Reparametrisation of the Lund symmetric fragmentation function

Sophie Li

Supervisor: Dr. Peter Skands

PHS3350: Physics and astronomy research project 1

Monash University

19 February 2018

Abstract

Jet fragmentation is a phenomenon seen at particle colliders. The process can be modelled by Monte Carlo models such as the string model, upon which the Lund symmetric fragmentation function (henceforth referred to as the Lund function) is based. The high level of correlation between two of the Lund function's parameters results in difficulty assigning them independent uncertainties. Therefore, in this report, we detail a reparametrisation of the function, which was done using numerical methods. Furthermore, we show that the new parameters exhibit a weaker degree of correlation on an important test observable: the average number of charged hadronic fragments produced in decays of the Z boson.

1 Introduction

Within its accelerator complex, CERN houses the well-known LHC which is notable for its detection of the Higgs boson in 2012 [1, 2]. As the boson decays quickly, it was detected by its decay products, two photons. While this mode of decay is particularly easy to detect, it does not occur most of the time. In order to study the properties of the Higgs boson for the more common decay modes, we require knowledge of a process called jet fragmentation.

Jet fragmentation is a phenomenon of quantum chromodynamics (QCD) at strong coupling. As such, there are no known perturbative models that describe it, but physicists have developed phenomenological models such as the string model [3]. This report focuses on the related Lund function which parametrises the probability of production of a hadron that takes a given fraction, z, of the remaining energy-momentum from a high-energy collision [4]. Even generators such as PYTHIA use this model [5]. This report details the function's reparametrisation, which is a helpful step in making data fitting and uncertainty assignments a meaningful process.

In order to explain the reasoning behind the reparametrisation, we will briefly cover the elementary particles and fundamental forces. We will then explore the colour charge, including its field and potential, as well as the strong coupling constant, which will lead us to the concept of confinement. This will bring us to the fragmentation process, which we will see can be described by the string model. From there, we will discuss the Lund function and its reparametrisation.

2 Physics theory

2.1 The Standard Model

The Standard Model (Fig. 1) is the currently-favoured theory that describes the interactions of the elementary particles as governed by the fundamental forces. The elementary particles are comprised of fermions and bosons. The fundamental forces are the electromagnetic force, the weak force, the strong force, and the gravitational force, although the latter has not been incorporated into the Standard Model as its effects are too weak to be measured on a subatomic scale.

The fermions are the particles that make up matter that we see every day. They are characterised by a half-integer spin and obey the Pauli exclusion principle, which states that no two fermions may be described by the same set of quantum numbers. The Standard Model recognises twelve types of fermions which are divided into the leptons and the quarks. The fermions come in three generations, with each generation containing one up quark, one down quark, one charged lepton, and one neutrino. Particles in later generations have a greater mass than those in previous generations, and tend to decay into lower generation particles. Conversely, the bosons are the mediators of interactions between the fermions and participating bosons. They have an integer spin and do not obey the Pauli exclusion principle. There are five bosons, each of which are involved in certain interactions. The gauge bosons are exchanged in interactions between fermions while the Higgs boson (H) is the mechanism that gives mass to all elementary particles which have a mass.

The quarks and gauge bosons are of particular interest in this report. There are six different flavours of quarks which are distinguished by their masses and electric charges. The first generation consists of the up (u) and down (d) quarks; the second consists of the charm (c) and strange (s) quarks; and the third consists of the top (t) and bottom (b) quarks. The bosons appear in processes between particles. The photon (γ) is the mediator of electromagnetic interactions, which only occur between charged particles. The gluons (g) are the bosons that are exchanged in strong processes. Only coloured particles can interact via this force. The W and Z bosons mediate weak interactions. For a more in-depth discussion of the elementary particles, see [6].

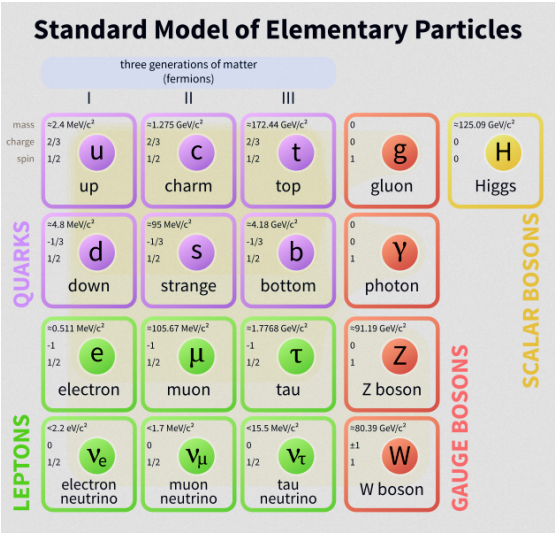


Figure 1: The Standard Model [7].

2.2 The colour charge

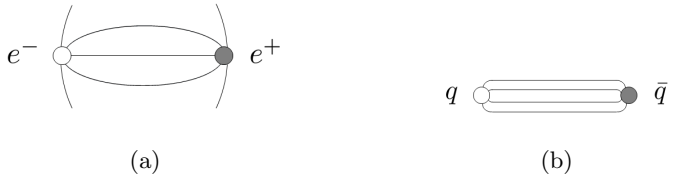
A familiar property of the quarks is their electric charge. The up, charm, and top quarks, collectively referred to as the up quarks, each have an electric charge of $\frac{2}{3}e$. Similarly, the down, strange, and bottom quarks, collectively called the down quarks, each have charge $-\frac{1}{3}e$. Since all quarks have a net charge, they can interact through the electromagnetic force via the exchange of photons.

Quarks and gluons also have an intrinsic colour property. All quarks carry a single colour out of red (R), green (G), or blue (B), with antiparticles carrying the respective anticolours. This is a convenient basis to use as colour theory tells us how quarks can be combined to give a net white colour. It turns out that all particles with a wavelength greater than 1 fm must be white [8].

The mediators of the strong force, the gluons, also have a net colour. Because of this, they are affected by the strong force. Strong interactions must be treated differently to electromagnetic interactions since they involve gluon self-coupling, while photons are invisible to the electromagnetic force as they are electrically neutral.

2.3 The strong coupling and total confinement

Continuing on with our analogy with the electromagnetic force, the force lines of the colour field take on a characteristic shape. At short distances of less than 1 fm, the field lines of both forces are very similar. However at greater distances, the behaviour of the field lines are noticeably different. While the electric field lines tend to spread out (Fig. 2a), the colour field lines are compelled to form a tube-like region (Fig. 2b). This arises because of the properties of photons and gluons. Photons are electrically neutral particles, so cannot interact through the electromagnetic force. As a consequence of the lack of self-attraction, the electric field lines spread out. However, gluons are bicoloured particles, so can interact via the strong force. The colour field lines are therefore attracted to each other and do not disperse [8].



(a) (b)

Figure 2: Diagrams of (a) the electric field, and (b) the colour field.

The value of the coupling constant depends on the type of force involved. Subsequently the constants are written with the appropriate subscript specifying the type of interaction. For electromagnetic interactions, the amplitude for a process such as the emission of a photon is proportional to $\sqrt{\alpha_{EM}}$, where $\alpha_{EM} = \frac{e^2}{4\pi\epsilon_0\hbar c} \approx \frac{1}{137}$ [9]. The strong and weak coupling constants, α_S and α_W, cannot be approximated to such an accurate number independent of a resolution scale.

The coupling constant is actually a function of the distance measured from the charged particle. In the case of electromagnetic interactions, it is known from electrostatics how this affects α_{EM}. As opposite charges attract, a negative charge is preferentially surrounded by positive charges. Thus when it is probed with a test charge, the magnitude of the original charge is reduced, or screened. At shorter distances, the screening effect is less noticeable so the effective charge is higher. Since α_{EM} depends on e^2, this corresponds to a higher α_{EM}. Further away from the original charge, the effective charge approaches a constant value. It is this asymptotic value that is quoted as α_{EM}.

This scenario is not seen for strong interactions; in fact, the above behaviour of charges is reversed. Colour charges are preferentially surrounded by charges of the same colour. In other words an antiscreening effect is observed as charges of the same colour cluster together. Thus when a quark or gluon is probed with a test charge, the effective colour charge increases at greater distances. This implies that α_S increases as the test charge moves further away from the original charge. This action is not energetically favourable, and in fact no finite amount of energy is sufficient to liberate the quarks or gluons. Because of this, quarks and gluons are always confined to hadrons in such a way that the net colour is white. This is referred to as infrared slavery [9].

2.4 Jet fragmentation

The nature of the colour field gives rise to a phenomenon called jet fragmentation. The lowest order Feynman diagram for quark-antiquark pair production in electron-positron pair annihilation is shown in Fig. 3. Ignoring the initial state, which has no coloured partons, we begin with a quark-antiquark pair and give the system some energy. This energy causes the quark-antiquark pair to move apart in opposite directions. As the particles get further apart, the colour field between them gets stronger. This increase in attraction between the quarks gradually converts their kinetic energy into potential energy that is stored in the field. At some point this potential energy is enough to form new quarks and gluons. These particles have some kinetic energy so move in a manner similar to the original particle pair. They also have an associated colour field from which new particles can form when the potential energy is sufficient. This production of new particles from the colour field of existing particles continues until the energy given to the original quark-antiquark pair resides solely in the hadrons produced [4].

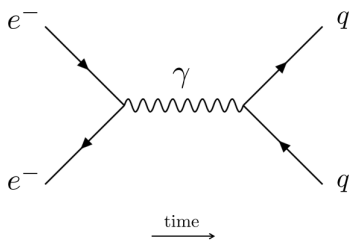


Figure 3: Feynman diagram of electron-positron pair annihilation that produces a quark-antiquark pair.

As the effects of the strong interaction are greater at larger distances, the quarks and gluons formed are strongly coupled together. They are confined to hadrons, and as a result the entire fragmentation process is visualised as jets of hadrons originating at the site of the original quark-antiquark pair moving in approximately opposite directions (Fig. 4).

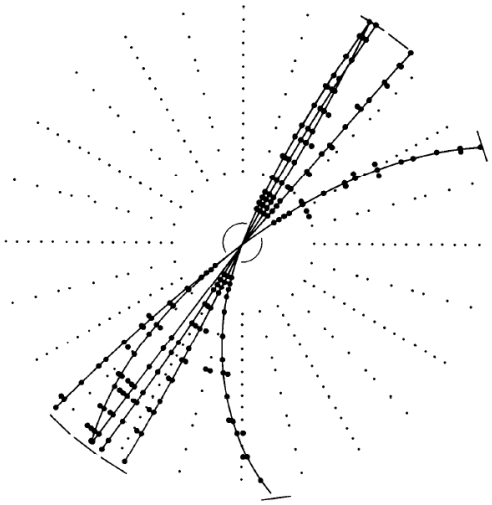


Figure 4: Tracks of the jets produced from a high-energy electron-positron pair annihilation [8].

2.5 The string model

The above description of fragmentation makes use of the concept of confinement to explain what is seen experimentally. A more physical model that describes the phenomenon was developed based on observations of the potential of the colour field. We go back to the idea of the strong coupling constant to do this.

As previously mentioned, the coupling constants change according to the distance from the charged particle from which they are measured. In the case of the electromagnetic force, perturbation theory can be used to obtain an asymptotic value, the above figure for α_{EM}, which can then be used to find the effective value of the coupling constant at shorter distance scales. Perturbation theory begins with the free field. It then adds corrections to this for the disturbances, or perturbations, that we see in quantum mechanics [10]. This approach can be used effectively for α_{EM} since the corrections become small after a certain number of iterations. However for α_S we cannot take the infinite distance limit in a similar manner at distances greater than 1 fm, since the perturbative expression for the coupling constant diverges [9].

The failure of perturbation theory forces physicists to use a different approach to address QCD dynamics at long distances. One such method is called lattice QCD. In this scheme, spacetime is discretised into a lattice-like structure [11]. Using this, the potential of the strong field can be measured. From the result of this, we can see that at distances less than 1 fm, the potential behaves much like the Coulomb potential of the electric field, with a characteristic $\frac{1}{r}$ behaviour. This changes into a linear potential at greater distances. This combination of a $\frac{1}{r}$ and r potential, as seen in Fig. 5, is called the Cornell potential, and characterises the strong field [12].

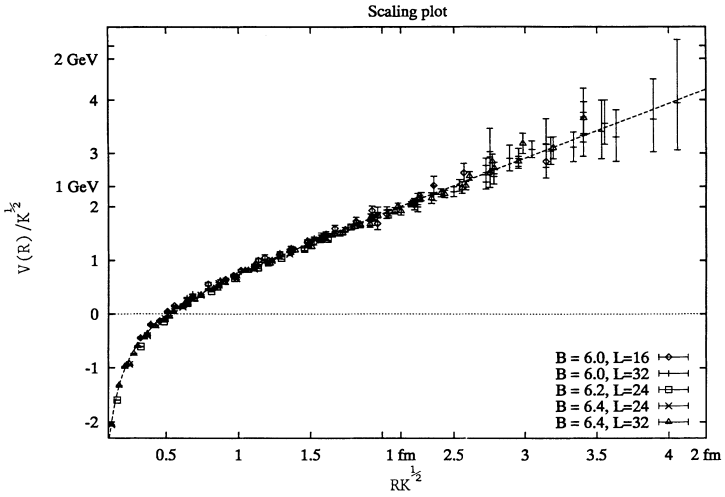


Figure 5: The quark-antiquark Cornell potential as a function of separation distance, measured using lattice QCD [13].

The behaviour of the Coulomb potential, which is documented from observations of the electric field, can be applied to the colour field at short distances of less than 1 fm. Beyond this point, the behaviour of the strong field is not well-known. Since the potential of the field becomes linear, its natural representation is a string. This string can be visualised as the tube-like field between a quark-antiquark pair (Fig. 6). The string tension, or the uniform energy density per unit length, is denoted by κ, with $\kappa \approx 1$ GeV fm^{-1} [14]. Its potential is therefore $V(r) = V_0 - \frac{A}{R} + \kappa r$ [12]. The string is labelled as a $1+1$ dimensional object as it moves through the z-direction and time, neglecting motion in the transverse (x and y) directions. This representation is called the string model.

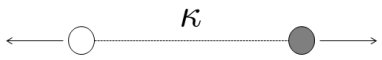


Figure 6: Visualisation of the string with tension κ between two quarks.

2.6 The Lund function

Since jet fragmentation is a phenomenon of the strong force, its details cannot be calculated from first principles. For this purpose, phenomenological models have been developed. Once such model is the string model, which the Lund function plays the crucial role of parametrising. This function parametrises the probability that a hadron with a certain mass will be produced with a given fraction of the endpoint quark's energy-momentum.

The production of hadrons at separate stages of the string breakup are causally disconnected events, as shown in Fig. 7 [15]. Because of this, the details of the fragmentation process can actually be calculated going backwards through time. This is what the Lund function does; it determines probability of formation of the end hadrons, with the largest energy-momentum, first, even though they are the last to be produced. It then iteratively splits off more hadrons until all the energy resides in the final hadrons produced.

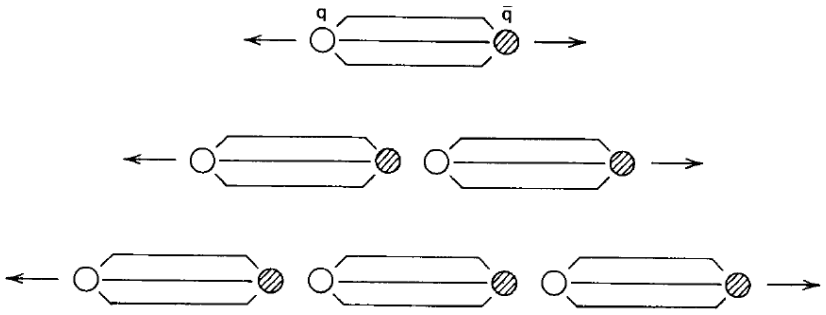


Figure 7: The string breakup when a quark and antiquark separate. Note that the colour fields between disconnected particles do not affect each other [8].

The Lund function is defined as

$$f(z) = N\frac{(1-z)^a}{z}e^{\frac{-bm_\perp^2}{z}} \tag{1}$$

(see [16] for a derivation). In this function, z denotes the fraction of the remaining energy-momentum that the hadron takes; N is the normalisation constant; $m_\perp^2$ denotes the sum of squares of the hadron's mass and transverse momentum; a is a dimensionless parameter related to the type of breakup; and b is a universal parameter with dimension GeV^{-2}. This function can be turned into a probability density when it is normalised to the unit integral.

3 Reparametrisation of the Lund function

3.1 The correlation between *a* and *b*

The Lund function has z values ranging between 0 and 1. When $z = 1$, all of the endpoint quark's energy is taken by a single hadron, while for lower z values, a fraction $(1 - z)$ will remain from which other hadrons can be produced. The shape of the function depends on the expected value and variance of z. The latter parameter is determined by the values of a and b, which are very closely correlated (Fig. 8). The issue with this strong correlation is that, if we want to fit the parameters to data and assign them uncertainties (as we need to due to this model being phenomenological), it would not make sense to do so individually. In this project, we therefore reparametrise the Lund function in terms of variables that have a weaker correlation.

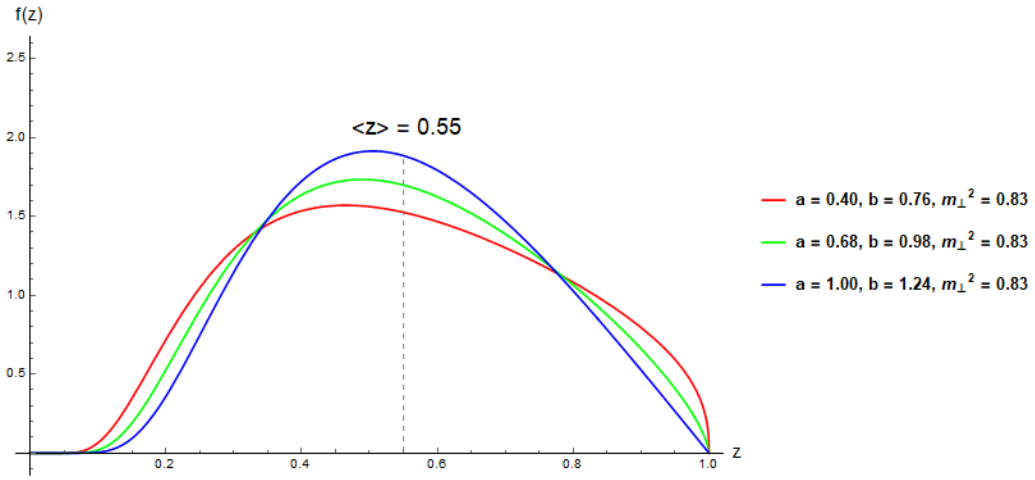
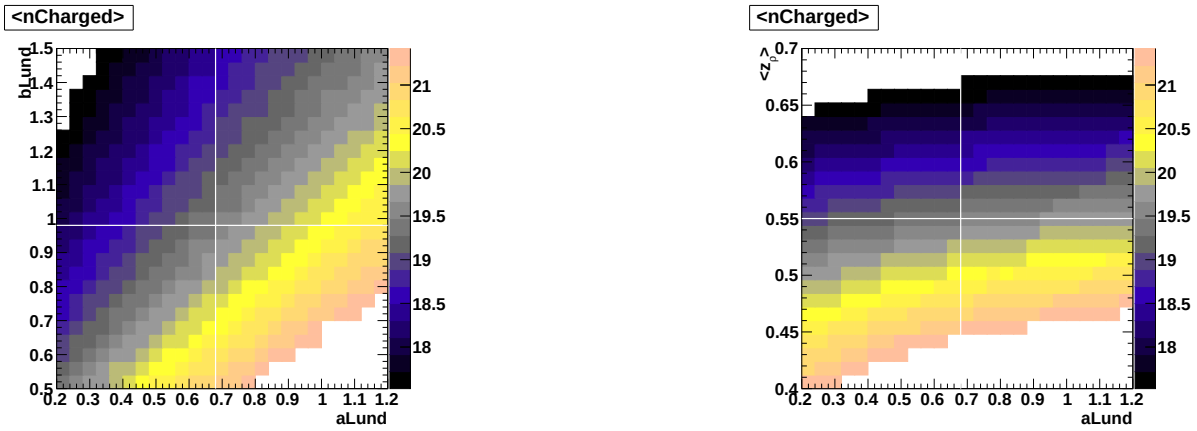


Figure 8: The normalised Lund function for various a and b values. The expected z value and $m_\perp$ was chosen to be the same for all curves. Note that a and b together control the width of the distribution. In the pathological case that both are very high, the graph becomes the Dirac delta function. Physically, this means that all hadrons produced would take the same fraction of the remaining energy-momentum of the string.

The variables chosen for the reparametrisation were a and the expected value of z, denoted by $\langle z \rangle$. To observe that these parameters have a weaker correlation than a and b, we refer to Fig. 9. We can see that with the new parameters, to produce the same number of charged particles, we can change a with a much smaller change in $\langle z \rangle$ than would be needed in b, as is evident from the flatter gradient of Fig. 9b compared to that of Fig. 9a.



(a) (b)

Figure 9: Contour plots showing the average number of charged particles produced in PYTHIA runs for the process $e^+e^- \to Z \to d\bar{d} \to$ hadrons, as functions of (a) a and b, and (b) a and $\langle z \rangle$ [17]. Note that the colour scale is the same for both figures, but the vertical axes have different scales.

The way the reparametrisation works is that, given some $\langle z \rangle$ and a (and fixing $m_\perp$), b can be found. This is done by using the definition of the expected value

$$\langle z \rangle = \frac{\int_0^1 z f(z)\, dz}{\int_0^1 f(z)\, dz}. \tag{2}$$

We therefore look for solutions to

$$\frac{\int_0^1 z f(z)\, dz}{\int_0^1 f(z)\, dz} - \langle z \rangle = 0, \tag{3}$$

or, using Eqn. 1:

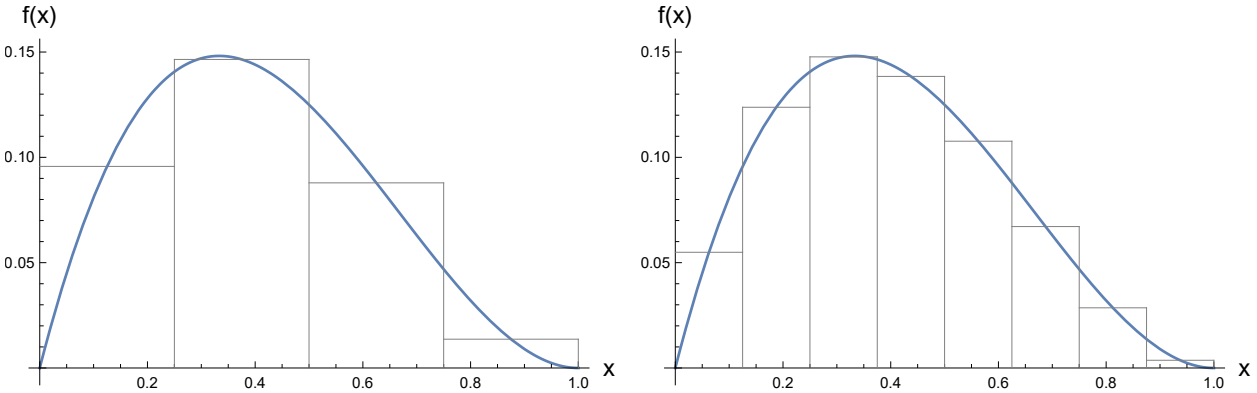
$$\frac{\int_0^1 (1-z)^a e^{\frac{-bm_\perp^2}{z}}\, dz}{\int_0^1 \frac{(1-z)^a}{z} e^{\frac{-bm_\perp^2}{z}}\, dz} - \langle z \rangle = 0. \tag{4}$$

From this, it is clear that the problem now involves integrating to find the expected value of the normalised Lund function, finding the root, and iterating until b is found. This is not a straightforward process. The integrals cannot be solved analytically. Similarly, the root-finding part is not trivial. This leaves us with numerical methods to find the solution. Fortunately, each of these problems is one-dimensional, for which there are well-established methods in the literature. We will now discuss some methods for numerical integration and root-finding.

3.2 Numerical integration

The first step to solving Eqn. 4 for b is integrating to find the expected value of the normalised Lund function. We recall from basic calculus that this is equivalent to finding the area under the curve. Since we cannot compute the integral directly, we have to do this step by calculating Riemann sums. To illustrate the following formulas, we take the cubic function $f(x) = x(x-1)^2$. Evaluating this function for x between 0 and 1 analytically, we get $\frac{1}{12}$.

We recall that Riemann sums are a way of approximating an integral by a finite sum. This works by dividing the area into shapes and summing their individual areas. The most basic method is to divide the region into rectangles. Using four panels, as shown in Fig. 10a, the sum gives $\frac{11}{128}$, which has an error of 3.125%. With eight panels, as shown in Fig. 10b, the area is $\frac{43}{512}$, which has an error of 0.781%. Clearly increasing the number of panels gives an answer closer to the actual integral.



(a) (b)

Figure 10: The Riemann sums for $f(x)$ with (a) four, and (b) eight panels.

The Newton-Cotes formulas give more accurate approximations than Riemann sums, but are only effective for integrals of the form

$$\int_a^b f(x)\,dx, \tag{5}$$

where $f(x)$ is a smooth function, such as a polynomial [18]. The familiar composite trapezoidal rule is one of the Newton-Cotes formulas. In this rule, rather than fixing the width of the area as a horizontal line, we take the trapezium with one side as the line joining the endpoints of the function evaluated at the chosen abscissas. Summing the areas of the eight trapeziums shown in Fig. 11 gives 0.083, which has an error of 0.040%. This is significantly better than the result from Fig. 10b. The error of the composite trapezoidal rule is of order $O(h^2)$, where h is the width of the panels, also called the step size [19].

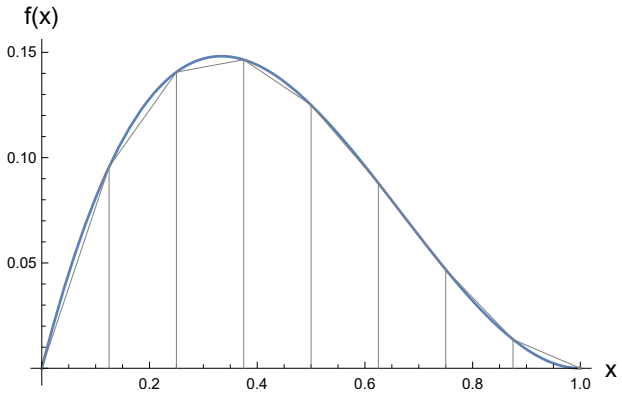


Figure 11: The trapezoidal rule with eight panels.

It is possible to enhance the convergence rate of some numerical methods, such as the trapezoidal rule, by using a process called Richardson extrapolation [20]. This procedure increases the order of the error for each iteration. More precisely, the numerical method gives a result of the form

$$F = F(h) + E(h), \tag{6}$$

where F is the desired property (in the case of the trapezoidal rule, this is the integral); h is the step size; $F(h)$ is the approximation; and $E(h)$ is the error. The latter can also be written as

$$E(h) = ch^k + O\left(h^{k+1}\right), \tag{7}$$

where c and k are constants. Substituting Eqn. 7 into Eqn. 6 gives

$$F = F(h) + ch^k + O\left(h^{k+1}\right). \tag{8}$$

Eqn. 8 can be solved by using another equation to eliminate the ch^k term. We obtain this by using a different step size of $\frac{h}{2}$, which gives

$$F = F\left(\frac{h}{2}\right) + c\left(\frac{h}{2}\right)^k + O\left(\frac{h}{2}^{k+1}\right) \tag{9}$$

$$= F\left(\frac{h}{2}\right) + c\left(\frac{h}{2}\right)^k + O\left(h^{k+1}\right). \tag{10}$$

We then subtract Eqn. 8 from 2^k multiplied by Eqn. 10 to give

$$(2^k - 1)F = 2^k F\left(\frac{h}{2}\right) - F(h) + O\left(h^{k+1}\right) \tag{11}$$

$$F = \frac{2^k F\left(\frac{h}{2}\right) - F(h)}{2^k - 1} + O\left(h^{k+1}\right). \tag{12}$$

If we define

$$G(h) = \frac{2^k F\left(\frac{h}{2}\right) - F(h)}{2^k - 1}, \tag{13}$$

then

$$F = G(h) + O\left(h^{k+1}\right). \tag{14}$$

F from Eqn. 14 has an error that is one order higher than $F(h)$ from Eqn. 8. We therefore need to calculate $G(h)$, which is precisely what Richardson extrapolation involves.

When Richardson extrapolation is combined with the trapezoidal rule, we have Romberg integration. In the trapezoidal rule, the exponent k in the error term is 2 [21]. The error in Romberg integration is of order $O(h^{2n+2})$, where n is the number of panels used in the calculation [22]. Using this method with just two iterations, the result of integrating $f(x)$ was exact to the computer's precision. Therefore this method is preferable to using Riemann sums or the composite trapezoidal rule.

Gaussian quadrature uses a distinctly different method to integrate functions. The previously mentioned methods all used equally-spaced abscissas at which the function's value was calculated, and multiplied by certain weighting coefficients. Gaussian quadrature allows us to choose the abscissas and the weighting coefficients. It accurately computes integrals of the form

$$\int_a^b w(x) f(x)\, dx, \tag{15}$$

where $w(x)$ is the weighting function and $f(x)$ is a smooth function. Rather than fixing the points of evaluation as the trapezoidal rule does, Gaussian quadrature balances the positive and negative errors of the new function (Fig. 12). One of the difficulties with Gaussian quadrature lies in computing appropriate abscissas and weighting functions; however in some cases these values have been tabulated in texts. If the quadrature rule uses n points to calculate the integral, the result is exact for polynomials up to order $2n-1$. Gaussian quadrature converges exponentially with n. Note that this convergence deteriorates if the integral does not follow the form of Eqn. 15 [21].

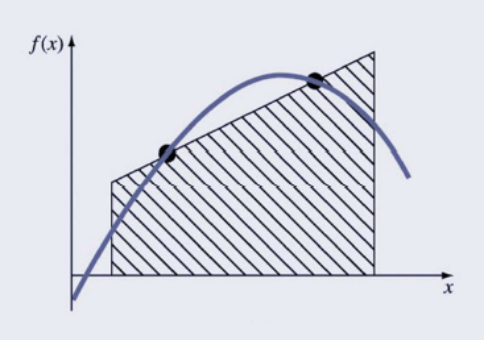


Figure 12: A visualisation of Gaussian quadrature [23]. The area of the shaded region is exactly the same as the area under the curve between the marked points since the positive and negative errors of the former cancel out.

3.3 Numerical root-finding

Once we have evaluated the integrals, we subtract the input expected value for z and find the root. From high school level mathematics, we can find the roots of first- and second-order equations of one variable using techniques such as the quadratic formula. However for more complex equations, an analytic solution may not be easy to find - if it exists at all - and this is the case with our problem.

Many root-finding methods require a starting guess for the root. It is important that this guess is 'good'. If it is not, the method may not converge, or converge to the wrong root. Unfortunately, for a general equation, there is no way of knowing what a good guess is. For our problem involving the Lund function, we know from dimensional analysis that b should be of order 1 GeV^{-2}. Furthermore, its default value in PYTHIA is 0.98 GeV^{-2}, which gives us a starting point [24].

In addition to a starting guess, some methods require the root to be bracketed. Again, it is essential that this bracket contains one root or else the method could either not terminate or find the wrong root. Since the Lund function is monotonic, we know that there is exactly one root in the range z_1, z_2 if $f(z_1)f(z_2) < 0$. From the initial guess of 0.98 GeV^{-1}, we can vary b by roughly one order of magnitude in each direction to give a bracket.

The simplest way of finding the root of an equation is the bisection method (Fig. 13). This method requires the root to be bracketed, and iteratively halves this bracket until it is small enough to meet the input tolerance requirement [21]. While the bisection method has a slow convergence rate of $\frac{1}{2}$ [25], it is the most reliable root-finding method because it always finds the root, provided that the bracket contains one.

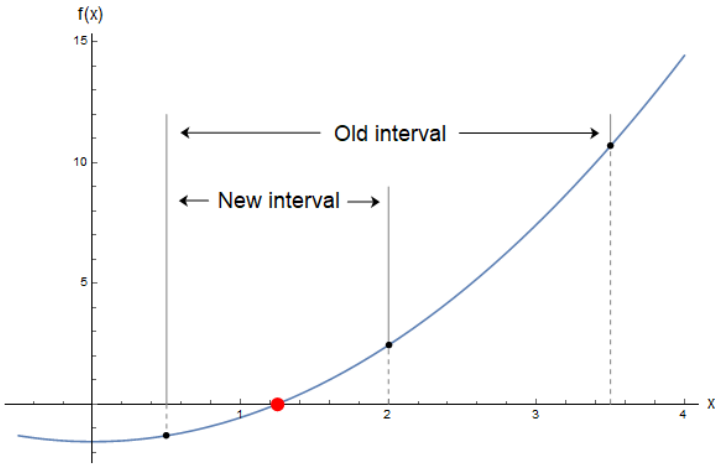


Figure 13: A graphical depiction of the bisection method with the root marked in red. The bracket halves for each iteration.

The secant method (Fig. 14) is a linear interpolation method. It requires two initial guesses (a and b) of the root but does not need a bracket. This method assumes that the function behaves in a linear fashion near its root. It creates a linear fit between a and b and interpolates or extrapolates until it finds the x-axis. The abscissa, given by

$$c = b - f(b)\frac{b-a}{f(b)-f(a)}, \tag{16}$$

is taken as the new guess for the root. The method then takes a as the old value of b, and b as the old value of c, and repeats the linear fit until the estimate is close enough to the root [21].

The main disadvantage of the secant method is that the root is never bracketed. This may cause the output to be the wrong root, or, if it encounters a local extremum, the method will not converge. The secant method's convergence rate is $\frac{1}{2^\phi}$, where ϕ is the golden ratio, $\frac{1}{2}(1+\sqrt{5})$ [25]. It is therefore less reliable than the bisection method but faster if it converges.

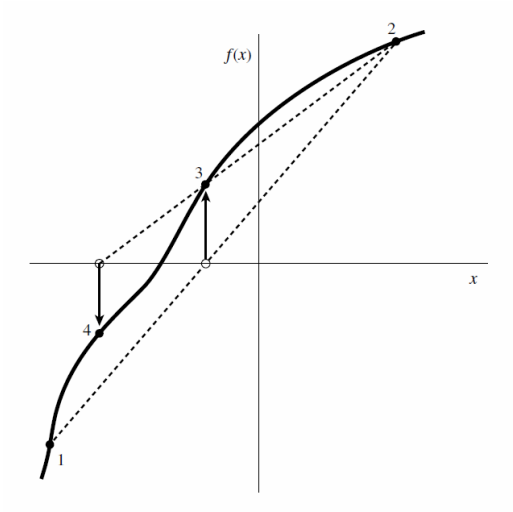


Figure 14: A graphical depiction of the secant method [18]. The method creates a linear fit between 1 and 2 to obtain 3 as the next point. It discards point 1 and repeats the fit between 2 and 3 to obtain 4.

If the function is smooth, then quadratic interpolation can be used to find its root. One such method is inverse quadratic interpolation (Fig. 15). This method assumes that the function behaves in a quadratic manner near its root. It fits an inverse quadratic function to three known points and takes the abscissa of the new function to be the next estimate of the root. If the three points are $(a, f(a))$, $(b, f(b))$, and $(c, f(c))$, then using Lagrange's formula,

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]}. \tag{17}$$

Because we look for the abscissa, we set $y = 0$. This gives

$$x = b + \frac{P}{Q}, \tag{18}$$

where, if we set

$$R = \frac{f(b)}{f(c)}, \tag{19}$$

$$S = \frac{f(b)}{f(a)}, \text{ and} \tag{20}$$

$$T = \frac{f(a)}{f(c)}, \tag{21}$$

then

$$P = S(T(R - T)(c - b) - (1 - R)(b - a)), \text{ and} \tag{22}$$

$$Q = (T - 1)(R - 1)(S - 1). \tag{23}$$

Now the $\frac{P}{Q}$ term from Eqn. 18 should be a small correction, so $x \approx b$. The method then keeps the two old points which bracket x the most tightly and repeats the fit for those three points. This process is repeated until x is close enough to the root to meet the tolerance requirement [18]. The convergence rate of inverse quadratic interpolation is $\frac{1}{2^{1.84}}$ [26]. This is faster than the secant method, but only converges if the function behaves quadratically near its root.

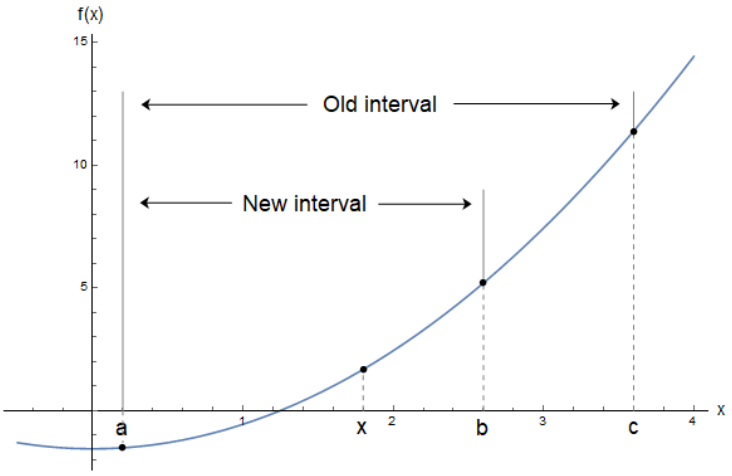


Figure 15: A graphical depiction of inverse quadratic interpolation [21]. The method fits a quadratic function to a, b, and c. The zero of this function, x, is taken as the next best guess of the root. The method then takes the two old points which most tightly bracket x and repeats the quadratic fit.

Brent's method is a more complicated root-finding method that combines the bisection method and inverse quadratic interpolation. It requires an initial guess for the root and its bracket. The method begins by using inverse quadratic interpolation to find the next best guess of the root. In the case that $Q \approx 0$, x falls outside its bracket, or the bracket does not shrink quickly enough, Brent's method uses the bisection method instead. This method is guaranteed to converge since it resorts to the bisection method if needed, but uses the much faster inverse quadratic interpolation to narrow the bracket. Its convergence rate is therefore bound between $\frac{1}{2}$ (from the bisection method) and $\frac{1}{2^{1.84}}$ (from inverse quadratic interpolation).

As the literature only gives bounds for the convergence rate of Brent's method, a better indication of its performance was desired. We compared the performance of Brent's method with the secant method, since from their convergence rates it was unclear which method was faster. Both methods were used to find the root of Eqn. 4, where all variables except for a were fixed. Brent's method was faster for all cases, using at least two iterations fewer than the secant method, which took eight iterations on average. The root was found to a precision of six decimal places, which is as precise as the user can reasonably expect given the constraints of the string model. It is therefore clear that Brent's method is indeed faster than the secant method for our problem involving the Lund function.

An even faster root-finding method than all those previously mentioned is known as the Newton-Raphson method. This method requires an initial guess of the root as well as the first derivative of the function. The next best estimate of the root is found from the Taylor series expansion,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta + \dots. \tag{24}$$

For small δ and well-behaved functions, the terms beyond linear are negligible, so we have

$$f(x+\delta) \approx f(x) + f'(x)\delta. \tag{25}$$

Since we seek the root, we set $f(x+\delta) = 0$ to get

$$\delta = -\frac{f(x)}{f'(x)}. \tag{26}$$

Eqn. 26 is the amount by which we increment x to obtain the next best guess of the root. If we look at this method graphically, we see that we approximate the function by its tangent curve. We then extrapolate this tangent to the x-axis and take the abscissa as the new estimate (Fig. 16) [21].

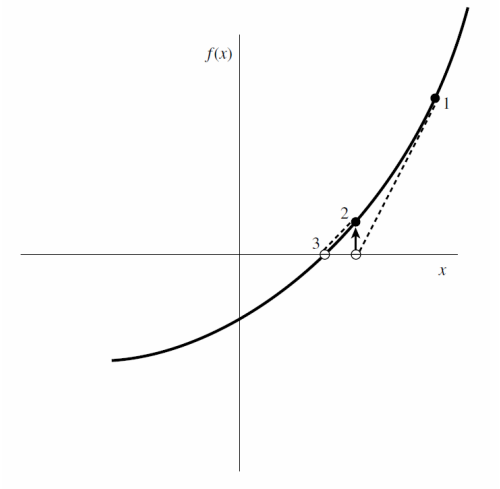
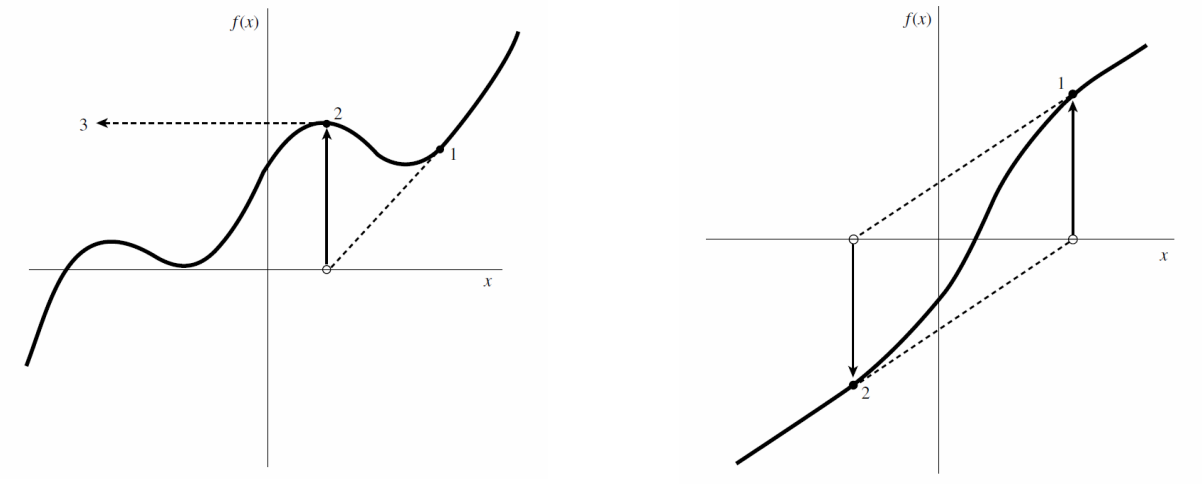


Figure 16: A graphical depiction of the Newton-Raphson method [18]. The method increments the initial guess of the root, 1, to obtain 2, and so on.

The convergence rate for the Newton-Raphson method is $\frac{1}{4}$ [25], which we see is significantly faster than all of the previously-discussed methods. However we can immediately see the possible issues we could have in the implementation of this method. The main problem is that every iteration requires computation of the function's derivative at arbitrary points. This may not be easy as the derivative may be difficult to find. It is possible to approximate it from first principles, but this would drive the convergence rate down [18]; and in fact using Brent's method is faster in this case. The function may also be poorly approximated by its tangent. In particular, if the method encounters a local extremum (Fig. 17a), or a cycle (Fig. 17b), then it will never converge. A 'better' initial estimate of the root could prevent these scenarios from occurring; however, as discussed, it is usually not possible to determine what a 'good' root is before running the program. Since we know that the Lund function is well-approximated by its tangent, we conclude that the Newton-Raphson method is useful for problems when the function is monotonic and we have an analytic expression for the derivative.



(a) (b)

Figure 17: Possible cases when the Newton-Raphson method does not converge as it encounters (a) an extremum, or (b) a cycle [18].

3.4 Discussion of the implementation of the numerical methods

Romberg integration was the integration method of choice in this project. This was because it has a faster convergence rate than simply applying the composite trapezoidal rule continuously with an increasing number of panels each time. While Gaussian quadrature would have been an even better method as it has a faster convergence rate, it was unclear which weighting function should be used. Furthermore, the Lund function is not a polynomial so the use of higher order quadrature methods could reduce the accuracy of the result.

The root-finding method used was Brent's method. This method was chosen as it combined the best features of the bisection method and quadratic interpolation. Since we found $\langle z \rangle$ numerically, we did not have access to an analytic form of the derivative, so the Newton-Raphson method could not be used.

There were some issues encountered in the implementations of the above numerical methods, particularly during the integration process. One of these was that Romberg integration requires substitution of the endpoints of the function in during its calculations. In the case of the Lund function, z ranges from 0 to 1, but from Eqn. 1, we can see that the Lund function divides by z in two places. This meant that the lower limit of integration could not be defined at 0, but had to be sufficiently close such that the error in the integration component of the program did not exceed the error in the root-finding component. After some testing, a value was found for which the Lund function evaluated to 0 (to the precision of the computer system). Since the integral of the Lund function between 0 and this value did not contribute to the total integral of the Lund function, this value was chosen as the lower limit of integration.

Another issue was the choice of the number of panels to use in the integration. As with any numerical integration method, increasing the number of panels increases the accuracy of the answer but also prolongs the time the program takes to run. After some testing, a balance was found where the value of the integral stayed sufficiently constant for a range of panels.

4 Conclusion

This project explored the Lund function and its background of the experimental phenomenon, fragmentation, and the theoretical model, the string model. Due to the high correlation between two of the parameters in the Lund function, a reparametrisation was helpful in making data fitting and uncertainty assignments a more meaningful process. This was done using numerical methods which were successfully implemented in a program in C++ and Python (see Appendices A and B). It is hopeful that this reparametrisation can help in developing a greater understanding of the physics behind jet fragmentation.

References

[1] G. Aad *et al.*, "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". *Physics Letters B* **716** (2012) 1-29. arXiv: `hep-ex/1207.7214`

[2] S. Chatrchyan *et al.*, "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC". textitPhysics Letters B **16** (2012) 30-61. arXiv: `hep-ex/1207.7235`

[3] B. Webber, "Hadronization: Concepts and Models (From Perturbative to Non-Perturbative QCD)". (2008) Available: `http://www.hep.phy.cam.ac.uk/theory/webber/TrentoHadro.pdf`

[4] B. Andersson, S. Mohanty, and F. Soderberg, "Recent Developments in the Lund Model". (2002). arXiv: `hep-ph/0212122`

[5] T. Sjöstrand, S. Ask, J. R. Christiansen, R. Corke, N. Desai, P. Ilten, S. Mrenna, S. Prestel, C. O.

Rasmussen, and P. Skands, "A Brief Introduction to PYTHIA 8.2". *Computer Physics Communications* **191** (2015) 159-177. arXiv: `hep-ph/1410.3012`

[6] D. Griffiths, "Introduction to Elementary Particles". 2nd ed., *WILEY-VCH Verlag GmbH & Co. KGaA* (2008) pp. 59-85

[7] Fermilab. *Inquiring Minds, What is the world made of?* (2004). Available: `http://www.fnal.gov/pub/inquiring/matter/madeof/index.html`

[8] F. Halzen and A. D. Martin, "Quarks and Leptons: An Introductory Course in Modern Particle Physics". *John Wiley & Sons, Inc.* (1984) pp. 1-32

[9] W. E. Burcham and M. Jobes, "Nuclear and Particle Physics". *Addison-Wesley Publishing Company* (1995) pp. 223-245

[10] E. Vanden-Eijnden, "Introduction to regular perturbation theory". *Intro to math modeling* New York University, Courant Institute of Mathematical Sciences, Available: `https://cims.nyu.edu/~eve2/reg_pert.pdf`

[11] K. G. Wilson, "Confinement of Quarks". *Physical Review D* **10** (1974)

[12] D. K. Choudhury and K. K. Pathak, "Comments on the perturbation of Cornell potential in a QCD potential model". *Journal of Physics: Conference Series* **481** (2014). doi: `10.1088/1742-6596/481/1/012022`

[13] G.S. Bali and K. Schilling, "Static quark - anti-quark potential: Scaling behavior and finite size effects in SU(3) lattice gauge theory". *Physical Review D* **46** (1992) 5. arXiv: `hep-lat/0306005`

[14] G. Dessertori, I. G. Knowles, and M. Schmelling, "Quantum Chromodynamics: High Energy Experiments and Theory". *Oxford University Press* (2005) pp. 157-172

[15] T. Sjostrand, "Old Ideas in Hadronization: The Lund String (a string that works)". (2009) Available: `http://home.thep.lu.se/~torbjorn/talks/durham09.pdf`

[16] M. Gebert, "Something Strange? A glance into Parton Fragmentation". (2016)

[17] P. Skands, private correspondence

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical Recipes. The Art of Scientific Computing". 3rd ed., *Cambridge University Press* (2007) pp. 155-194, 442-463

[19] M. Rudman, *MAE2403: Aerospace Computational Mechanics.* Monash University lecture notes (2014, Sem. 2)

[20] J. Feldman, "Richardson Extrpolation", The University of British Columbia. (2000) Available: `http://www.math.ubc.ca/~feldman/m256/richard.pdf`

[21] J. Kiusalaas, "Numerical Methods in Engineering with Python". *Cambridge University Press* (2005) pp. 142-167, 198-230

[22] D. W. Harder, "Romberg Integration", University of Waterloo. (2005) Available: `https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/13Integration/romberg/complete.html`

[23] Y. Padayatchy, *ENG1060: Computing for Engineers.* Monash University lecture notes (2013 Sem. 2)

[24] P. Skands, S. Carrazza, and J. Rojo, "Tuning PYTHIA 8.1: the Monash 2013 Tune". *European Physical Journal C* **74** (2014) 3024. arXiv: `hep-ph/1404.5630`

[25] J. Rieger, *MTH3051: Introduction to Computational Mathematics.* Monash University lecture notes (2017, Sem. 1)

[26] B. S. van Lith, J. H. M. ten Thije Boonkkamp, and W. L. IJzerman, "Full linear multistep methods as root-finders". *CASA-report* **1705** (2017). arXiv: `math.NA/1702.03174`

Appendix

A: Reparametrisation source code in C++

```
#include <stdlib.h>
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;


class FunctionEncapsulator {

public:

  FunctionEncapsulator() {};

  virtual double f(double, vector<double>) {
    return 0.0;
  }

};


class LundFF : public FunctionEncapsulator {

public:

  LundFF() {}

  double fLund(double z, double aLund, double bLund, double mT2) {
    double val = pow(1.0 - z, aLund) * (exp(-bLund * mT2 / z) / z);
    return val;
  }

  virtual double f(double z, vector<double> args) {
    if (args.size() != 3) {
      cout << "lundFF.f(): Problem: wrong number of arguments" << endl;
      return 0.0;
    }

    double aLund = args[0];
    double bLund = args[1];
    double mT2  = args[2];

    return fLund(z, aLund, bLund, mT2);
  }

};


class ZLundFF : public LundFF {
```

```
50
51 public:
52
53   ZLundFF() {}
54
55   virtual double f(double z, vector<double> args) {
56     return z * LundFF::f(z, args);
57   }
58
59 };
60
61
62 class bSolve {
63
64 public:
65
66   bSolve() {};
67
68   void init(double aLundIn, double mIn, double pTIn, double zExpIn, double
         xIn, double yIn, int nIn, double bMinIn, double bMaxIn) {
69     aLund = aLundIn;
70     mT2 = pow(mIn, 2) + pow(pTIn, 2);
71     zExp = zExpIn;
72     x = xIn;
73     y = yIn;
74     n = nIn;
75     bMin = bMinIn;
76     bMax = bMaxIn;
77   }
78
79   double trapezoid(FunctionEncapsulator& f, vector<double> args, double
        iOld, int k, double x, double y) {
80     double iNew;
81
82     if (k == 1) {
83       iNew = (f.f(x, args) + f.f(y, args)) * (y - x) / 2.0;
84     }
85
86     else {
87       int l = pow(2, k - 2);
88       double h = (y - x) / l;
89       double w = x + h / 2.0;
90       double s = 0.0;
91
92       for (int i = 0; i < l; ++i) {
93         s = s + f.f(w, args);
94         w = w + h;
95       }
96
97     iNew = (iOld + h * s) / 2.0;
98     }
99
100    return iNew;
101  }
```

```
102
103   double romberg(FunctionEncapsulator& f, vector<double> args, double x,
          double y, int n) {
104     double entry0, newEntry, entry;
105
106     vector<double> rArray;
107     entry0 = trapezoid(f, args, 0, 1, x, y);
108     rArray.push_back(entry0);
109
110     for (int i = 2; i < n + 1; ++i) {
111       newEntry = trapezoid(f, args, rArray.back(), i, x, y);
112       rArray.push_back(newEntry);
113
114       for (int j = rArray.size() - 2; j > -1; --j) {
115         entry = (pow(4, rArray.size() - j - 1) * rArray[j + 1] - rArray[j
              ]) / (pow(4, rArray.size() - j - 1) - 1);
116         rArray[j] = entry;
117       }
118     }
119
120     return rArray[0];
121   }
122
123   double brent(double tol = 1e-10) {
124     double bLower, bUpper, b, b1, b2, b3, db, f1, f2, f3, num, numerator,
          den, denominator;
125
126     bLower = bMin;
127     bUpper = bMax;
128     b1 = bLower;
129     b2 = bUpper;
130     f1 = 0.0;
131
132     vector<double> args;
133     args.push_back(aLund);
134     args.push_back(b1);
135     args.push_back(mT2);
136     numerator  = romberg(zLundFF, args, x, y, n);
137     denominator = romberg(lundFF, args, x, y, n);
138
139     if (denominator == 0.0) {
140       cout << "Brent:ERROR! denominator = 0" << endl;
141     }
142
143     f1 = numerator/denominator - zExp;
144     if (abs(f1) < tol) {
145       return b1;
146     }
147
148     args[1] = b2;
149     f2 = romberg(zLundFF, args, x, y, n) / romberg(lundFF, args, x, y, n)
          - zExp;
150     if (abs(f2) < tol) {
151       return b2;
```

```
152     }
153
154     if (f1 * f2 > 0.0) {
155       cout << "Root not bracketed." << endl;
156       return 0.0;
157     }
158
159     b3 = 0.5 * (bLower + bUpper);
160
161     while (true) {
162       args[1] = b3;
163       f3 = romberg(zLundFF, args, x, y, n) / romberg(lundFF, args, x, y, n
              ) - zExp;
164
165       if (abs(f3) < tol) {
166         return b3;
167       }
168
169       if (f1 * f3 < 0.0) {
170         bUpper = b3;
171       }
172       else {
173         bLower = b3;
174       }
175
176       if ((bUpper - bLower) < tol * (abs(bUpper) < 1.0 ? bUpper : 1.0)) {
177         return 0.5 * (bLower + bUpper);
178       }
179
180       den = (f2 - f1) * (f3 - f1) * (f2 - f3);
181       num = b3 * (f1 - f2) * (f2 - f3 + f1) + f2 * b1 * (f2 - f3) + f1 *
              b2 * (f3 - f1);
182
183       if (den == 0.0) {
184         db = bUpper - bLower;
185       }
186       else {
187         db = f3 * num / den;
188       }
189
190       b = b3 + db;
191
192       if ((bUpper - b) * (b - bLower) < 0.0) {
193         db = 0.5 * (bUpper - bLower);
194         b = bLower + db;
195       }
196
197       if (b < b3) {
198         b2 = b3;
199         f2 = f3;
200       }
201       else {
202         b1 = b3;
203         f1 = f3;
```

```
204         }
205
206         b3 = b;
207         }
208     }
209
210     double aLund, mT2, zExp, x, y, bMin, bMax;
211     int n;
212     LundFF lundFF;
213     ZLundFF zLundFF;
214
215 };
216
217
218 int main() {
219     double aLund = 0.5;
220     double m = 0.3;
221     double pT = 0.3;
222     double zExp = 0.5;
223     double x = 1e-9;
224     double y = 1.0;
225     int n = 15;
226     double bMin = 0.1;
227     double bMax = 5.0;
228
229     bSolve instance1;
230     instance1.init(aLund, m, pT, zExp, x, y, n, bMin, bMax);
231     double ans = instance1.brent();
232
233     cout << " Ans = " << ans << endl;
234
235     return 0;
236 }
```

B: Reparametrisation source code in Python

```
1  import math
2  import sys
3
4
5
6  class bSolve():
7
8      def __init__(self, aLund, m, pT, zExp, x = float('1E-9'), y = 1.0, n =
           15, bMin = 0.1, bMax = 5.0):
9        self.aLund = aLund
10       self.m = m
11       self.pT = pT
12       self.zExp = zExp
13
14       self.x = x
15       self.y = y
16       self.n = n
17
18       self.bMin = bMin
19       self.bMax = bMax
20
21       self.lower = self.romberg(self.zLund, bMin, x, y, n) / self.romberg(
             self.lund, bMin, x, y, n)
22       self.upper = self.romberg(self.zLund, bMax, x, y, n) / self.romberg(
             self.lund, bMax, x, y, n)
23
24       if not self.lower <= self.zExp <= self.upper:
25         print("z out of bounds.")
26         sys.exit()
27
28     def lund(self, b, z):
29       return (((1 - z) ** self.aLund) / z) * (math.e ** ((-b * (self.m **
             2 + self.pT ** 2)) / z))
30
31     def zLund(self, b, z):
32       return z * self.lund(b, z)
33
34     def trapezoid(self, f, b, Iold, k, x, y):
35       if k == 1:
36         Inew = (f(b, x) + f(b, y)) * (y - x) / 2.0
37
38       else:
39         l = 2 ** (k - 2)
40         h = (y - x) / l
41         w = x + h / 2.0
42         s = 0.0
43
44         for i in range(l):
45           s += f(b, w)
46           w += h
47         Inew = (Iold + h * s) / 2.0
48
```

```
49        return Inew
50
51      def romberg(self, f, b, x, y, n):
52        rArray = [self.trapezoid(f, b, 0, 1, x, y)]
53
54        for i in range(2, n + 1):
55          rArray.append(self.trapezoid(f, b, rArray[-1], i, x, y))
56
57          for j in range(len(rArray) - 2, -1, -1):
58            entry = (4 ** (len(rArray) - j - 1) * rArray[j + 1] - rArray[j])
                  / (4 ** (len(rArray) - j - 1) - 1)
59            rArray[j] = entry
60
61        return rArray[0]
62
63
64      def brent(self, tol = float('1e-10')):
65        bMin = self.bMin
66        bMax = self.bMax
67        b1 = bMin
68        b2 = bMax
69
70        f1 = self.romberg(self.zLund, b1, self.x, self.y, self.n) / self.
             romberg(self.lund, b1, self.x, self.y, self.n) - self.zExp
71        if f1 == 0.0:
72          return b1
73        f2 = self.romberg(self.zLund, b2, self.x, self.y, self.n) / self.
             romberg(self.lund, b2, self.x, self.y, self.n) - self.zExp
74        if f2 == 0.0:
75          return b2
76
77        if f1 * f2 > 0.0:
78          print("Root not bracketed.")
79          return
80
81        b3 = 0.5 * (bMin + bMax)
82
83        while True:
84          f3 = self.romberg(self.zLund, b3, self.x, self.y, self.n) / self.
               romberg(self.lund, b3, self.x, self.y, self.n) - self.zExp
85
86          if abs(f3) < tol:
87            return b3
88
89          if f1 * f3 < 0.0:
90            bMax = b3
91          else:
92            bMin = b3
93
94          if (bMax - bMin) < tol * max(abs(bMax), 1.0):
95            return 0.5 * (bMin + bMax)
96
97          den = (f2 - f1) * (f3 - f1) * (f2 - f3)
98          num = b3 * (f1 - f2) * (f2 - f3 + f1) + f2 * b1 * (f2 - f3) + f1 *
```

```
                b2 * (f3 - f1)
 99
100           try:
101             db = f3 * num / den
102           except ZeroDivisionError:
103             db = bMax - bMin
104           b = b3 + db
105
106           if (bMax - b) * (b - bMin) < 0.0:
107             db = 0.5 * (bMax - bMin)
108             b = bMin + db
109
110           if b < b3:
111             b2 = b3
112             f2 = f3
113           else:
114             b1 = b3
115             f1 = f3
116           b3 = b
117
118
119
120   if __name__ == "__main__":
121       x = float('1e-9')
122       y = 1.0
123       n = 15
124
125       aLund = 0.5
126       m = 0.3
127       pT = 0.3
128       z = 0.5
129
130       bMin = 0.1
131       bMax = 5.0
132
133       instance = bSolve(aLund, m, pT, z, x, y, n, bMin, bMax)
134       ans = instance.brent()
135       print(ans)
```