$\ln\left[\ln\left(\frac{x}{0.25}\right)\right]$	$0.25 \cdot \exp\left[\exp\left(\hat{x}\right)\right]$
`log10`	$\log_{10} x$	$10^{\hat{x}}$
`sqrtlog10`	$-\sqrt{-\log_{10} x}$	$10^{-\hat{x}^2}$
`3rdrtlog10`	$-\sqrt[3]{-\log_{10} x}$	$10^{-\hat{x}^3}$
`4thrtlog10`	$-\sqrt[4]{-\log_{10} x}$	$10^{-\hat{x}^4}$

3. and determining how many nodes are needed for a sufficiently small interpolation error.

Due to the interplay of these three challenges the approach used in fastNLO is best explained starting from the second point.

As stated before, equidistantly spaced nodes can be a suboptimal choice. Therefore fastNLO has implemented several *transfer functions* that transform the interval on which the parton distribution functions are interpolated to another interval. Nodes are then spaced equidistantly on the transformed interval which results in non-equidistant spacing on the non-transformed interval. This non-equidistant spacing reduces the interpolation error. The transfer functions implemented in fastNLO are shown in table 3.2. By default `sqrtlog10` is used for the momentum fraction and `loglog025` is used for the scales. Notably `log10`, `sqrtlog10`, `3rdlog10`, and `4thlog10` transform the bounded interval $[0, 1]$ to the unbounded interval $(-\infty, 0]$. When using these transfer functions for the momentum fraction x the nodes are distributed more densely at low x values (and in fact become infinitely dense for $x \to 0$). This is because for actual parton PDFs a better momentum fraction resolution at low values is desirable. However, these transfer function open up a new problem: determining the lower bound for x nodes.

To get back to the first point, the lower bound for the momentum fraction can be estimated by doing a so-called "warmup" run. In this mode only the minimum and maximum values of the momentum fraction and the scales that occur are being recorded. Then, for the actual "production" run those minimum and maximum values can be used to set the minimum and maximum node values.

This then only leaves the number of nodes that should be used for the interpolation as the last parameter. One option is to simply set a constant number of nodes per observable bin (called `NodesPerBin` from now on). Unfortunately there is no automatic way of determining a good value for this parameter; the values used in production are based on experience and experimentation. Generally the momentum fraction requires something like 15-30 nodes per bin while the scales require about 6. Alternatively the number of nodes can be set per magnitude spanned by the warmup min./max. values. In the latter case the number of nodes provided by the user is scaled up by a factor of $\log_{10} x_{\max} - \log_{10} x_{\min}$ (rounded down). This mode will in the following be referred to as `NodesPerMagnitude`.

3.2. Monte Carlo Integration

So far this chapter has only explained how a PDF (or a function in general) can be interpolated. However, it is important to also understand the interplay between this interpolation, the Monte Carlo integration, and the actual calculation of a cross section

given an a posteriori PDF choice. As described in chapter 2 the cross section can be calculated as

$$\int \sigma(\mu) d\mu \approx \sum_{i,j,k,n} f(x_i, \mu_k)\ f(x_j, \mu_k)\ \alpha_s^n(\mu_k)\ \tilde{\sigma}_{ijkn}, \tag{3.8}$$

with the coefficients $\tilde{\sigma}_{ijkn}$ defined as:

$$\tilde{\sigma}_{ijkn} = \sum_{a,b} \int d\mu \int_0^1 dx_1 \int_0^1 dx_2\ c_{a,b,n}(x_1, x_2, \mu)\ g_i(x)\ g_j(x)\ h_k(\mu). \tag{3.9}$$

In practical terms this means that instead of calculating a single Monte Carlo integral for a given PDF instead many Monte Carlo integrals for the function products $g_i(x)g_j(x)h_k(\mu)$ are calculated. The results of these integrations are then written to the coefficient table.

When an event is generated for a Monte Carlo integration that event would in principle affect the entire coefficient table. However, when e.g. using the fastNLO Lagrange splines for interpolation almost all $g_i(x)$, $g_j(x)$, and $h_k(\mu)$ equal to zero. The event only affects those coefficients whose nodes are close to the event. In this way the weight of the MC event is "spread" across multiple coefficients. With the fastNLO Lagrange splines four coefficients per dimension are affected which would equal to $4^3 = 64$ coefficients. As implied by eq. 3.9 the final weight for a specific $\tilde{\sigma}_{ijkn}$ can then be obtained by simply multiplying the event weight with the function values $g_i(x)$, $g_j(x)$, and $h_k(\mu)$ (with x and μ taken from the event).

4. `NodeDensity`: A New Method for x Node Spacing

Section 3.1 has described the preexisting methods for determining momentum fraction x nodes x_i from a warmup run. This chapter conceptually describes the new method `NodeDensity` added by this thesis which determines the nodes x_i *without* such a warmup run. Appendix A contains the implementation details; they are largely invisible to regular users but provide relevant information for developers.

The main motivation for developing methods that do not require a warmup run is to reduce the amount of manual effort needed for creating coefficient tables by removing one of the steps in the workflow. A secondary benefit is that the CPU hours needed for the warmup run can also be saved (though compared to the production run the warmup run is relatively fast anyways). For compute clusters workflows with fewer jobs are also advantageous for reducing the clock time needed for creating coefficient tables by reducing the overhead from e.g. jobs waiting to be scheduled or the user checking the warmup results before submitting further jobs.

The approach for `NodeDensity` is to define x nodes via a constant density in transformed x space (the space transformed by the transfer functions in table 3.2). The highest node is placed at $x = 1$ in non-transformed space. In the space transformed with `sqrtlog10` (the default for the momentum fraction) this corresponds to $\hat{x} = 0$. All other nodes are then defined inductively by placing it a set distance below the previous node (in transformed space).

Because no warmup run was performed there is no estimate for the lowest x value that will be encountered during the production run. It is therefore necessary to dynamically extend the x nodes and the x dimension of the coefficient table $\tilde{\sigma}$ towards lower x values as they are encountered. Conceptually this makes $\tilde{\sigma}$ infinitely large in the x dimension. However, after the production run all $\tilde{\sigma}$ value below a certain x index (which depends on the minimum x value encountered during the production run) will be zero. And because the contributions to the actual cross section in eq. 2.4 are proportional to $\tilde{\sigma}$ any zero values can be ignored. It is therefore sufficient to store the finite range of $\tilde{\sigma}$ values from the lowest (in terms of x) nonzero value to the upper bound at $x = 1$.

Figure 4.1 compares the distribution of nodes for `NodesPerBin` and `NodeDensity` for a toy example with two bins. The first bin stands for the case of high transverse momentum p_T and low absolute rapidity $|y|$ which necessitates that both partons have a high momentum fraction. The minimum momentum fraction for events in this bin is therefore high. The second bin stands for low p_T and high $|y|$ which necessitates that the momentum fraction of one of the partons is high while that of the other parton is low. The minimum momentum fraction for events in this bin is therefore low. Also, the events in the second bin form a double peak structure in x that for the same number of nodes is expected to have a higher interpolation error than the single peak structure in the first bin. In order to minimize the

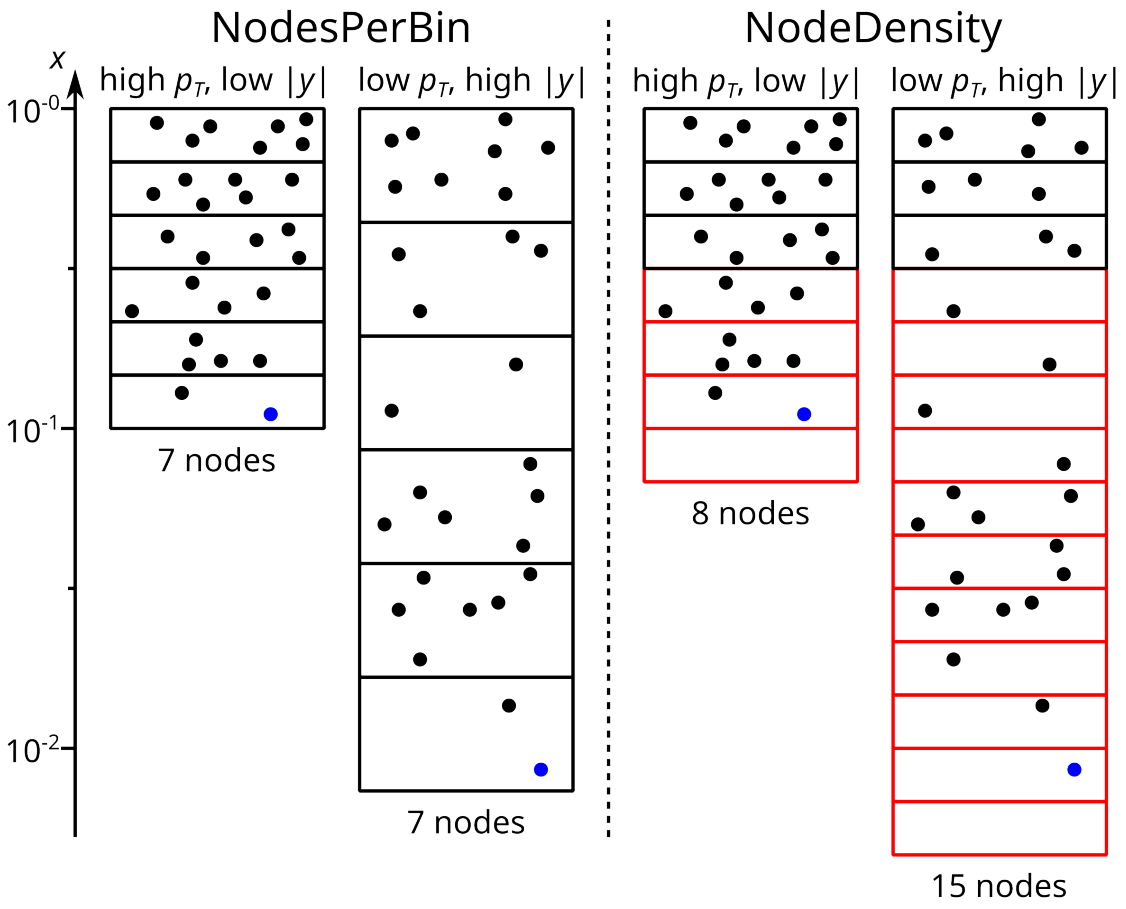

Figure 4.1.: Comparison of `NodesPerBin` and `NodeDensity` for a toy example with only two observable bins. The nodes are placed at the x values corresponding horizontal lines of the boxes. The circles represent individual events. The blue circles represent the event with minimum momentum fraction. The nodes marked as red are the ones added dynamically during the production run as new minimal x values are encountered.

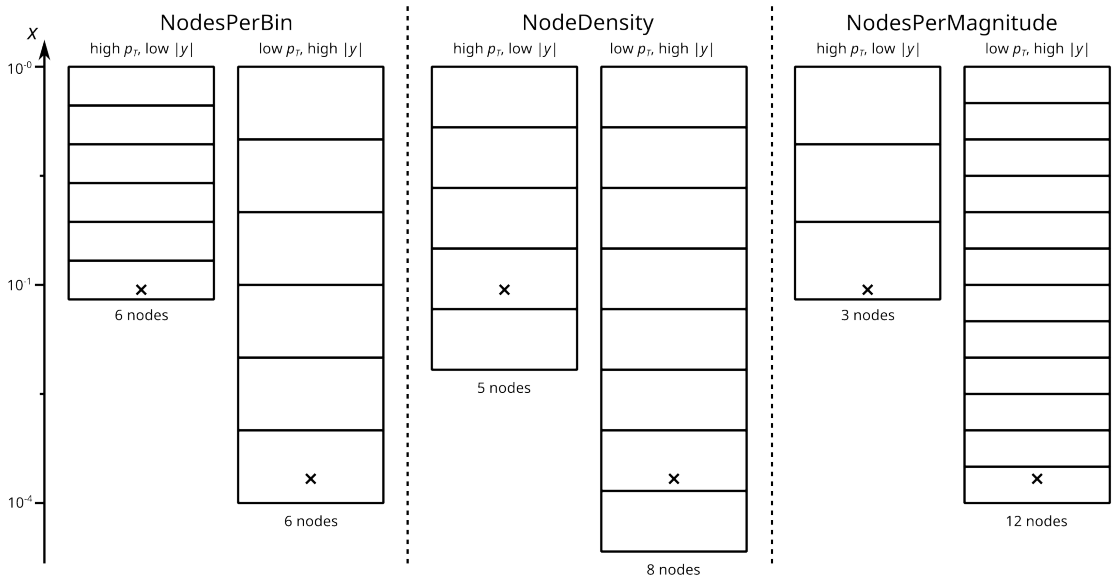

Figure 4.2.: Comparison of `NodesPerBin`, `NodeDensity`, and `NodesPerMagnitude` for a toy example with only two observable bins. The nodes are placed at the horizontal lines of the boxes. The crosses represent the events with minimum momentum fraction.

maximum interpolation error across all bins it would therefore be desirable to assign more nodes to the second bin. At the same time assigning more nodes to the first bin would not affect the maximum interpolation error at all. It is therefore desirable to scale the number of nodes assigned to each bin with the minimum x value of that bin. `NodesPerBin` does not do this. Instead all observable bins receive the same number of nodes. When the number of nodes for the worst bin is increased in order to reduce the maximum interpolation error, the number of nodes for all other bins is also increased. This provides no benefit for reducing the maximum interpolation error but it greatly increases the overall size of the coefficient table and the associated resource usage.

By contrast `NodeDensity` can be expected to automatically assign more nodes to those observable bins with low minimum x values: due to the constant density of x nodes in transformed space the total number of nodes between a low minimal x value and $x = 1$ will be higher than the total number of nodes between a high minimal x value and $x = 1$.

At this point it should be noted that the preexisting method `NodesPerMagnitude` also automatically assigns more nodes to those observable bins with low minimum x values. Figure 4.2 shows a comparison between `NodesPerBin`, `NodeDensity`, and `NodesPerMagnitude`. The transfer function used for this toy example is `sqrtlog10`. As a consequence `NodesPerMagnitude` assigns more nodes to bins with low minimum x values than `NodeDensity` does. If `log10` were to be used `NodeDensity` and `NodesPerMagnitude` could be configured to result in the same nodes.

As can be inferred from figures 4.1 and 4.2, compared to `NodesPerBin` and `NodesPerMagnitude` `NodeDensity` adds an additional node below the minimum momentum fraction. This is because when using Lagrange or Catmull-Rom interpolation kernels (regardless of the method for determining the x nodes) the weight of an event is spread across four nodes (see figure 4.3). In the center of the coefficient table these are the two nodes directly adjacent to either side of an event in the interpolated dimension. However, the behavior at the edges of the grid needs to be different because on one side of the event there is only a single node. To compensate three instead of two nodes from the other side are used. If the size of the grid is constant then this is a minor detail. However, if the size of the grid is

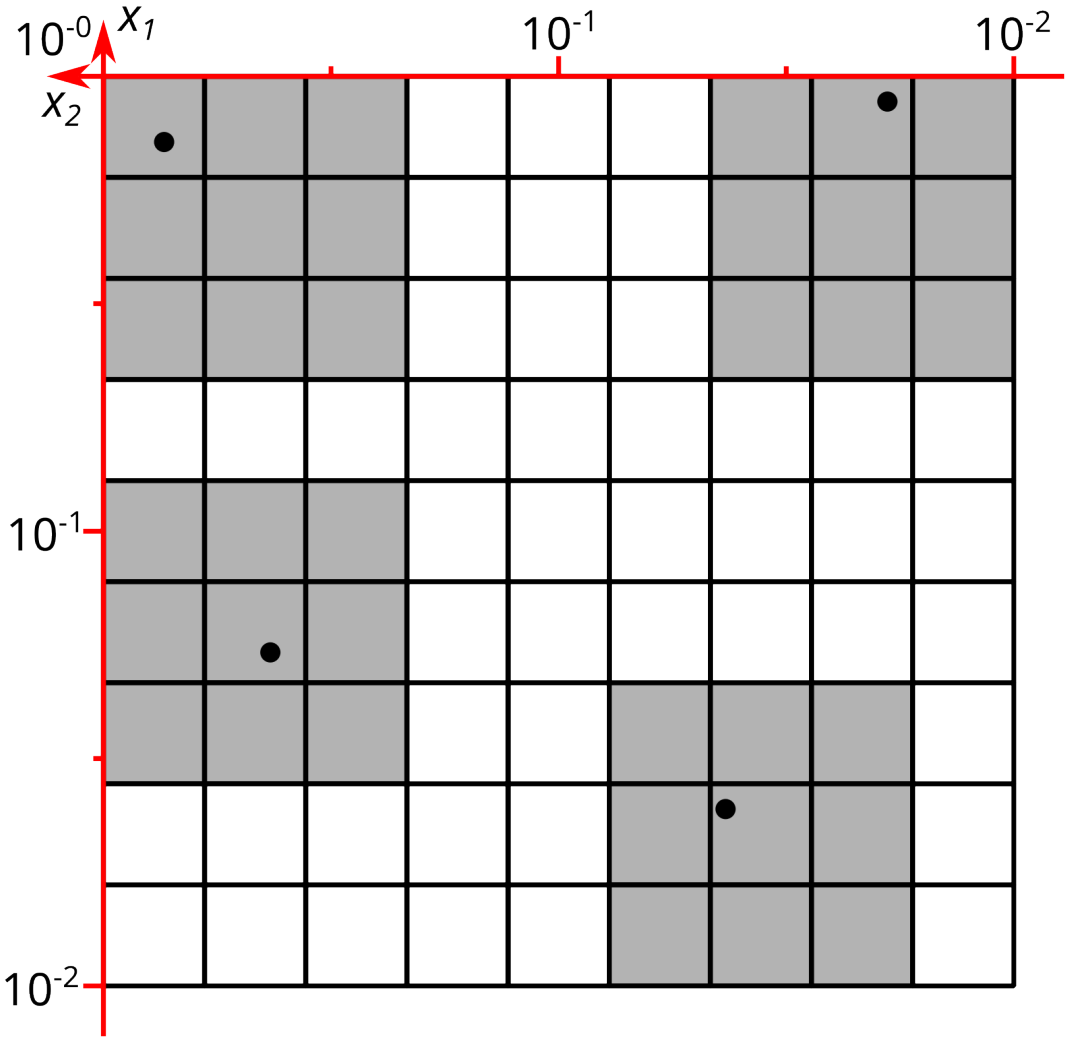

Figure 4.3.: Visualization of how the weight from single events (black points) is spread across nodes (grid intersections) in two dimensions x_1 and x_2 using Lagrange splines. At the edges of the grid (red lines) the event weight spread across four nodes per dimension is asymmetric. When using node density there is no lower edge so the weight is spread symmetrically.

dynamic towards small x values then there is no actual lower bound. At any x value the two adjacent x nodes to either side are affected. Therefore, for the same event x value the weight of the event is spread to lower x nodes for `NodeDensity` compared to `NodesPerBin` and `NodesPerMagnitude`. As a consequence the coefficient table as a whole ends up slightly larger when using `NodeDensity`.

4.1. Scale nodes

So far the focus has been on the momentum fraction nodes. This is because the interpolation of PDFs in that dimension is much more difficult than in the scale dimensions. Not only is the total number of nodes needed much higher (roughly 2.5-5 times) but furthermore the total grid size for proton-(anti)proton collisions scales quadratically with the number of x nodes because for each event there are two separate momentum fractions. Still, the scale dimensions need to be taken into account. Compared to the momentum fraction the main difference is that there is no natural upper bound. Since the scales are measured in units of energy there is theoretically a lower bound at 0. But because the strength of α_s increases at low energy scales the underlying assumption of asymptotic freedom is then no longer valid. As a consequence perturbative QCD cannot be used for theory predictions at low energies; for this reason the lower bound at 0 is not useful. This makes the scale dimensions effectively unbounded in either direction. Because of this there is no clear starting point for defining nodes which complicates a general implementation of node density for the scales. As of writing there is no `NodeDensity` implementation for the scales.

With the node density implementation for the momentum fraction the warmup run is only needed for the scales. So ideally, if there is a way to determine the min./max. scale values without a warmup run then the warmup run can be skipped entirely. Luckily there are instances where the scale choice is equal to the binning choice (e.g. dijet mass m_{12}). In those cases the min./max. scale values can simply be read from the binning and no warmup run is necessary for the scales. Unfortunately there are also cases where a non-observable quantity is used as an energy scale (e.g. the sum of parton transverse momenta H_T^*). In those cases a warmup run is still necessary.

5. Results

This chapter details the results of the `NodeDensity` implementation. The most important result is that part of the workflow can now be automated. In addition the refactoring has made the code simpler which will reduce maintenance for the fastNLO project. Actually measuring the amount of person time saved with these improvements is out of scope for this thesis. Instead this chapter will present measurements relating to computational resources that can be collected with relative ease via benchmarks. The first measurement is the code speedup from the table filling code overhaul as well as an investigation of whether more optimizations of the fastNLO code would be worthwhile. The second measurement is the interpolation quality as a function of coefficient table size for `NodeDensity` compared to the preexisting methods `NodesPerBin` and `NodesPerMagnitude`.

5.1. Performance optimization

As detailed in appendix A, as part of the code changes for this thesis the code for filling fastNLO coefficient tables was refactored. The primary goal was to simplify the code and make maintaining the project easier. But as a side effect the code also became more performant.

This first became apparent when running simple test productions at leading order. The time needed for a flexible-scale dijet production run with 10^6 events and NLOJet++ as the event generator improved from 110 s to only 23.2 s, resulting in a speedup of 4.7 on a consumer platform with an AMD Ryzen 3700X processor. The speedup comes from a more efficient use of partial results when multiplying the interpolation weights of multiple dimensions, thus reducing the amount of floating point arithmetic. The time needed for fixed-scale tables was essentially unchanged: on the test machine the runtime improved only marginally from 12.0 s to 11.8 s.

However, despite this measured speedup for flexible-scale tables one cannot expect a universally significant improvement in runtime. To explain why, *Ahmdahl's law*[25] is considered. It assumes a system with multiple parts where one part initially takes up a fraction p of the runtime and is sped up by a factor of s. The speedup S of the system as a whole then becomes:

$$S = \frac{1}{1 - p + \frac{p}{s}}. \tag{5.1}$$

Notably this law implies $S < \frac{1}{1-p}$ regardless of s. In other words, the speedup of the system as a whole is limited by the fraction of the runtime that the optimized part takes up. In the case of the test production almost the entire runtime is taken up by filling fastNLO tables ($p \approx 1$). However, this is to a large part because LO calculations with NLOJet++ are fast, so the event generator takes up only a small percentage of the runtime with $s \approx S = 4.7$. However, for state-of-the-art NNLO calculations the event generator can be expected to take up a much larger percentage of the runtime. It is also likely that the simple test code

Table 5.1.: Profiling data for NNLOJET rev5918, nnlo-bridge v0.0.40

Subprocess	Runtime [s]	Incl. bridge + fastNLO [s]		Incl. fastNLO [s]	
LO	0.658	0.086	13.01%	0.053	7.98%
V	2.383	0.621	26.08%	0.535	22.46%
R	6.259	2.536	40.52%	1.950	31.15%
NLO	9.300	3.243	34.87%	2.538	27.29%
VV	6.999	0.379	5.42%	0.327	4.67%
RV	57.732	5.912	10.24%	4.624	8.01%
RRa	114.915	40.565	35.30%	31.452	27.37%
RRb	114.430	40.256	35.18%	32.395	28.31%
NNLO	303.376	90.355	29.78%	71.336	23.51%

is simply not sufficiently optimized in terms of reducing the number of fastNLO function calls to a minimum, thus spending more runtime than necessary in fastNLO code. For these reasons the parameter p can be expected to be lower in production code which then in turn reduces the observed speedup S.

To then conclude whether or not further fastNLO performance optimizations would be worthwhile it is imperative to determine the parameter p for NNLO, i.e. what percentage of the total runtime is used by fastNLO for NNLO calculations. If p is low then further performance optimizations are not worthwhile. The tool used for this purpose is Valgrind[6]. It can be used to "profile" programs, i.e. to determine which percentage of the total runtime the program spends in individual functions or libraries. There are two primary runtime percentages determined by Valgrind: "self" and "inclusive". Self refers to the amount of runtime spent in a function/library while *excluding* the runtime from any sub-calls within the function. Inclusive refers to the amount of runtime spent in a function/library while *including* the runtime from any sub-calls within the function. For determining whether or not more performance optimizations would be worthwhile the inclusive runtime percentage was used. This is because the performance of a function can be improved not only by improving the efficiency of the code in the function itself but also from reducing the number of calls to expensive other functions. Furthermore the inclusive percentage is at least as high as the self percentage by definition. Therefore, if optimizations are not worthwhile under the inclusive percentage they would also not be worthwhile under the self percentage. In addition to determining whether more performance optimizations would be worthwhile the profiling investigation also intended to benchmark different software configurations and estimate the additional overhead from filling fastNLO tables.

At the time of the investigation the only fastNLO-compatible event generator capable of NNLO calculations was NNLOJET. For NNLO calculations the contributions to the coefficient table are split into leading order, virtual, real, virtual-virtual, real-virtual, real-real a, and real-real b with each contribution requiring a separate run (results are merged afterwards). Notably in a real analysis the number of events generated is *not* the same per contribution. This is because it is more economical to put more CPU time towards those contributions that are cheap. However, the computational cost of the event generator also affects p, the percentage of the runtime spent on fastNLO. Therefore the number of events for each contribution was taken from a real dijet analysis at a center-of-mass energy of 7 TeV (but scaled down by a factor of 10^6) to ensure that the runtime percentages of the combined calculation are representative. Table 5.1 shows the profiling data when using NNLOJET rev5918 to create a fastNLO coefficient table. The runtimes were obtained by simply running the NNLOJET binary. Profiling data had to be obtained in separate runs because the profiling interferes with the runtime measurement, causing the program to

Table 5.2.: Leading color profiling data for NNLOJET rev6591, nnlo-bridge v0.0.46

Subprocess	Runtime [s]	Incl. bridge + fastNLO [s]		Incl. fastNLO [s]	
LO	0.696	0.092	13.20%	0.062	8.91%
V	2.154	0.124	5.77%	0.082	3.83%
R	12.407	1.040	8.38%	0.605	4.88%
NLO	15.257	1.256	8.23%	0.749	4.91%
VV	9.710	0.201	2.07%	0.142	1.46%
RV	180.825	4.014	2.22%	2.387	1.32%
RRa	159.221	4.188	2.63%	2.341	1.47%
RRb	183.860	5.993	3.26%	3.659	1.99%
NNLO	548.873	15.652	2.85%	9.278	1.69%

become roughly 50 times slower.

The original runtimes of the individual contributions ranged from seconds (LO) to minutes (RR). For profiling this presented an issue because at very short runtimes parts of the code that do not scale with the number of events (e.g. file reads) can take up a large portion of the total runtime and skew the results. Therefore, for LO, V, R, and VV the number of events was scaled up for the profile run (so that the program runs for at least 60 seconds). This was done to suppress the parts of the code with constant runtime, making the percentages representative of a run with a large number of events as would be the case for an actual physics analysis. The runtimes for NLO and NNLO (both total and for parts of the program) were determined by summing up the runtimes of the respective individual contributions. The corresponding percentages were then determined from those sums.

The procedure described above where the runtime fractions are determined from separate runs with a scaled up number of events biases the fractions towards higher values. However, because the total runtime for NNLO is dominated by the contributions that were not scaled up this bias is negligible when considering NNLO as a whole. For NNLOJET rev5918 29.78% of the combined runtime was spent in the "bridge" code, the code responsible for filling either a fastNLO or an APPLGrid table. Filling a fastNLO table with this revision therefore increases the total runtime by roughly 40% (compared to just calculating a cross section for a given PDF). Further, 23.51% of the total runtime was spent in the actual fastNLO code which would limit the speedup from further fastNLO optimizations to $S \leq 1.31$.

Table 5.2 shows the equivalent profiling results for NNLOJET rev6591. The first thing to note is that the total runtime has increased by a factor of 1.81. At the same time the runtime spent in bridge/fastNLO code has decreased by a factor of 7. Taken together the overhead from calculating fastNLO tables has decreased significantly to only 2.85% of the total runtime. Further fastNLO performance optimizations would be insignificant with a speedup of $S \leq 1.017$.

Notably NNLOJET rev6591 has added new configuration options for "leading color" and "full color" calculations. The difference is that leading color only adds contributions to matrix elements from those Feynman diagrams which are not suppressed by powers of the number of color charges $\mathrm{nc}^2 = 3^2 = 9$ or $\mathrm{nc}^4 = 3^4 = 27$. By contrast full color does include those contributions. In rev5918 those contributions were not implemented. The rev5918 results are therefore comparable to leading color rev6591.

Tables 5.3 and 5.4 show the profiling data for full color calculations. When switching only to full color the total runtime increases by a factor of 5.66. However, NNLOJET rev6591 has also added a new performance optimization option: "multichanneling". This

Table 5.3.: Full color profiling data for NNLOJET rev6591, nnlo-bridge v0.0.46

Subprocess	Runtime [s]	Incl. bridge + fastNLO [s]		Incl. fastNLO [s]	
LO	0.730	0.096	13.16%	0.065	8.88%
V	2.157	0.124	5.76%	0.082	3.82%
R	12.359	0.597	4.83%	0.582	4.71%
NLO	15.246	0.817	5.36%	0.729	4.78%
VV	20.731	0.369	1.78%	0.251	1.21%
RV	1700.880	13.607	0.80%	7.994	0.47%
RRa	672.888	17.629	2.62%	9.689	1.44%
RRb	696.900	22.719	3.26%	14.008	2.01%
NNLO	3105.745	55.141	1.78%	32.671	1.05%

Table 5.4.: Full color profiling data for NNLOJET rev6591 with multi_channel set to true, nnlo-bridge v0.0.46

Subprocess	Runtime [s]	Incl. bridge + fastNLO [s]		Incl. fastNLO [s]	
LO	0.676	0.010	1.52%	0.010	1.00%
V	0.691	0.021	2.99%	0.014	2.01%
R	0.971	0.068	7.04%	0.040	4.12%
NLO	2.338	0.099	4.23%	0.064	2.74%
VV	1.084	0.017	1.55%	0.011	1.06%
RV	10.086	0.179	1.77%	0.103	1.02%
RRa	14.809	0.244	1.65%	0.144	0.97%
RRb	15.282	0.319	2.09%	0.192	1.26%
NNLO	43.599	0.858	1.97%	0.514	1.18%

option optimizes Monte Carlo convergence via *importance sampling*[18]: the larger the contribution of a given Feynman diagram is to the total cross section of an observable bin, the more events are generated for said diagram. Conversely Feynman diagrams with smaller contributions receive fewer events. To compensate the contributions are scaled in such a way that the expected cross section remains the same. The effect of this is that the comparatively larger statistical uncertainties from large contributions are reduced at the cost of increasing the statistical uncertainties of smaller contributions. Importantly this reduces the statistical uncertainty on the total result for a given number of events.

With multichanneling enabled the runtime decreases by almost two orders of magnitude so that the runtime ends up 12.59 times lower than the LC rev6591 results, and even 6.96 times lower than the LC rev5918 results. However, this result is not directly comparable to the result without multichanneling because the speed of convergence per event in the configuration file is not the same. Both with and without multichanneling the runtime taken up by the bridge/fastNLO code is low: the total overhead from fastNLO was only 2% of the runtime while the potential for further speedup was $S \leq 1.02$. Notably the runtime needed for RV increased by a disproportionate amount for full color without multichanneling.

Appendix B includes further analysis of the NNLOJET profiling data that was collected in order to aid with the development process but is not of general interest.

5.2. Node Density Efficiency Benchmarks

As mentioned before, there is a tradeoff between the size of fastNLO coefficient tables and the resulting interpolation error: a large table with more nodes will be able to interpolate a PDF with smaller error but will also require more memory and disk space. At the same time the interpolation error beyond some table size becomes negligible due to the presence of other sources of uncertainty. Further increasing the size of tables at that point only incurs additional cost with no benefit. In particular, a larger table would require more memory and disk space to create and calculating a cross section from the table would be slower because more values would need to be summed up.

An important check for determining the quality of a fastNLO table is to investigate the so-called "closure". First, during the creation of a fastNLO table the conventional cross sections that would have been obtained for a given PDF is calculated. Then, once the fastNLO table is finished it is used to calculate cross sections using the exact same PDF. If the difference between the conventionally calculated cross section and the one obtained via interpolation is negligible compared to other sources of uncertainty then it is assumed that this will also be the case for other PDFs.

One metric that can be used to investigate the closure between an expected value a and an obtained value b is their ratio. It is simply defined as:

$$R = \frac{b}{a}, \tag{5.2}$$

with good closure given for $R \approx 1$. However, this definition has the disadvantage that it diverges for $a \leftarrow 0$. The asymmetry As of two values a and b on the other hand is defined as:

$$\mathrm{As} = \frac{a - b}{a + b}. \tag{5.3}$$

If a and b are both non-negative As is bounded to the interval [-1, 1]. The error from fastNLO interpolation is assumed to be negligible compared to other sources of uncertainty if $|\mathrm{As}| < 10^{-3}$ for the conventional and the fastNLO cross sections. Because the interpolation quality generally increases with the number of x nodes and because the interpretation of

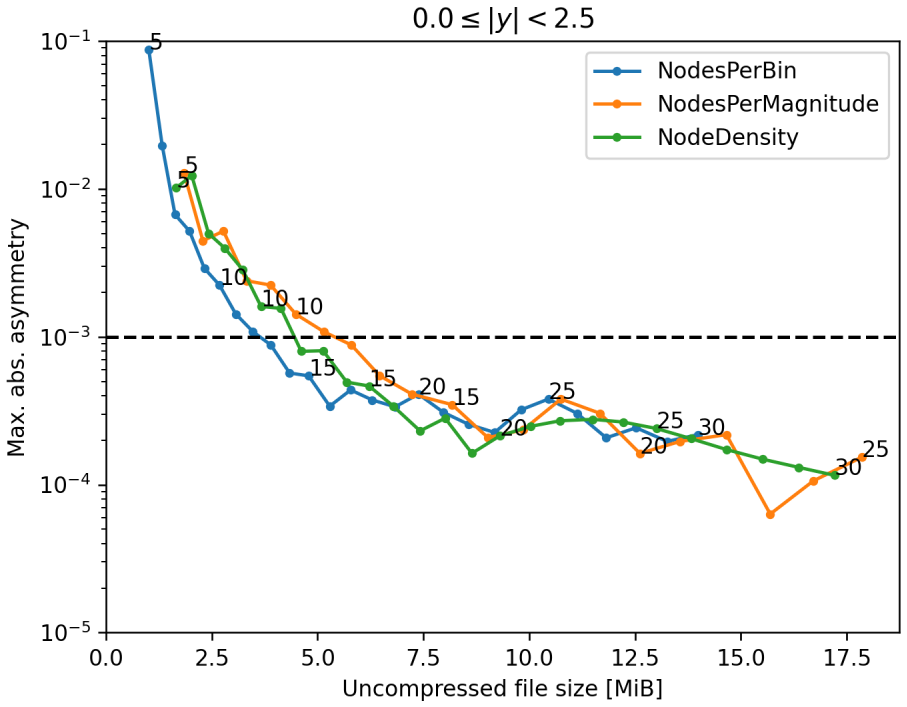

Figure 5.1.: Maximum asymmetry across all observable bins as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

the configuration value `X_NNodes` differs between `NodesPerBin`, `NodesPerMagnitude`, and `NodeDensity` a comparison between the three is not straightforward. For a like-for-like comparison the asymmetry is investigated as a function of the table size. The metric used for determining the table size is the size of the uncompressed fastNLO table files on disk. Because the file contains all relevant information for a fastNLO table this is also a good approximation for the memory use during table filling.

The first investigation is for LO dijet production at 7 TeV binned by dijet mass m_{12} and absolute rapidity $|y| < 2.5$. Figure 5.1 shows the maximum asymmetry across all observable bins as a function of the uncompressed file size (UFS). A method is considered better if it achieves a lower asymmetry at the same UFS value (or needs less disk space for a given asymmetry value). The results are very close and there does not seem to be a method that is universally the best at all file sizes. The asymmetry eventually flattens out at roughly $2 \cdot 10^{-4}$ for all methods, possibly because the number of scale nodes was kept constant at 6 which limits the interpolation precision in the scale dimension. However, at $\mathrm{As} = 10^{-3}$ `NodesPerBin` performs the best with `X_NNodes`=12.4 and UFS=3.64 MiB, followed by `NodeDensity` with `X_NNodes`=11.7 and UFS=4.49 MiB and `NodesPerMagnitude` with `X_NNodes`=11.4 and UFS=5.41 MiB. The values for `X_NNodes` and UFS that correspond to $\mathrm{As} = 10^{-3}$ have been obtained via linear interpolation between individual results. Note that it is not actually possible to define fractional values for `X_NNodes` when using `NodesPerBin` and `NodesPerMagnitude`. However, for comparison purposes the interpolated fractional value is used in order to judge whether the upper or lower next integer is closer to $\mathrm{As} = 10^{-3}$.

Figures 5.2, 5.3, 5.4, 5.5, and 5.6 show the asymmetry as a function of file size for the different rapidity bins. Against expectation the central rapidity bin with $|y| < 0.5$ showed higher maximum absolute asymmetry than the outer rapidity bin with $2.0 \leq |y| < 2.5$ when using the same number of nodes (equal values for `NodesPerBin`). When using `NodesPerBin`

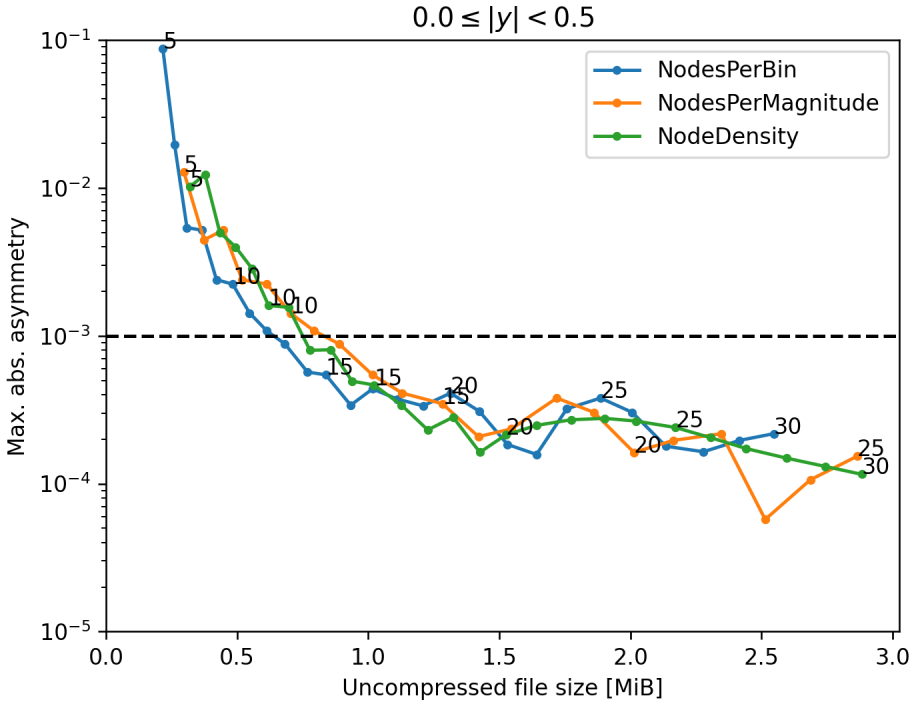

Figure 5.2.: Maximum asymmetry across all observable bins with $0.0 \leq |y| < 0.5$ as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

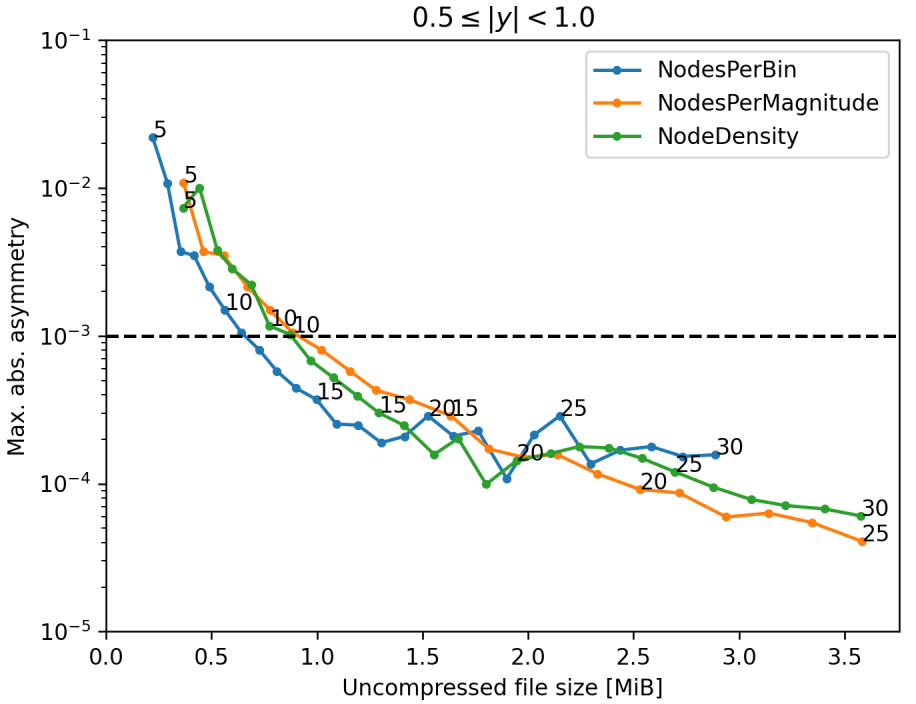

Figure 5.3.: Maximum asymmetry across all observable bins with $0.5 \leq |y| < 1.0$ as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

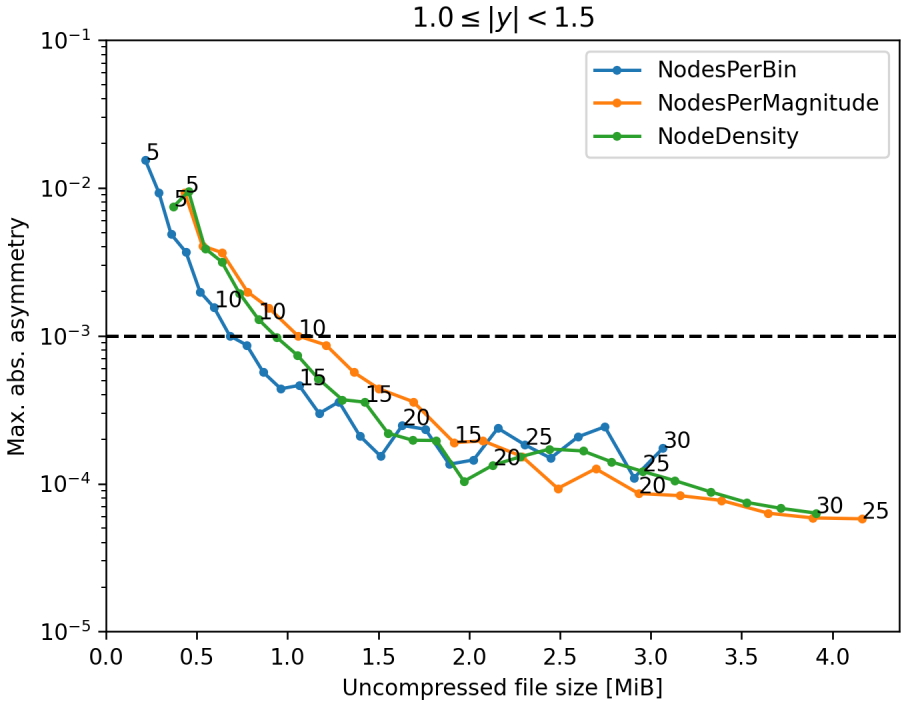

Figure 5.4.: Maximum asymmetry across all observable bins with $1.0 \leq |y| < 1.5$ as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

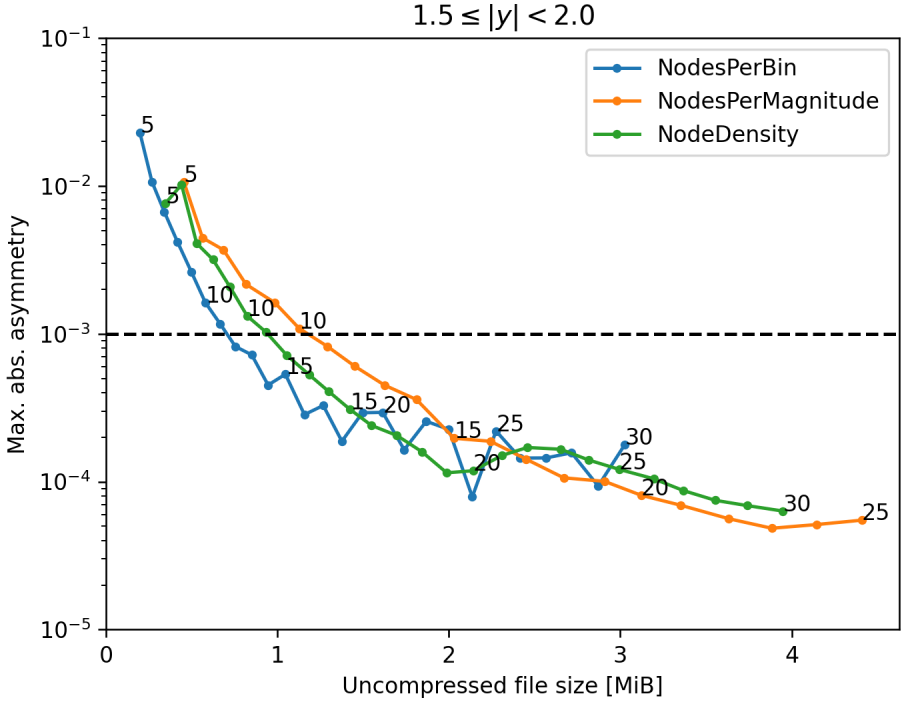

Figure 5.5.: Maximum asymmetry across all observable bins with $1.5 \leq |y| < 2.0$ as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

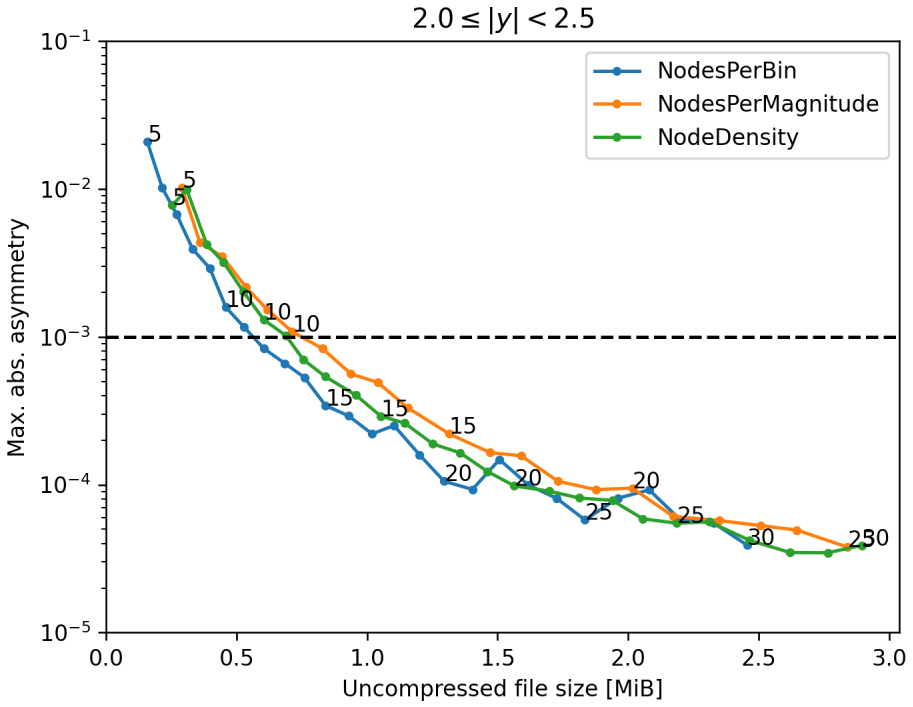

Figure 5.6.: Maximum asymmetry across all observable bins with $2.0 \leq |y| < 2.5$ as a function of the uncompressed file size. The black numbers at function points indicate the value for `X_NNodes`.

the number of nodes per bin is constant. When using `NodeDensity` or `NodesPerMagnitude` the number of nodes per bin increases for bins with low minimum x values (with the increase in `NodesPerMagnitude` being steeper). Because the minimum x values are comparatively high/low at central/outer rapidity, assigning more nodes to bins with low minimum x is counterproductive.

Next Z+jet production at 13 TeV is investigated. The binning is in the transverse momentum of the Z boson p_{TZ} and transformed rapidity bins y_b ("y-boost") and y_* ("y-star"). Given the Z boson rapidity y_Z and the jet rapidity y_{jet} they are defined as:

$$y_b = 0.5 \cdot |y_Z + y_{\text{jet}}|, \quad y_* = 0.5 \cdot |y_Z - y_{\text{jet}}|. \tag{5.4}$$

The transformed rapidity bins have a width of 0.5 and only rapidity bins with $\max y_b + \max y_* < 3.0$ are considered. This leads to a total of 15 rapidity bins (see Figure 5.7).

Figure 5.8 shows the absolute asymmetry as a function of UFS at LO. `NodesPerMagnitude` performs the best with `X_NNodes`=6.8 and UFS=21.6 MiB, followed by `NodeDensity` with `X_NNodes`=12.0 and UFS=26.1 MiB and `NodesPerBin` with `X_NNodes`=23.1 and UFS=33.6 MiB. Notably this is the *exact reverse order* compared to dijet production. For Z+jet the bins with high y_b and y_* were the ones with higher asymmetry (when assigning the exact same number of nodes) so assigning more nodes to those bins (via scaling with minimum x) is beneficial. This proves that among the investigated methods there is not a single one that is universally better than the others for all physical processes.

Also the value for `X_NNodes` with $|\text{As}| = 10^{-3}$ when using `NodeDensity` is much closer to the value found for dijets (12.0 vs. 11.7) compared to `NodesPerBin` (23.1 vs. 12.4) and `NodesPerMagnitude` (6.8 vs. 11.4). While this is irrelevant in terms of the efficiency given the optimal configuration it does make a big difference for actually finding that

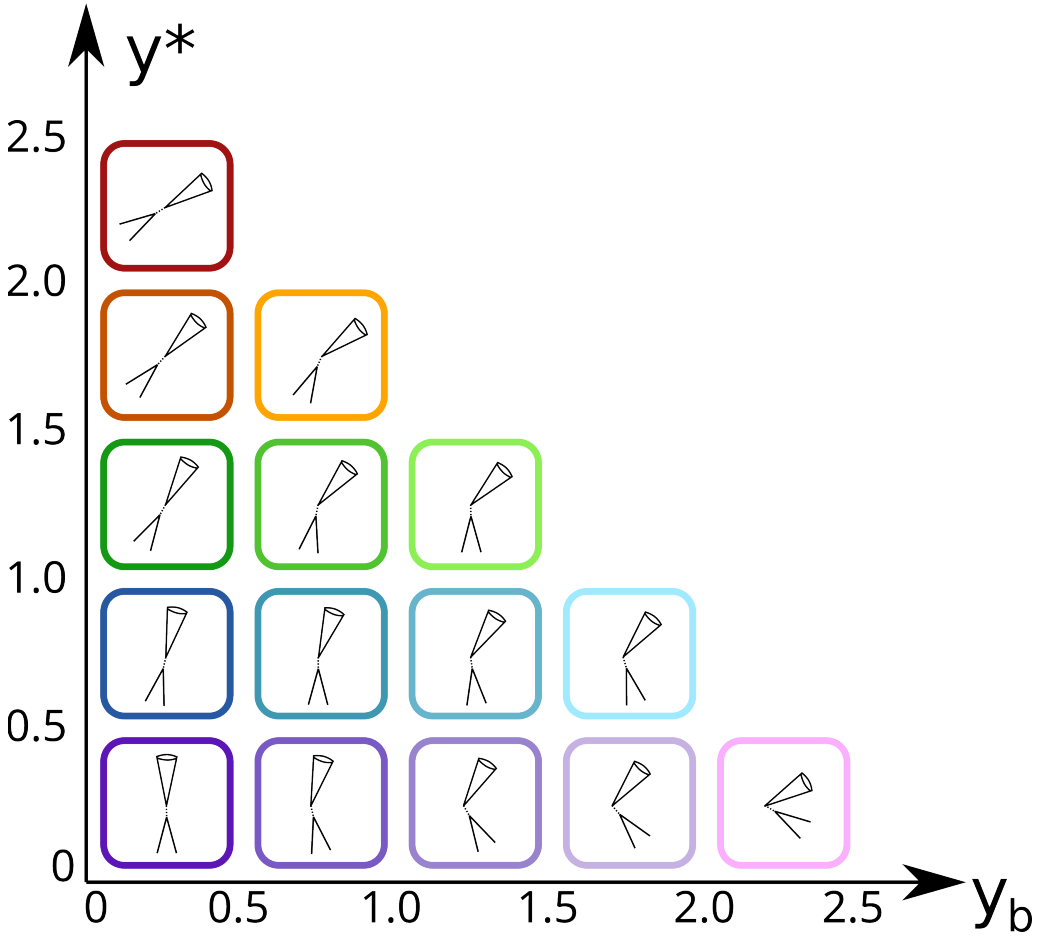

Figure 5.7.: Visualization of used Z+jet rapidity bins in y_b and y_*[17].

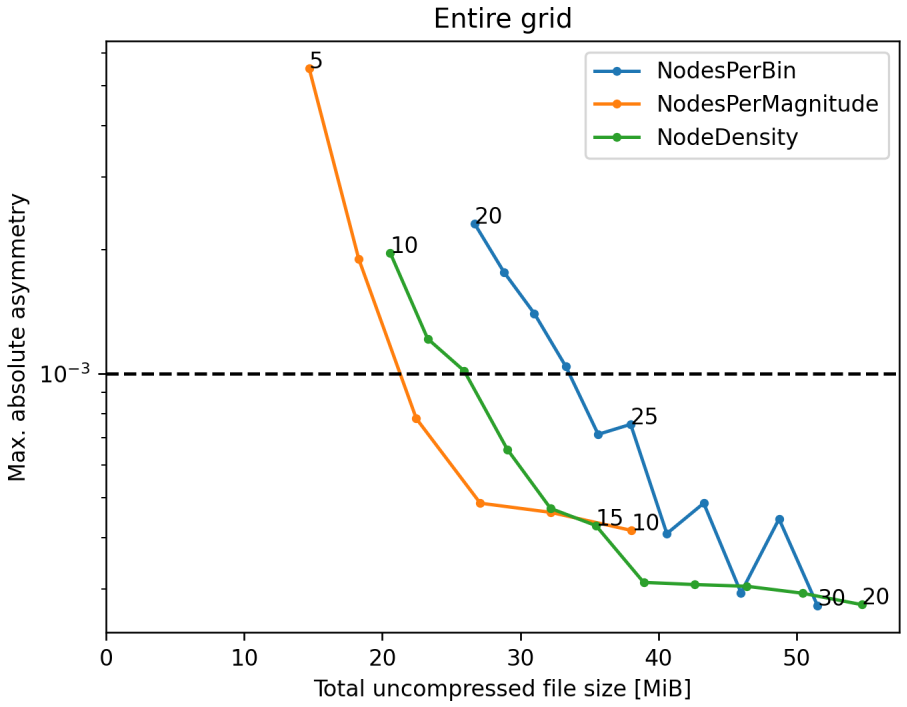

Figure 5.8.: Maximum asymmetry across all Z+jet observable bins as a function of the uncompressed file size at LO. The black numbers at function points indicate the value for `X_NNodes`.

optimal configuration. As mentioned before, there is no known method for predicting the exact value for `X_NNodes` that will result in a targeted precision. Therefore in practice the value for `X_NNodes` is determined iteratively: a user inputs a value that he assumes from experience will work well, and then initiates a test production run and adjusts the `X_NNodes` value based on the results. The larger the spread between optimal values is between different physical processes the more iterations are needed until the optimal (or a sufficiently good) value is found. The seemingly lower spread of `NodeDensity` is therefore a desirable property for reducing the amount of person time needed for producing fastNLO tables. However, due to the low number of investigated processes it is difficult to extrapolate these results.

Z+jet was investigated further because compared to dijets much larger tables are required. Figure 5.9 shows the same plot of max. absolute asymmetry as a function of UFS as before except for NLO rather than LO. None of the methods achieve an asymmetry lower than $2 \cdot 10^{-3}$. The issue is caused by individual bins with high y_b and y_* values (with y_* being more influential). The asymmetry of those bins eventually stops improving as `X_NNodes` is increased. Because of this the max. absolute asymmetry in turn also stops improving once that point is reached. The exact reason why some bins do not further improve is not clear. It could simply be a limitation of the interpolation in the scale dimension (since the number of scale nodes is constant at 6). It could also be an issue with floating point arithmetic since there are more problematic bins with `NodesPerBin` and `NodesPerMagnitude` than with `NodeDensity` (and `NodeDensity` should be slightly less vulnerable to Runge's phenomenon due to not having a hard lower edge).

Figure 5.10 and 5.11 show the 98th and 95th percentiles. These percentiles ignore the problematic bins which do not improve. The ranking of the methods is the same as for LO. When using `NodesPerMagnitude` for the 98th percentile $|\mathrm{As}| = 10^{-3}$ is achieved with

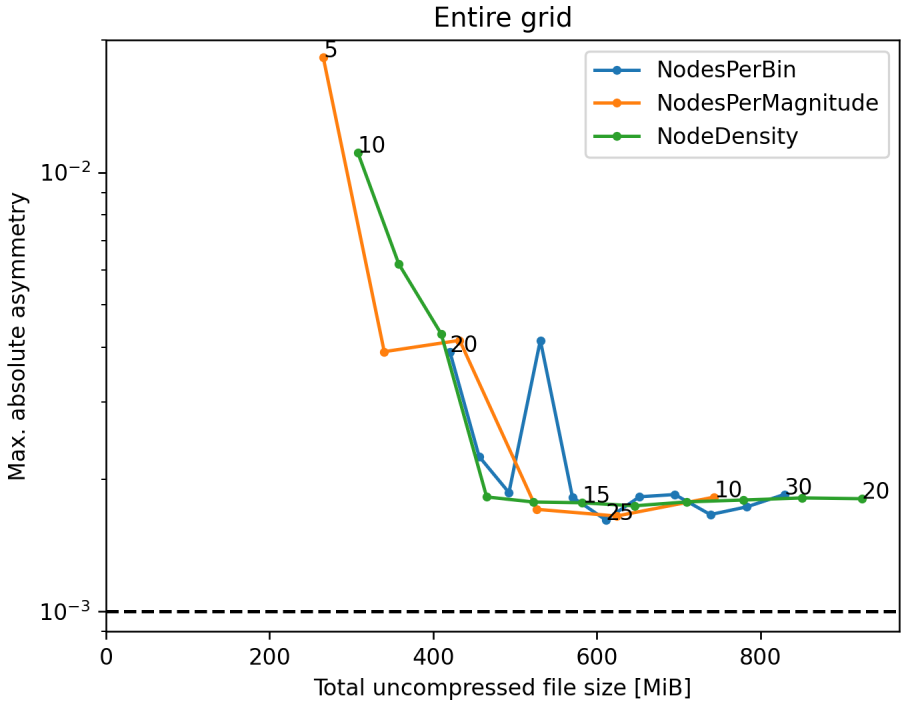

Figure 5.9.: Maximum asymmetry across all Z+jet observable bins as a function of the uncompressed file size at NLO. The black numbers at function points indicate the value for `X_NNodes`.

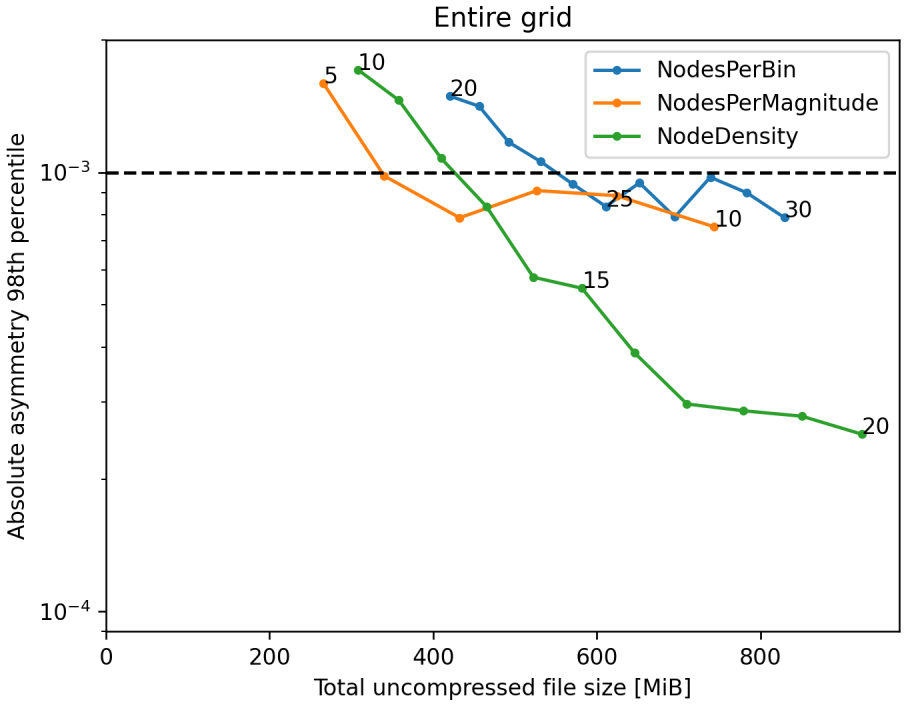

Figure 5.10.: 98th percentile of the asymmetry across all Z+jet observable bins as a function of the uncompressed file size at NLO. The black numbers at function points indicate the value for `X_NNodes`.

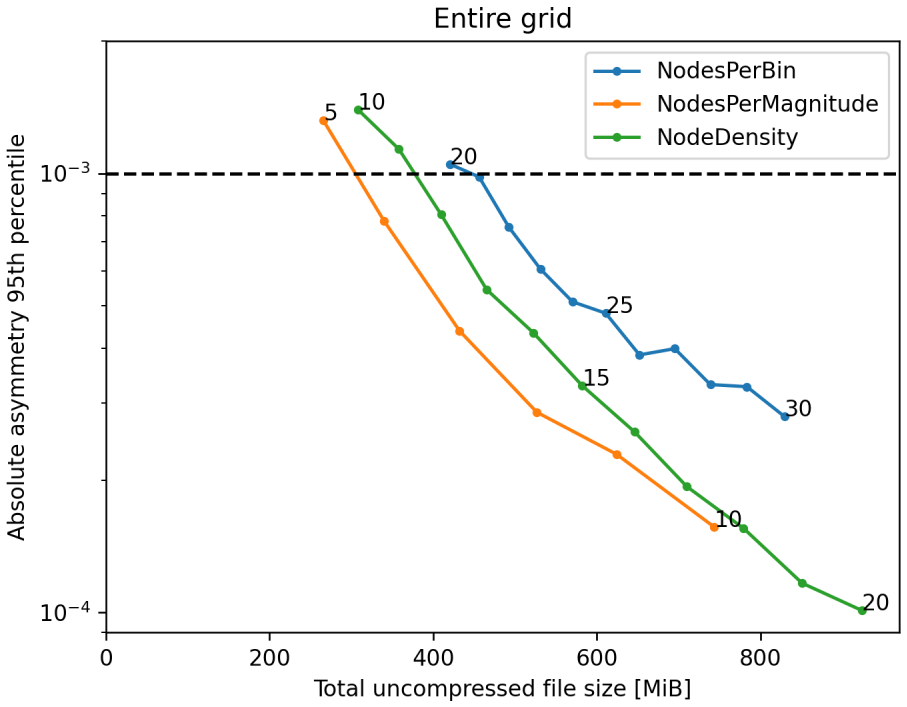

Figure 5.11.: 95th percentile of the asymmetry across all Z+jet observable bins as a function of the uncompressed file size at NLO. The black numbers at function points indicate the value for `X_NNodes`.

`X_NNodes`=6.0 and UFS=338 MiB, when using `NodeDensity` with `X_NNodes`=12.3 and UFS=427 MiB, and when using `NodesPerBin` with `X_NNodes`=23.5 and UFS=551 MiB. For the 95th percentile when using `NodesPerMagnitude` with `X_NNodes`=5.6 and UFS=310 MiB, when using `NodeDensity` with `X_NNodes`=11.4 and UFS=379 MiB, and when using `NodesPerBin` with `X_NNodes`=20.7 and UFS=446 MiB.

Lastly Z production via the Drell-Yan process was investigated at a center-of-mass energy of 13 TeV. The binning is one-dimensional in the Z-boson rapidity ranging from 0.0 to 2.4 in increments of 0.2. Figure 5.12 shows the maximum asymmetry as a function of uncompressed file size. `NodesPerMagnitude` again performs the best with `X_NNodes`=6.97 and UFS=0.47 MiB, followed by `NodesPerBin` with `X_NNodes`=25.01 and UFS=0.55 MiB, and `NodeDensity` with `X_NNodes`=14.37 and UFS=0.59 MiB. The ranking is again different compared to dijets and Z+jet. The values for `X_NNodes` at which the target asymmetry is reached are comparably similar to Z+jet.

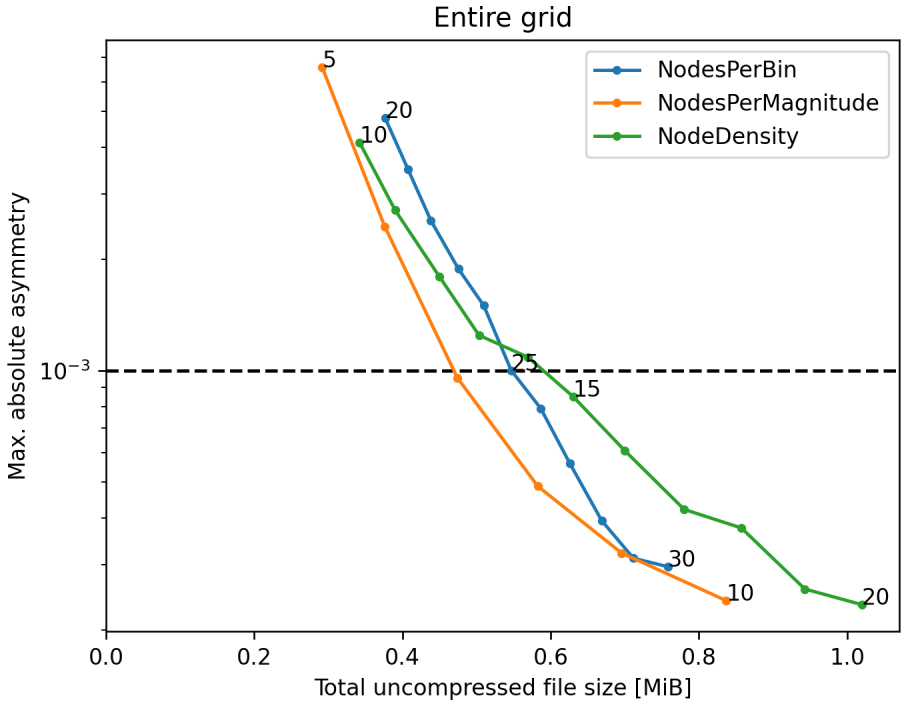

Figure 5.12.: Maximum asymmetry across all Drell-Yan observable bins as a function of the uncompressed file size at LO. The black numbers at function points indicate the value for `X_NNodes`.

6. Conclusion

In chapter 2 the theoretical background behind the fastNLO project has been explained. Via the interpolation of parton distribution functions the results of a Monte Carlo integration can be reused for fast cross section calculations. The PDF, the value of the strong coupling constant α_s, and potentially the values of the scales can be chosen a posteriori. In chapter 3 several interpolation techniques (nearest neighbor, linear, Lagrange, Lagrange spline, Catmull-Rom spline) have been introduced. The techniques were benchmarked in terms of interpolation error, relative interpolation error, and worst-case floating point cancellation on a synthetic problem. The preexisting procedures `NodesPerBin` and `NodesPerMagnitude` have been described. In chapter 4 the new procedure for selecting interpolation nodes `NodeDensity` has been described conceptually and contrasted with the preexisting procedures `NodesPerBin` and `NodesPerMagnitude`. Finally chapter 5 has shown a 4.7 times speedup of the refactored fastNLO table filling code when filling LO flexible-scale tables with NLOJet++. To determine the optimal settings and whether further performance optimizations would be worthwhile NNLO coefficient table creation using NNLOJET was profiled. Using the newest versions for the software stack only 1% of the runtime was spent in fastNLO, making further performance optimizations of the fastNLO code not worthwhile. The new `NodeDensity` implementation was tested for LO dijet, NLO Z+jet, and LO Z production and contrasted with `NodesPerBin` and `NodesPerMagnitude`. None of these methods has consistently performed the best in terms of maximum absolute cross section asymmetry as a function of table size. The ranking of methods was different for each physical process. However, `NodeDensity` was more consistent in terms of which value needs to be set for `X_NNodes` in the fastNLO configuration file to achieve a targeted asymmetry of 10^{-3}, making it easier for the user to guess a good value.

6.1. Outlook

The momentum fraction x interpolation nodes can now be determined for all physical processes without a warmup run. For the energy scale interpolation nodes there are still cases where a warmup run is necessary because the quantity used as the energy scale is not observable. A concept (and following implementation) for efficiently treating these cases would be useful.

Furthermore the interpolation efficiency investigation in section 5.2 only covered a small number of physical processes and only at LO/NLO. A more comprehensive analysis that investigated more processes at up to NNLO could provide valuable insights for making the optimal choice between `NodesPerBin`, `NodesPerMagnitude`, and `NodeDensity`. However, apart from the method for x node spacing there are many more configuration options: the number of scale nodes, the interpolation kernels for x/scale nodes, and the transfer functions for the x/scale nodes. The large number of options greatly increases the amount of manual effort required for finding the optimal configuration. A method for automatically optimizing these options for a given physical process would therefore be very useful.

The automation of configuration optimization also allow for an extension of the configuration standard that would not be feasible to use without automation. For instance, as of right now fastNLO is intended to be used with a single value for `X_NNodes` (the number of nodes per bin is always proportional to this value). Since the number of nodes needed for a given precision differs between observable bins setting this value per observable bin would allow for smaller coefficient tables without reducing the worst-case precision. However, manually setting and optimizing values for up to hundreds of observable bins would not be feasible in terms of person time. If the `X_NNodes` value per bin could be determined automatically however this would no longer be an issue. Similarly, instead of fixed, predetermined transfer functions it would be feasible to add free parameters to the transfer functions that could be optimized per observable bin.

One approach with which the above optimization could be achieved would be an iterative calculation of small test coefficient tables. First a cost function that penalizes inefficient resource usage and precision below the user-requested value for each bin would need to be defined. Then for a set of potential configuration parameters a small test coefficient table would be calculated and its precision and size used to calculate a cost function value. Standard numerical optimization algorithms could then be used to determine the optimal configuration parameters. One issue with this approach is that the cost function may not converge smoothly towards its global minimum, causing the optimization to get stuck in a local minimum instead. Some parameters like the number of scale nodes or the `X_NNodes` value for `NodesPerBin` and `NodesPerMagnitude` can also only take on discrete values and would therefore be difficult to optimize. The calculation of coefficient tables for evaluating the cost function may also be too expensive; ideally it would be possible to determine good configuration values for NNLO from LO tables. Lastly the optimization would add an additional step to the workflow similar to the warmup run mentioned in this thesis (but the warmup can be merged with the parameter optimization). The added manual effort from an extra workflow step would probably be offset by the automation of determining optimal configuration values though.

Another approach that could potentially be used to determine optimal configuration value would be machine learning. Given enough data an artificial neural network of sufficient size would for example be guaranteed to be able to learn the mapping from the parameters of an analysis to the optimal fastNLO configuration parameters. The main challenge would be to create a good dataset from which such a mapping could be learned. However, at this point it should be noted that an approach based on machine learning is not mutually exclusive with the iterative approach described above. The iterative approach could be used to create a dataset for machine learning and machine learning could in turn be used to estimate good initial configuration parameters. It may also be possible to learn the mapping from optimal LO configuration parameters to optimal NNLO configuration parameters, making the iterative approach much cheaper.

Appendix

A. Implementation Details

This chapter describes the implementation of the new mode `NodeDensity`, how to use it, how it is integrated into existing workflows, and how the code had to be changed.

A. Configuration

In fastNLO run parameters can be set via a configuration file that is referred to as the "steering file". For example, steering files can specify the bin edges of the observable bins for which a fastNLO coefficient table should be calculated. These steering files are written in human-readable form and prior to the work of this thesis the specification already had a parameter `X_NNodeCounting` that specifies how the momentum fraction x nodes for PDF interpolation should be placed. When `X_NNodeCounting` is set to `NodesPerBin` a constant number of x nodes per observable bin is used, as determined by the parameter `X_NNodes`. When set to `NodesPerMagnitude` the number of nodes for an observable bin with minimal x value $x_{\min}$ is multiplied by a factor of $-\log_{10} x_{\min}$. With this thesis a new value for `X_NNodeCounting` was added: if set to `NodeDensity` then the new x node density implementation described in chapter 4 is used.

For the scale nodes new parameters `Mu1_NNodeCounting` and `Mu2_NNodeCounting` that control how the minimum and maximum scale values $\mu_{\min}, \mu_{\max}$ are determined for the production run scale nodes were added. If set to `NodesPerBin` then $\mu_{\min}, \mu_{\max}$ are determined from a warmup run (default value, same behavior as prior to this thesis). If set to `NodesPerDim1`, `NodesPerDim2`, or `NodesPerDim3` then the bin edges of the first/second/third observable dimension are used as $\mu_{\min}, \mu_{\max}$.

Prior to this thesis, when running a fastNLO binary, the program would first check whether a fastNLO warmup file containing the min./max. values for the coefficient table already exists. If the file exists, a production run is started. Otherwise a warmup run to produce the file is started. In the current version, if the user specifies both an x node density and that $\mu_{\min}, \mu_{\max}$ should be taken from the observable binning then no warmup run is needed and fastNLO immediately starts with a production run regardless of the warmup file's existence. Figure A.1 shows an activity diagram for code used to determine whether a warmup run is needed.

B. Workflow Integration

In principle fastNLO can be used directly by end users to calculate coefficient tables. In practice however, at least one layer of indirection is needed. The first reason for this indirection is that the large number of CPU hours needed for NNLO calculations necessitates the use of massive parallelism via compute clusters. To achieve good hardware utilization these compute clusters typically provide computing resources to a large number of users and use specialized job scheduling software to assign computing resources to individual jobs. So typically a user will need to use code outside fastNLO to specify the logic of the

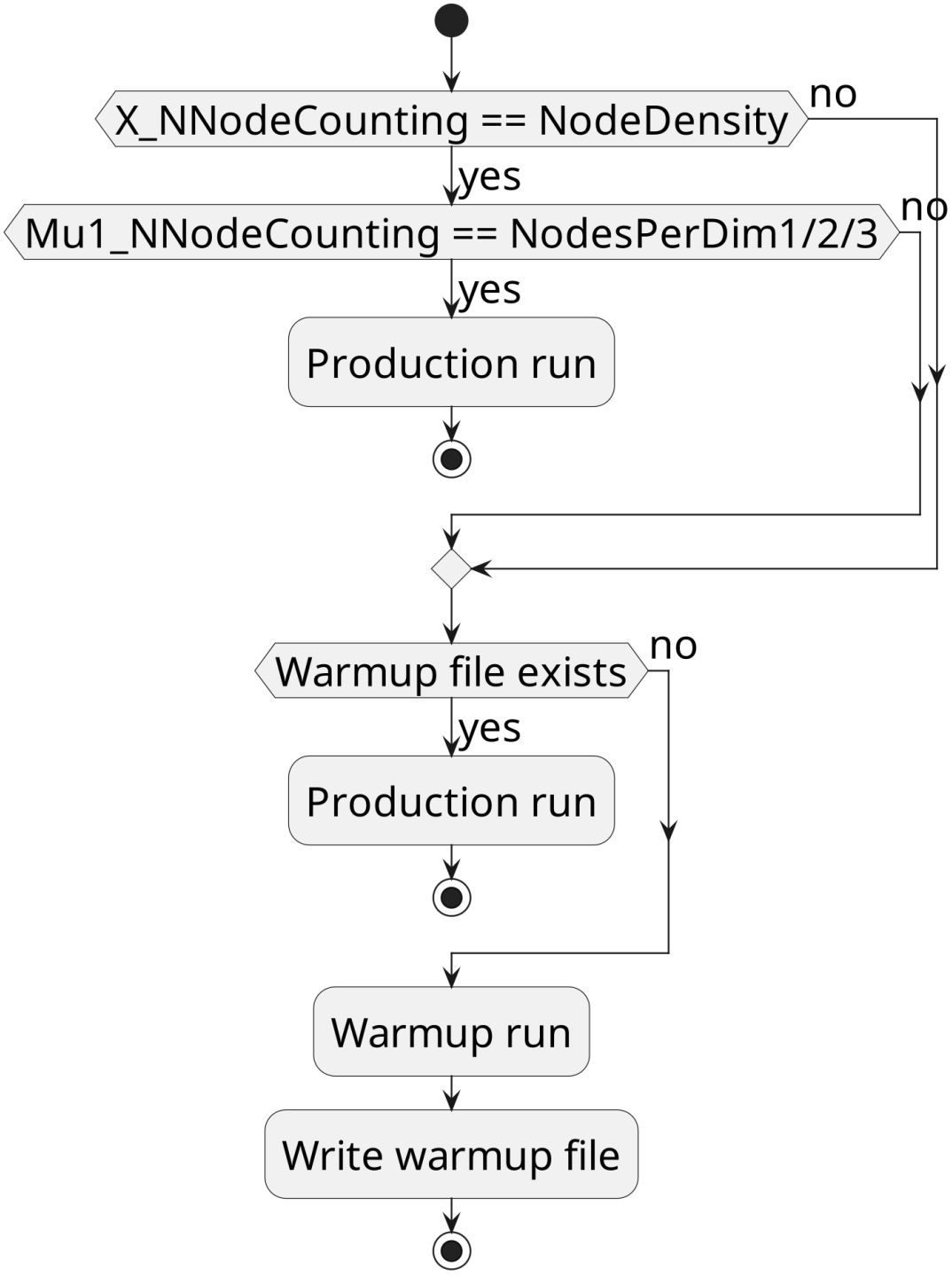

Figure A.1.: Activity diagram of the fastNLO code for determining whether a warmup run is needed.

overall workflow for creating coefficient tables. This external code then serves as an entry point that the job scheduling software can use to indirectly launch fastNLO once it decides that the corresponding job should receive computing resources. As an additional benefit, explicitly formulating the logic of a workflow as code also makes the workflow easier to reproduce since it reduces the amount of (potentially undocumented) manual steps.

The new code added by this thesis was tested with both Slurm[5] (Simple Linux Universal Resource Manager) and HTCondor[3], the latter of which with one more layer of indirection via Luigi[4] and LAW (Luigi Analysis Workflow)[24]. The SLURM workflow was comparatively light-weight (essentially just a small Bash script that launches the SLURM jobs) and only required manually initiating a production run without a previous manual warmup run. The LAW workflow was more elaborate, particularly in regards to its automatic handling of dependent tasks: when a task that depends on another task is run by the user, it first checks whether all tasks listed as dependencies have been successfully run. If not, then these dependencies are run first. Though there are more steps in-between, with the original code the `FastProd` task (fastNLO production run) depended unconditionally on the completion of the `FastWarm` task (fastNLO warmup run). However, due to the code changes of this thesis fastNLO warmup runs are now only necessary if the steering files specify the pre-existing modes for `X_NNodeCounting`, `Mu1_NNodeCounting`, or `Mu2_NNodeCounting`. Therefore, as part of this thesis the logic for determining whether a warmup is needed (see figure A.1) was reimplemented as part of the LAW workflow: the LAW code now reads in the fastNLO steering file(s) at runtime and dynamically sets the dependencies of the `FastProd` task to reflect the behavior of the fastNLO code. This way the user only has to edit the steering file(s) and run the `FastProd` task. The workflow will then automatically schedule a warmup run if needed.

One additional thing to note in this context is that the aforementioned LAW workflow makes use of NNLOJET as the event generator. As such there is additional program logic in the NNLO Bridge [1] code that also needs to be accounted for. More specifically, the effective configuration used for fastNLO can be a combination of multiple steering files. Firstly, there are multiple possible paths for global steering files. These paths have different priorities and the first existent file will be read and used to set default values for all NNLOJET histograms. Afterwards the code checks for the existence of a histogram-specific steering file for each NNLOJET histogram. If such files exist they can potentially override the options set in the global steering file. If at least one NNLOJET histogram is configured in such a way that a fastNLO warmup is needed then the production run as a whole requires a fastNLO warmup.

C. Table Filling

fastNLO has methods that are called by user code to increment (to "fill") the coefficient table with events from an event generator. To dynamically extend the x nodes upon encountering a new minimal x value it was therefore necessary to modify these methods. Unfortunately the original code had five different methods that modified the coefficient table. Figure C.2 shows an activity diagram for the original code. In it, when the method `FillContribution` of the class `fastNLOCreate` is called it checks whether the fastNLO coefficient table is calculated as fixed or flexible scale and whether the physical scenario describes deep inelastic scattering or hadron-hadron collisions. In the method `FillAllSubprocesses`, if a two-dimensional vector of hadron-hadron-collision events is used to fill a fixed-scale table, then the coefficients are incremented in this method via a dedicated implementation that was added for better performance. Otherwise the filling is delegated to one of the following methods inside a

[1] As a reminder, NNLO Bridge is the adapter for fastNLO and APPLGrid shipped with NNLOJET, see section 2.2

Figure C.2.: Activity diagram of the original fastNLO table filling code. Red indicates code that modifies the coefficient table.

loop: `FillContributionFixDIS`, `FillContributionFixHHC`, `FillContributionFlexDIS`, or `FillContributionFlexHHC`. As a consequence an implementation of x node density would have required code modifications in five different places. The code for each fill method was mostly the same. Therefore, the fill methods were refactored and deduplicated as part of this thesis prior to the x node density implementation: the filling of events is now always handled in the method `FillContribution`. Figure C.3 shows an activity diagram for the refactored table filling code. There is no longer a special treatment of fixed-scale two-dimensional vectors. The differentiation between fixed and flexible scale is handled via polymorphism: the pre-existing classes `fastNLOCoeffAddFix` and `fastNLOCoeffAddFlex` (both subclasses of `fastNLOCoeffAddBase`) were extended with a method `Fill` that handles the differentiation. The method `Fill` is then simply called from the method `FillContribution`. This is more consistent with object-oriented design philosophy since the `fastNLOCreate` class can now always handle objects of type `fastNLOCoeffAddBase` in the same way via an interface. Previously the class `fastNLOCreate` would directly modify attributes of `fastNLOCoeffAddBase`. The new implementation ensures consistent behavior regardless of the implementation details of the subclasses of `fastNLOCoeffAddBase`. As a side effect the code for filling flexible-scale tables has also become faster (see section 5.1).

The program logic of table filling is shown in more detail in figure C.4 via a sequence diagram. Figure C.5 shows the corresponding class diagram (reduced to the relevant parts). The user code is intended to work with a single `fastNLOCreate` object (a subclass of `fastNLOTable`) which encapsulates all relevant information for a fastNLO coefficient table. Each `fastNLOCreate` object has four `fastNLOInterpolBase` objects per observable bin to represent the nodes: two for the momentum fraction (one for a proton PDF and a second one for a potential antiproton PDF) and one each for up to two possible scales. The property `fGrid`/`fHGrid` represents the locations of the nodes in non-transformed/transformed space. The two-dimensional vectors `XNode1` and `XNode2` contain a duplicate of the information stored in `fGrid`. The method `CalcNodeValues` determines which nodes are affected by an event and how the event weight should be distributed across those nodes. It is the main point where the subclasses of `fastNLOInterpolBase` differ: different subclasses will distribute the event weight differently. Each `fastNLOCreate` object has at least one `fastNLOCoeffAddBase` object to represent the actual coefficient table values. `fastNLOCoeffAddFix` has only a single five-dimensional vector to store the grid values. The vector's dimensions are in order:

1. the observable bin index,
2. the scale variation index,
3. the scale node index,
4. the momentum fraction node index,
5. and the subprocess index.

By contrast `fastNLOCoeffAddFlex` has six different five-dimensional vectors. One to store the factors independent of scale values as well as five to store the scale-dependent factors. Specifically the scale-dependent coefficient tables are factors of: $\log \mu_r$, $\log \mu_f$, $\log^2 \mu_r$, $\log^2 \mu_f$, and $\log \mu_r \log \mu_f$. A scale variation index is therefore not needed. Instead the vectors' dimensions are in order:

1. the observable bin index,
2. the momentum fraction node index,
3. the first scale node index,
4. the second scale node index,

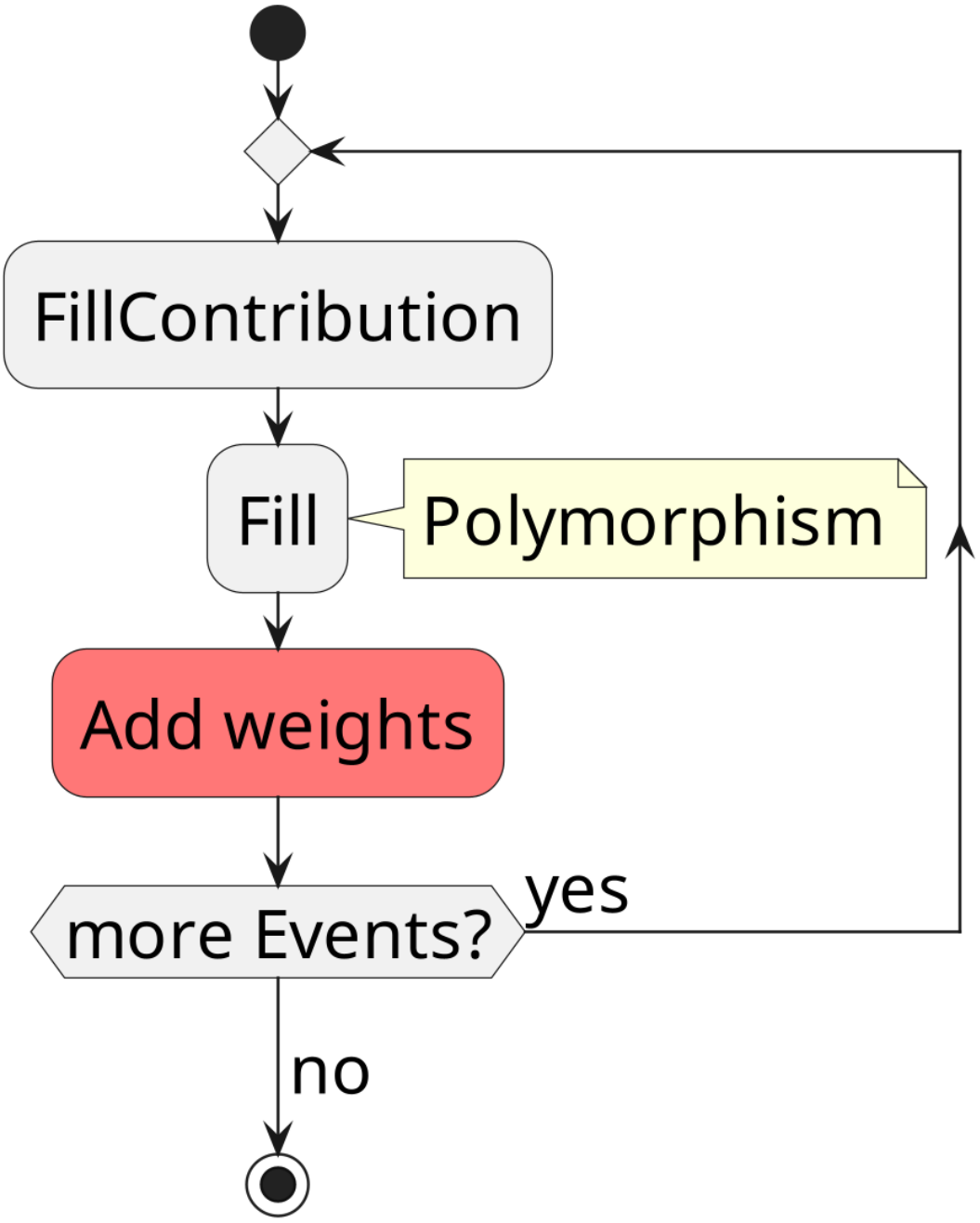

Figure C.3.: Activity diagram of the refactored fastNLO table filling code. Red indicates code that modifies the coefficient table.

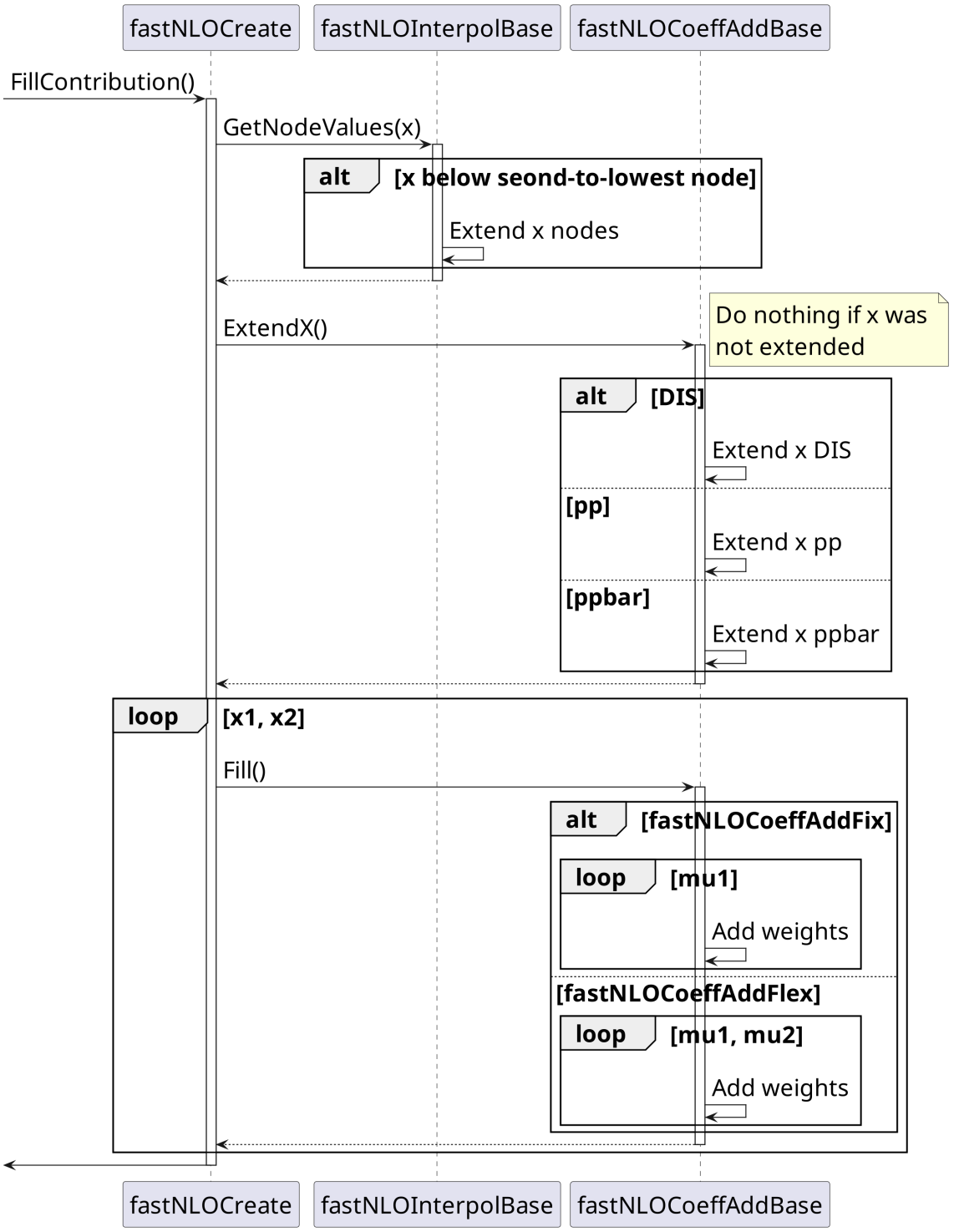

Figure C.4.: Sequence diagram of the refactored fastNLO table filling code.

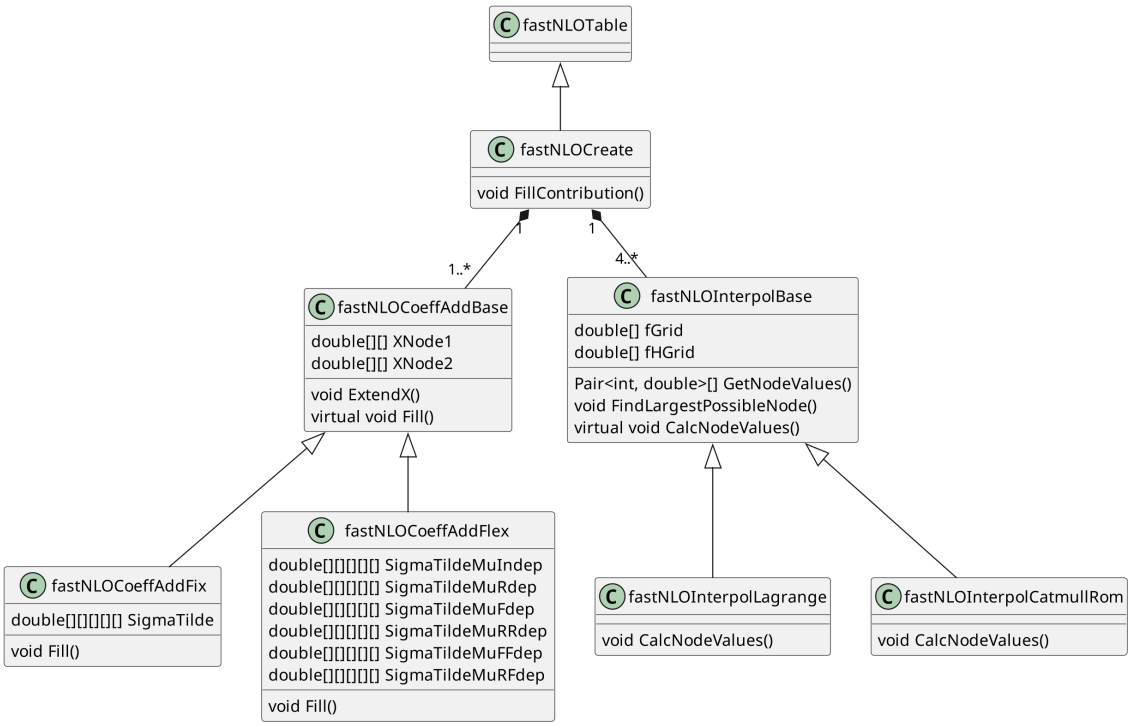

Figure C.5.: Class diagram of the fastNLOCreate class showing only the parts relevant for the table filling code. Vectors are shown as arrays for simplicity.

5. and the subprocess index.

Note that the order of the indices for the momentum fraction and scale nodes has changed. The scale variation index is replaced with a second scale node index.

To explain the way the x node density implementation works, first the implementation for deep inelastic scattering (i.e. for a single proton) with a fixed number of x nodes should be considered. In the first step, the code calls `GetNodeValues` to determine which x nodes will be affected by the events to be filled. For the Lagrange and Catmull-Rom x interpolation kernels this will usually be the two nodes directly adjacent to either side of the x value of an event for a total of four nodes. For the linear x interpolation kernels only the one directly adjacent node to either side is affected for a total of two nodes. The information regarding how the event weight should be distributed between nodes is represented as sparse vectors of pairs: the first value of each pair is an integer representing the index of an x node while the second value is a floating point number representing the weight assigned to the node (from the interpolation kernel only). In the case of proton-proton or proton-antiproton collisions two partons are involved. Therefore, each event has two separate x values and requires two separate vectors of indices and weights for the partons from both particles. In practical terms this means that `GetNodeValues` is called two times with two different x values. The calculation is otherwise identical (but protons and antiprotons may use different x nodes and therefore different `fastNLOInterpolBase` objects).

In addition to the x dimensions at least one scale dimension needs to be interpolated (flexible scale can have two scale dimensions, the corresponding function calls are not shown in the sequence diagram). The weights are again obtained by a call to `GetNodeValues` on a different object. The final interpolation weights for a given coefficient table value are then obtained by simply multiplying the individual interpolation weights from each dimension. All possible combinations are calculated and added to the coefficient table.

Once the weights from interpolation have been determined the final weight to be added

to the coefficient table can be obtained by multiplying the interpolation weight with the weight of the event. The x nodes can be treated in the same way for fixed and flexible scale tables and as such the same code is used. The scale nodes need to be treated differently between fixed and flexible scale tables and the corresponding code is split across the `fastNLOCoeffAddFix` and `fastNLOCoeffAddFlex` classes.

The first part of the code specific to x is an iteration over the x nodes affected by the event. Afterwards the effective x index needs to be calculated from the nominal x indices of the first proton and a potential antiproton or second proton. While the nominal x indices can be two-dimensional the effective x index (which corresponds to the x index of the coefficient table) is always flattened to one dimension. In the case of deep inelastic scattering the nominal and effective x indices are identical. In the case of proton-antiproton collisions the combinations of the nominal x indices are simply iterated over and the effective x index corresponds to an element of a flattened matrix (with the nominal x indices corresponding to the column and row indices of the matrix). In the case of proton-proton collisions the same scheme as with proton-antiproton collisions could be used and still produce correct results. However, it is possible to exploit the symmetry in the PDFs of the ingoing particles to reduce the size of the coefficient table: nominal x index combinations in which the second index is smaller than the first index are flipped so that the first index is smaller. This way only a triangular matrix is needed to store the coefficient table which reduces its size by roughly 50%. Once the effective x index has been determined the coefficient table can be modified by iterating over the scale indices.

The `NodeDensity` implementation differs from the previously described code as follows: firstly the `fGrid` and `fHGrid` properties of the x-associated `fastNLOInterpolBase` object(s) and the x dimension of the coefficient table start out with a small size of only four nodes (the minimum for Lagrange/Catmull-Rom kernels). The first index corresponds to the lowest x value while the last index corresponds to $x = 1$. Then, when `GetNodeValues` is called to determine the x interpolation weights, one of the first things it does is to in turn call `FindLargestPossibleNode` (also for methods other than `NodeDensity`). This method then iterates over the nodes and returns the index of the largest node that is still below the x value of the event. The aforementioned method also handles the case when the x value is smaller than the lowest node. For the pre-existing implementations this case is handled by rounding up the x value to the lowest node, potentially introducing some bias to the coefficient table. When using `NodeDensity` that case is instead handled by extending the node vectors to accommodate the new minimum x value. The nodes are equidistantly spaced in the transformed space (`fHGrid`). This makes adding more nodes below the ones that already exist simple. The new non-transformed values for `fGrid` can then be calculated by simply applying the inverse of the transfer function. After expanding the node vectors the x node weights can be calculated as normal.

However, it is important to note that because the node vector is sorted in ascending order the new nodes have to be inserted *at the beginning* of the vector. As a consequence the indices for the pre-existing nodes change as new nodes are inserted. These index changes must be carefully synchronized between objects to ensure correct results. This is done by calling `fastNLOCoeffAddBase::ExtendX`. This method copies the new entries from `fGrid` to `XNode1` and `XNode2`. Afterwards the coefficient table represented by one or more `SigmaTilde` properties is extended. This extension needs to be handled differently for deep inelastic scattering, proton-proton collisions, and proton-antiproton collisions. The case of deep inelastic scattering is the simplest since the x coefficient table dimension is a simple one-dimensional vector. Therefore it is sufficient to simply insert zeros at the front of the node until the x dimension of the coefficient table is the same size as the node vector. For

example, inserting a single zero at the beginning of a vector with three elements:

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & x_1 & x_2 & x_3 \end{pmatrix}, \tag{C.1}$$

where the values x_i are simply the values that were stored in the coefficient table prior to the expansion. Because the coefficient table as a whole is five-dimensional these are not simple scalars but they can be treated as such without affecting the logic of the expansion. Also, while in this example only a single zero is inserted at the front this procedure can simply be repeated until the desired size is reached. For the more complex cases this greatly simplifies the implementation. For instance, when simulating proton-proton collisions the x dimension of the coefficient table is stored in half-matrix form. The coefficient table extension from 3 to 4 nodes then takes on this form:

$$\begin{pmatrix} x_1 & & \\ x_2 & x_3 & \\ x_4 & x_5 & x_6 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & & & \\ 0 & x_1 & & \\ 0 & x_2 & x_3 & \\ 0 & x_4 & x_5 & x_6 \end{pmatrix}. \tag{C.2}$$

In this case the first zero is inserted at index 0 at the front of the vector. The second zero is then inserted at index 1, immediately behind the first zero. The third index is then inserted at index 3, the fourth zero is inserted at index 6, the fifth zero is inserted at index 10, and so on. The next index at which a zero should be inserted is the previous index plus the number of already inserted zeros. For proton-antiproton collisions the insertion is relatively simple:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & x_1 & x_2 & x_3 \\ 0 & x_4 & x_5 & x_6 \\ 0 & x_7 & x_8 & x_9 \end{pmatrix}. \tag{C.3}$$

Assuming the x dimension is a matrix of size N the first step is to insert N zeros at the front. Then simply a single zero at the beginning of each row needs to be inserted. After both the node vectors and the coefficient table have been extended the event can be filled as normal.

D. Table Merging

Due to the large amount of CPU hours necessary to create fastNLO coefficient tables at NNLO it is necessary to parallelize the computation. This is done by running multiple fastNLO processes in parallel using different starting seeds. Each process creates its own coefficient table. Afterwards the results from individual processes are merged to create a single final coefficient table.

When using `NodesPerBin` or `NodesPerMagnitude` all processes use the same fastNLO warmup file as input. The lower bound of the momentum fraction x as well as the number of nodes for each individual bin is therefore the same. Tables can be merged by simply iterating the individual values and summing up those with the same indices. However, when using `NodeDensity` this is no longer the case. Depending on the starting seed, the minimum x values encountered by a process during the production run will vary. The size of the coefficient tables between processes can therefore also vary. To complicate matters further, the indices of the coefficients that need to be merged are *not* the same: because the coefficient table is extended by inserting zeros at the front of a vector the indices of the pre-existing coefficients change with each extension. This problem is very similar to the one encountered during table filling and luckily the same code can be reused for table merging. By simply inserting zeros into the smaller coefficient table until it is the size of the larger one the indices can be made to align. Afterwards the tables can be merged in

the same way as the other tables. Inserting zeros into the smaller table is slightly inefficient in terms of memory and runtime (compared to computing which indices should align for merging) but maintainability was judged to be more important.

E. Testing

In order to ensure that the `NodeDensity` implementation is consistent with `NodesPerBin` automated tests have been added to the project. The tests first generate two LO fixed/flexible scale tables each for both `NodesPerBin` and `NodeDensity` using the same seeds for both methods. The warmup file for `NodesPerBin` has been manually edited so that the nodes are expected to align. First the contents of the calculated individual coefficient tables are compared. Again, care must be taken to compare the correct values since the indices only align if the minimum x node is the same. Next the tables using the same method for x node spacing are merged and the merged tables are compared again. If all compared tables are the same (within numerical precision) then the test is passed, otherwise it fails.

B. NNLOJET Runtime Percentages

With the profiling data already collected for section 5.1, further analysis was done to aid in the development of the NNLOJET project. Firstly the performance regression of rev6591 relative to rev5918 was investigated. To this end the RRa profiling data was compared and functions with a disproportionate increase in runtime were identified (see table .1). RRa was chosen because it is the contribution with by far the highest runtime per event. The method `__evalobs_mod_clearcache_obs` in particular stands out because while its purpose seems to be the clearing of caches it takes up 19.8% of the total runtime.

In addition, differences between leading color and full color (with or without multichanneling) were investigated. Table .2 shows how the percentage of runtime taken up by the previously investigated functions varies depending on LC/FC and multichanneling. No significant difference in terms of runtime percentages was observed. LC/FC/multichanneling was also investigated for RV (see table .3). There are some differences in runtime percentages but there is nothing that immediately stands out (without inspecting the source code).

Table .1.: Comparison of RRa profiling results for select NNLOJET methods.

Method/program	Incl. rev5918 [s]	Self rev5918 [s]	Incl. rev6591 [s]	Self rev6591 [s]
NNLOJET without fastNLO	74.350		155.033	
___observables_mod_MOD_evalall_obs	27.154	24.121	90.421	1.576
___mapping_mod_MOD_set_map	0.000	0.000	39.996	5.318
___mapping_mod_MOD_apply_map	0.000	0.000	33.118	5.366
___evalobs_mod_clearcache_obs	0.000	0.000	30.682	30.682
___evalobs_mod_MOD_applycuts_jet	0.586	0.195	28.739	11.512
___kindata_mod_MOD_fills_kin	0.391	0.069	20.762	2.930
___kindata_mod_MOD_spinab_kin	0.322	0.287	17.833	15.301
log (libm.so.6)	0.000	0.000	8.375	8.375
___evalobs_mod_MOD_initrecomb_jet:	0.195	0.057	7.961	2.627
atan2 (libm.so.6)	0.103	0.103	6.496	6.496

Table .2.: Comparison of inclusive rev 6591 RRa profiling results for select NNLOJET methods.

Method/program	LC	FC	FC MC
___mapping__mod__MOD__set__map	25.12%	23.98%	25.95%
___mapping__mod__MOD__apply__map	20.80%	19.88%	21.49%
___evalobs__mod__clearcache__obs	19.27%	19.29%	19.49%
___evalobs__mod__MOD__applycuts__jet	18.05%	17.09%	18.69%
___kindata__mod__MOD__fills__kin	13.04%	12.40%	13.45%
___kindata__mod__MOD__spinab__kin	11.20%	10.65%	11.56%
log (libm.so.6)	5.26%	4.99%	5.35%
___evalobs__mod__MOD__initrecomb__jet:	5.00%	4.74%	5.16%
atan2 (libm.so.6)	4.08%	3.77%	4.16%

Table .3.: Comparison of inclusive rev 6591 RV profiling results for select NNLOJET methods.

Method/program	LC	FC	FC MC
hpl2__	37.69%	65.46%	42.26%
zzcf__	28.70%	61.12%	32.55%
hpl2else__	34.58%	60.51%	38.59%
cli2__	26.42%	47.64%	29.48%
__gfortran__pow__c8__i4	17.26%	30.90%	19.38%
cli2__'2	12.64%	24.43%	14.33%
___evalobs__mod__clearcache__obs	10.61%	5.49%	9.33%
___mapping__mod__MOD__set__map	6.13%	2.60%	5.20%
___evalobs__mod__MOD__applycuts__jet	5.11%	2.86%	4.55%
log (libm.so.6)	4.10%	2.58%	4.10%
___mapping__mod__MOD__apply__map	5.03%	2.13%	4.26%
___kindata__mod__MOD__fills__kin	3.07%	1.30%	2.61%
___kindata__mod__MOD__spinab__kin	2.63%	1.11%	2.24%
___evalobs__mod__MOD__initrecomb__jet:	1.34%	0.73%	1.19%
atan2 (libm.so.6)	1.26%	0.60%	1.09%

Bibliography

[1] *Catastrophic Cancellation*. https://en.wikipedia.org/wiki/Catastrophic_cancellation. Accessed: 2023-12-13.

[2] *Cubic Hermite Spline*. https://en.wikipedia.org/wiki/Cubic_Hermite_spline. Accessed: 2023-12-13.

[3] *HTCondor*. https://htcondor.org/. Accessed: 2023-12-08.

[4] *Luigi*. https://github.com/spotify/luigi. Accessed: 2023-12-08.

[5] *Slurm (Simple Linux Universal Resource Manager)*. https://slurm.schedmd.com/overview.html. Accessed: 2023-12-08.

[6] *Valgrind*. https://valgrind.org/. Accessed: 2023-12-08.

[7] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008, Seiten 1–70, 2008.

[8] Ball, Richard D, Valerio Bertone, Stefano Carrazza, Christopher S Deans, Luigi Del Debbio, Stefano Forte, Alberto Guffanti, Nathan P Hartland, José I Latorre, Juan Rojo *et al.*: *NNPDF*. arXiv preprint arXiv:1410.8849, 2014.

[9] Bothmann, Enrico, Gurpreet Singh Chahal, Stefan Höche, Johannes Krause, Frank Krauss, Silvan Kuttimalai, Sebastian Liebschner, Davide Napoletano, Marek Schönherr, Holger Schulz, Steffen Schumann und Frank Siegert: *Event generation with Sherpa 2.2*. SciPost Physics, 7(3), September 2019, ISSN 2542-4653. http://dx.doi.org/10.21468/SciPostPhys.7.3.034.

[10] Britzger, Daniel, Klaus Rabbertz, Fred Stober und Markus Wobisch: *New features in version 2 of the fastNLO project*. In: *20th International Workshop on Deep-Inelastic Scattering and Related Subjects*, Seiten 217–221, 2012.

[11] Carli, Tancredi, Dan Clements, Amanda Cooper-Sarkar, Claire Gwenlan, Gavin P. Salam, Frank Siegert, Pavel Starovoitov und Mark Sutton: *A posteriori inclusion of parton density functions in NLO QCD final-state calculations at hadron colliders: the APPLGRID project*. The European Physical Journal C, 66(3):503–524, Apr 2010, ISSN 1434-6052. https://doi.org/10.1140/epjc/s10052-010-1255-0.

[12] Carrazza, S., E. R. Nocera, C. Schwan und M. Zaro: *PineAPPL: combining EW and QCD corrections for fast evaluation of LHC processes*. JHEP, 12:108, 2020.

[13] Catmull, Edwin und Raphael Rom: *A CLASS OF LOCAL INTERPOLATING SPLINES*. In: BARNHILL, ROBERT E. und RICHARD F. RIESENFELD (Herausgeber): *Computer Aided Geometric Design*, Seiten 317–326. Academic Press, 1974, ISBN 978-0-12-079050-0. https://www.sciencedirect.com/science/article/pii/B9780120790500500205.

[14] Collins, John C., Davison E. Soper und George Sterman: *Factorization of Hard Processes in QCD*, 2004.

[15] Gehrmann, T., X. Chen, J. Cruz-Martinez, J. R. Currie, E. W. N. Glover, T. A. Morgan, J. Niehues, D. M. Walker, R. Gauld, A. Gehrmann De Ridder, A. Huss und Joao Pires: *Jet cross sections and transverse momentum distributions with NNLOJET*, 2018.

[16] Gross, David J. und Frank Wilczek: *Ultraviolet Behavior of Non-Abelian Gauge Theories.* Physical Review Letters, 30(26):1343–1346, Juni 1973.

[17] Horzela, Maximilian Maria: *Measurement of Triple-Differential Z+Jet Cross Sections with the CMS Detector at 13 TeV and Modelling of Large-Scale Distributed Computing Systems.* Dissertation, Karlsruher Institut für Technologie (KIT), 2023.

[18] Kloek, T. und H. K. van Dijk: *Bayesian Estimates of Equation System Parameters: An Application of Integration by Monte Carlo.* Econometrica, 46(1):1–19, 1978, ISSN 00129682, 14680262. `http://www.jstor.org/stable/1913641`, besucht: 2023-12-13.

[19] Kluge, T., K. Rabbertz und M. Wobisch: *FastNLO: Fast pQCD calculations for PDF fits.* In: *14th International Workshop on Deep Inelastic Scattering*, Seiten 483–486, September 2006.

[20] Lagrange, Joseph Louis: *Leçon Cinquième. Sur l'usage des courbes dans la solution des problèmes.* Leçons Elémentaires sur les Mathématiques, 1795.

[21] Nagy, Zoltan: *NLOJet++.* `https://www.desy.de/~znagy/Site/NLOJet++.html`. Accessed: 2023-12-07.

[22] Politzer, H. David: *Reliable Perturbative Results for Strong Interactions?* Physical Review Letters, 30(26):1346–1349, Juni 1973.

[23] Pumplin, Jonathan, Daniel Robert Stump, Joey Huston, Hung Liang Lai, Pavel Nadolsky und Wu Ki Tung: *New Generation of Parton Distributions with Uncertainties from Global QCD Analysis.* Journal of High Energy Physics, 2002(07):012–012, Juli 2002, ISSN 1029-8479. `http://dx.doi.org/10.1088/1126-6708/2002/07/012`.

[24] Rieger, Marcel, Martin Erdmann, Benjamin Fischer und Robert Fischer: *Design and Execution of make-like, distributed Analyses based on Spotify's Pipelining Package Luigi*, 2017.

[25] Rodgers, David P.: *Improvements in Multiprocessor System Design.* In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, Seite 225–231, Washington, DC, USA, 1985. IEEE Computer Society Press, ISBN 0818606347.

[26] Runge, Carl: *Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten.* Zeitschrift für Mathematik und Physik, 1901.

[27] Weierstrass, Karl: *Über die analytische Darstellbarkeit sogenannter willkürlicher Functionen einer reellen Veränderlichen.* Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin, 1885.