UNIVERSITY OF CAPE TOWN

MSc THESIS

Parallel computing solutions to the Balitsky-Kovchegov equation

Author:
Charlotte Stephanie Hillebrand-Viljoen

Supervisor:
Heribert Weigert

A thesis submitted in fulfilment of the requirements for the degree of Master of Science (Theoretical Physics)

in the

Department of Physics

April 22, 2016

Declaration of Authorship

I, Charlotte Stephanie Hillebrand-Viljoen, declare that this thesis titled, "Parallel computing solutions to the Balitsky-Kovchegov equation" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Signed by candidate

Date: 12 February 2016

"Curiouser and curiouser!" cried Alice. (She was so much surprised that for the moment she quite forgot how to speak good English.)

So many out-of-the-way things had happened lately, that Alice had begun to think that very few things indeed were really impossible.

Lewis Carroll's *Alice in Wonderland*

UNIVERSITY OF CAPE TOWN

Abstract

Faculty of Science
Department of Physics

Master of Science (Theoretical Physics)

Parallel computing solutions to the Balitsky-Kovchegov equation

by Charlotte Stephanie Hillebrand-Viljoen

The JIMWLK (Jalilian-Marian, Iancu, McLerran, Weigert, Leonodiv and Kovner; pronounced "gym walk") equation describes the energy evolution of observables in the colour glass condensate (CGC) state of matter, which is particularly relevant to collider physics. Currently there are many implementations of JIMWLK evolution in the spirit of the factorised Balitsky-Kovchegov (BK) equation for the total cross section, including a number of efforts to consistently implement evolution at next-to-leading-order[1–5]. Aside from NLO there is a growing interest in studying new, more exclusive, observables, such as single transverse spin asymmetries and transverse momentum distributions[6–8]. These require the inclusion of new degrees of freedom, which can be done systematically by extending the Gaussian truncation of the JIMWLK equation[9][10]. This necessarily increases the computational demands, both in terms of floating point operations and of storage requirements. After a discussion of the theoretical context, we address the first computational step and introduce new, parallelised methods in code that evolves the BK equation. Parallelisation of BK evolution using NVIDIA CUDA with implementation on a commercially available graphical processing unit (GPU) results in performance improvements of roughly an order of magnitude over comparable serial programmes. This also allows us to implement test cases which are often neglected. The code presented here covers only the total cross section case, but it is written with extension to more interesting cases in mind and we discuss some such potential applications.

Acknowledgements

I certainly could not have come this far without a great deal of help from many people, some of whom I will attempt to call out by name.

My family have been wonderfully supportive: thank you especially for providing impartial eyeballs and good advice on many occasions. Brandon is responsible for the current state of my sanity, being mostly intact, as well as other lovely things, not least of which is assisting in the debugging process far beyond the call of duty.

The UCT Physics department has taught me how little I know and how to get by in spite of it. My supervisor, Heribert Weigert, must be thanked for all of his guidance and especially for having the patience of a very patient person in explaining over and over and over and over when I don't understand things. Judith Alcock-Zeilinger has been my partner in ~~crime~~ figuring out how to deal with this QCD-thing.

Andrecia Ramnath and Kari Rummukainen each provided a number of helpful discussions about the numerics of the BK equation, for which I am very grateful.

I am thankful for the financial support of the National Research Foundation (although they would like you to know that they don't necessarily endorse the contents of this thesis).

Contents

List of Figures

List of Symbols

A	Gauge field (usually the colour field).
b	Impact parameter.
g	COupling strength.
$N(r)$	In older sources, $T(r)$ is called $N(r)$.
p	Momentum of the 'target' particle in DIS.
q	Momentum of the photon probe in DIS.
Q_s	Saturation scale = $\frac{1}{R_s}$
R_s	Correlation length.
$S(r)$	Dipole correlation as a function of separation. Corresponds to the S matrix.
$S_{\boldsymbol{xy}}$	The dipole correlation for a dipole with points at x and y, or the energy-dependent expectation value thereof.
t^a	An element of the colour group algebra.
$T(r)$	$1 - S(r)$. Corresponds to the T matrix. Called $N(r)$ in older sources.
$U_{\boldsymbol{x}}$	Wilson line (path ordered exponential) representing the colour rotation experienced by a quark at $\boldsymbol{x}$.
$U^\dagger_{\boldsymbol{x}}$	Wilson line (path ordered exponential) representing the colour rotation experienced by an antiquark at $\boldsymbol{x}$.
$\tilde{U}_{\boldsymbol{x}}$	Adjoint Wilson line (path ordered exponential) representing the colour rotation experienced by a gluon at $\boldsymbol{x}$.
x_{bj}	Bjorken x $= \frac{Q^2}{2p.q} = e^{-Y}$ (where $p.q$ is the Minkowski product)
Y	Rapidity $= \frac{1}{\ln x_{bj}}$
$\nabla^a_{\boldsymbol{x}}$	Functional differential operator used to define the JIMWLK Hamiltonian.
λ	scaling speed
Λ	sets the scale for Q_s and is typically of order Λ_{QCD}
σ	cross-section

Chapter 1

Introduction

1.1 The CGC, background fields and Wilson lines

The colour glass condensate is a state of matter particularly relevant to collider physics. It describes a particle moving at very high momentum, from a reference frame in which the particle is length-contracted and time-dilated, so that it appears as an extremely thin sheet of very slowly changing material – analogous to the eponymous glass. The non-abelian QCD gauge field leads to the kinematic enhancement of gauge boson production by a factor of $\ln \frac{1}{x_{bj}}$ (see Section 1.2) through Feynman diagrams as in Figure 1.1 – particularly the three-gluon vertex. For the massive W and Z bosons, this enhancement is prevented from appearing by suppressions due to the mass, but this is not the case for the massless gluon, resulting in a huge number of radiated low-momentum "soft" gluons (hence "colour condensate"). These soft gluons produce a significant background field which must be included in correlator calculations [11].

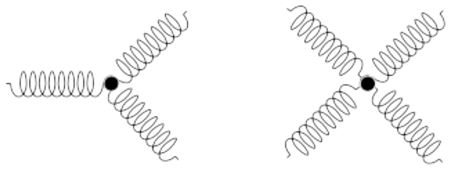


FIGURE 1.1: QCD is a non-abelian theory and the gauge boson, the gluon, can directly produce more gluons, as illustrated in these Feynman diagram vertices. The three-gluon vertex in particular is kinematically enhanced, leading to significant background fields of soft gluons.

This concept is key to managing the contributions of the radiated gluons. All soft particles (those which fall below some momentum cutoff) are not treated individually, but have their effect absorbed into the background field. Because the background field consists of soft particles only, it will have negligible effect on the momenta of the remaining hard (high momentum) particles. The background field can nonetheless interact with the hard particles, specifically through colour rotation. The study of situations involving such fields has a long history[12–33]. The colour rotation for a hard particle due to the background field is expressed by the Wilson line for that particle. This Wilson line contains all the information about the interaction between the hard particle and the indefinite, but generally large, number of soft particles in the background field, since these interactions must be restricted to colour rotations by the very nature of the background field. These Wilson lines can then be used in calculating correlation functions. Most applications of QCD will consider n-point functions, which describe the interaction between n particles in a vacuum, we are forced to consider the concept of a "Wilson line n-point function", which describes the interaction of n hard particles and indefinitely many soft particles, which form the background field. The Wilson line is a path ordered exponential, given by[34]

$$U\big(x(\tau), x(\tau_0)\big) = P\Big\{ \exp\big[ig \int_0^1 d\tau \frac{dx^\mu}{d\tau} A_\mu^a(x(\tau))t^a\big]\Big\} \tag{1.1}$$

where A is the gauge field – here the colour field – t^a is a generator of the gauge group algebra and g gives the coupling.

A consequence of this formulation and of the existence of a background field is that coincidence limits in correlators take on a new importance. In an ordinary n-point function, particle positions merely describe their relative separations, but in a Wilson line n-point function, the positions of the particles additionally describe their situation relative to the background field and thus are important in determining their Wilson lines. Coincident particles thus have Wilson lines determined by identical field configurations; an antiparticle coincident with a particle will have a Wilson line that is conjugate to the particle's Wilson line. These properties frequently result in significant simplifications to n-point functions in coincidence limits. These symmetries are a guiding principle in making necessary truncations in correlator calculations, such as the Gaussian truncation (GT)[9, 10, 35], which is discussed in more detail in Chapter 2.

1.2 Deep inelastic scattering

It is convenient to discuss the details of $q\bar{q}$ calculations and the kinematic region in which we are working by using the context of deeply inelastic scattering (DIS) events. These are events in which a projectile electron interacts with a target proton or nucleus, shattering the target with overwhelming probability.

(A quantum theory must allow for the possibility that the target is not shattered, but the interaction is inelastic in the sense that this possibility is strongly suppressed.)[36] The leading order contribution to the DIS cross-section is qualitatively different for different choices of the kinematic variables[34] and so we begin this section by discussing the kinematics of DIS and the position of the CGC in a kinematic phase space.

For the electron to interact with the target, it must radiate a photon. This is described kinematically in Figure 1.2. The four-momentum of this photon is labelled q. We consider the t-channel interaction in which q is spacelike and define the positive quantity $Q^2 = -q^2$. We only consider cases where Q^2 large enough that perturbative calculations are valid. The momentum of the target is labelled p, which allows us to calculate the invariant energy of the system using $s = p^2 + 2p.q + q^2$. The p^2 term simply gives the mass of the photon, which can be neglected at the relevant energies, leaving two terms which contribute to the invariant energy. The limits in which each of these terms dominate have distinct behaviour and different leading order diagrams describing the interaction.

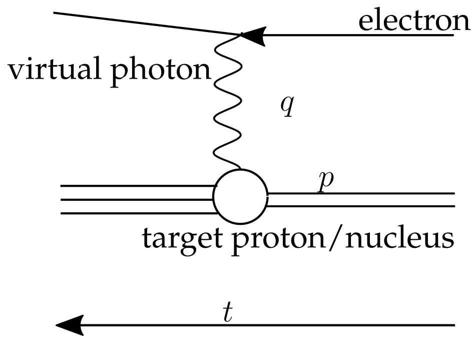


FIGURE 1.2: Diagrams are to be read with time flowing from right to left. We adopt this convention to improve the readability of equations involving diagrams. Here we present a kinematic description of a deep inelastic scattering (DIS) event: an electron radiates a virtual photon with momentum q, which interacts with a target that has momentum p. The kinematic variables $Q^2 = -q^2$ and $x_{bj} = \frac{Q^2}{2p.q}$ parametrise a phase space of possible kinematic situations.

A commonly considered situation is that in which the emitted photon is highly spacelike and $Q^2 \gg 2p.q$. The large value of Q^2 means that the probing photon resolves the photon finely and can interact with a constituent quark at leading order[36]. This situation is described with a Feynman diagram in Figure 1.3. However, in this work we consider the other limit, in which the $2p.q$ term dominates. To do so, we introduce the Bjorken-x variable, $x_{bj} = \frac{Q^2}{2p.q}$ and consider the limit in which x_{bj} is very small. Then the $2p.q$ term must dominate the invariant energy. In this situation Q^2 is relatively small (although still larger

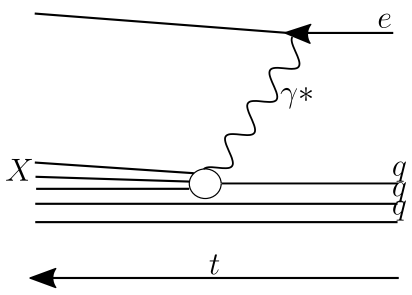


FIGURE 1.3: A projectile electron interacts with a target via deep inelastic scattering. This diagram gives the leading contribution to the cross-section when the photon has very high momentum q and the q^2 term dominates the calculation of the invariant energy. The deeply inelastic nature is evident in that the target is broken into the particle collection X.

than Λ_{QCD}, so that perturbative calculations are valid) and the probing photon has limited resolution. Instead of resolving the photon into constituent quarks, at leading order the photon produces a $q\bar{q}$ pair which radiates a gluon. This gluon interacts with the overall target field (including soft gluons), as shown in Figure 1.4 [11, 34]. In fact, the ratio between the time before the interaction during which the $q\bar{q}$ pair has been produced and the time over which the interaction itself occurs (this latter being the highlighted section of Figure 1.4) is proportional to $\frac{1}{x_{bj}}$. In the small-x_{bj} case, the $q\bar{q}$ pair is produced long before the interaction with the target. This is the situation with which we will be concerned: in particular, we will calculate the contribution of the $q\bar{q}$ dipole to the overall DIS cross-section.

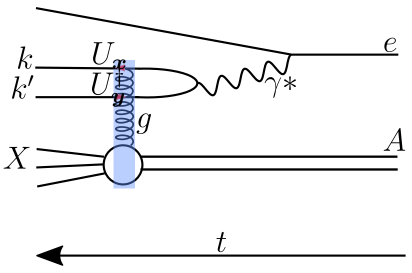


FIGURE 1.4: The leading diagram for a projectile electron interacting with a target via deep inelastic scattering at small x_{bj}. Because x_{bj} is small, the interaction is mediated by a $q\bar{q}$ pair which emits a gluon. Note that in recognition of the background field present for such a QCD interaction and in anticipation of the notation we will need to discuss the cross-section and the JIMWLK Hamiltonian, we indicate the Wilson lines with arrows on the quark and antiquark. Additionally, the interaction period is highlighted.

This kinematical situation lends itself to a phase space description, as in Figure 1.5, where the resolution of the photon probe, Q^2 and the rapidity, which depends on Bjorken-x via $Y = \ln \frac{1}{x_{bj}}$, span the space. As the rapidity (or equivalently energy) increase along one axis, more and more particles are generated. If this increasing abundance of particles is not accompanied by increasing resolution, there comes a point where the particles can be treated as overlapping: in this small-x_{bj} region is the CGC [11]. At slightly higher resolution is the extended scaling region, where the assumptions of the CGC do not strictly hold, but where CGC calculations continue to give surprisingly good fits to data[37]. It is only recent data from HERA that has introduced a tension in the extent to which Q^2 can be extended without compromising the goodness of the fit[38].

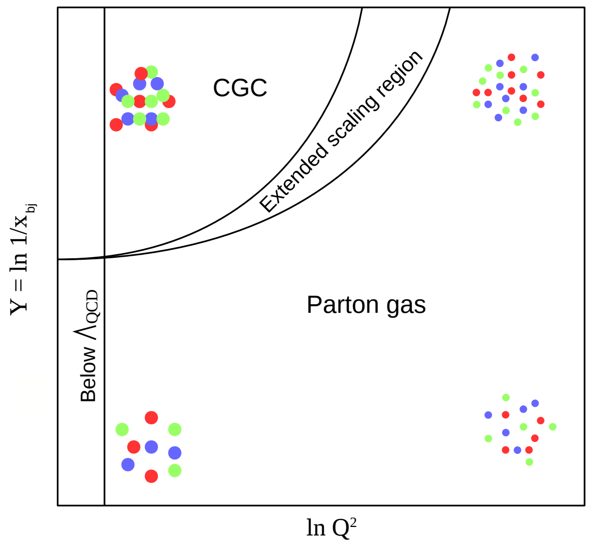


FIGURE 1.5: The kinematical variables Q^2 and x_{bj} define a phase space where parton resolution varies with Q^2 and parton density with x_{bj} At the top right, where the particles begin to overlap, is the CGC. Beyond the CGC is a region denoted as "extended scaling", which appears to have CGC-like properties, although it lies beyond the strict definition of the CGC. The diagram is inspired by [34].

The cross-section for this DIS process is given by[34]

$$\sigma_{DIS}(Y) = \int_0^1 d\alpha \int d^2r |\psi|^2(\alpha, r^2Q^2) \int d^2b \left\langle \frac{\text{tr}(1 - U_{\boldsymbol{x}}U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y + \left\langle \frac{\text{tr}(1 - U_{\boldsymbol{y}}U_{\boldsymbol{x}}^\dagger)}{N_c} \right\rangle_Y . \tag{1.2}$$

where $\alpha \in [0, 1]$ is the longitudinal momentum fraction; $r = |\boldsymbol{x} - \boldsymbol{y}|$ is the dipole separation; and Q^2 sets the transverse resolution scale. The virtual photon wavefunction is given by ψ and the impact parameter is defined by $\boldsymbol{b} = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{y}$. This will be discussed in much greater detail in the following chapter, where we will make use of diagrammatic notation to elucidate the behaviour of these objects.

1.3 Outline

Having discussed the CGC and the deep inelastic scattering process, we will go on in Chapter 2 to introduce diagrammatic notation for the DIS cross section and obtain its energy evolution by the JIMWLK Hamiltonian. We then consider JIMWLK evolution of the Wilson line 2-point function, obtaining the BK equation with some discussion of truncation methods. We also derive the scaling behaviour of the BK equation. In Chapter 3 we consider numerical methods of solving the BK equation, with a focus on the parallelisation of the algorithm and the computational techniques required for such a procedure. The structure of the code produced for this project is also described in Chapter 3. Chapter 4 presents the numerical results of the code, together with some benchmarking statistics to enable comparison of the performance of the parallel code with other implementations; and such comparisons are made. Further work and applications making use of this tool are briefly discussed and the thesis concluded in Chapter 5. The parallel code described in the thesis is included thereafter as an appendix.

Chapter 2

JIMWLK Evolution

2.1 The DIS cross-section

Before discussing the cross-section for this DIS process in terms of algebraic expressions, we consider the process in terms of diagrams. In Figure 2.1 we introduce further, more compact, diagrammatic notation, following the principles of [35, 39]. In this notation we simplify Figure 1.4 by omitting the electron and showing only the virtual photon, which produces a $q\bar{q}$ pair and a target with a colour field. Interaction between the $q\bar{q}$ pair is indicated by a highlighted strip; the special case in which there is no interaction is indicated by only a dotted line at the point of coincidence ($x^- = 0$ in lightcone coordinates). The Wilson lines are indicated, although in the case of no interaction they must be one or gauge equivalent to one; they are connected by a fermion line and we read in the case of any constant field $U^\dagger_{\boldsymbol{y}} U_{\boldsymbol{x}} = U^\dagger U = 1$ by unitarity. This is particularly simple since the $q\bar{q}$ pair is a colour singlet.

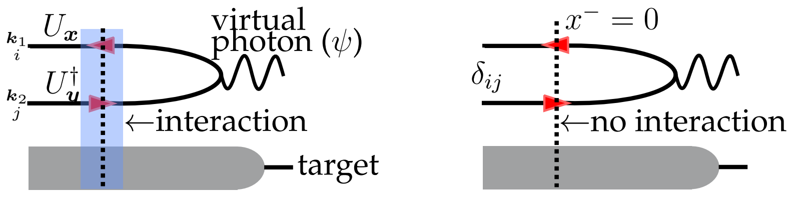
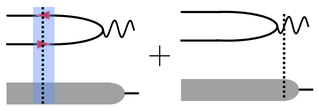
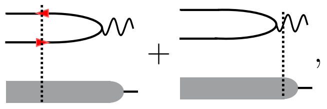


FIGURE 2.1: These diagrams give a more compact notation for the DIS process (and, by extension, other processes involving the CGC). The origin of the virtual photon is omitted from the diagram, while the target, below, is immediately represented by its colour field, which implies a rapidity-dependent averaging procedure. Wilson lines are again represented by arrows, while the mediator of the interaction is indicated only abstractly by the presence of a highlighted interaction strip. The outgoing momenta and colour indices are indicated on the free legs of the diagram. In the case where the particles pass without interaction, this is indicated (on the right) by a dotted line at the lightcone coordinate point $x^- = 0$. Although the Wilson lines are included in the diagram without interaction, they must necessarily be one in the absence of such interaction. Because of this, the colour indices i and j are simply the indices of the Kronecker delta, δ_{ij}.

For convenience we refer to the sums of all relevant interacting and non-interacting diagrams as $|\text{out}\rangle$ and $|\text{in}\rangle$ states respectively. To zeroth order (with the order indicated by the subscript) these are

$|\text{out}\rangle_0 =$  (2.1)

and

$|\text{in}\rangle_0 =$  (2.2)

where we now include the possibility that the collision occurs before the $q\bar{q}$ pair is produced, precluding QCD interactions. Note that this diagram in which the collision occurs before $q\bar{q}$ emission cancels immediately in the cross-section calculation below.

To discuss probabilities, it is necessary to consider conjugate diagrams. This is done by reversing the time direction of the diagrams and then restoring the Wilson line arrows to their original directions, resulting in Wilson lines conjugate to those in the original diagrams. This is illustrated in Figure 2.2. (For further discussion of diagrammatic manipulations see [39, 40].)

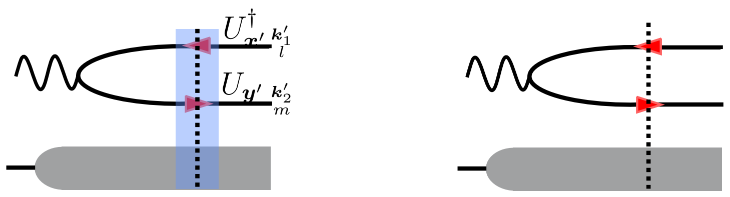
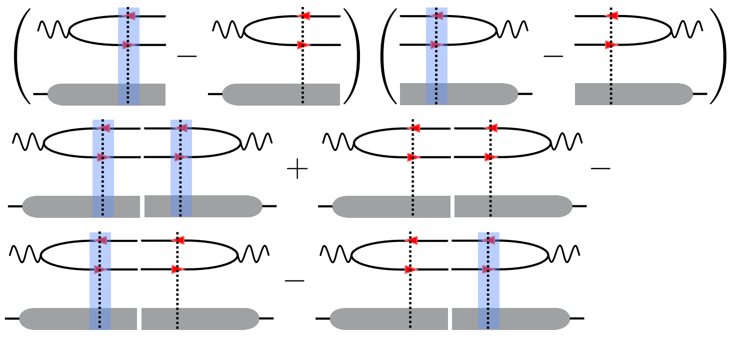


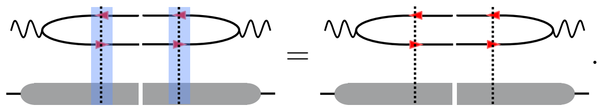
FIGURE 2.2: The conjugate of a diagram is obtained by reversing the time direction of the diagram and then restoring the Wilson line arrows to their original directions, making them conjugate to the original Wilson lines. The diagrams here are conjugate to those in Figure 2.1. Note the the coordinates and colour indices are distinct from those in Figure 2.1, but that they will be identified by the phase space integration and colour sum in the cross-section calculation.

We can now represent the cross-section diagrammatically by looking at the difference between all possible interactions, the $|\text{out}\rangle$ state, subtracting the case in which nothing happens, the $|\text{in}\rangle$ state, and multiplying by the conjugate.

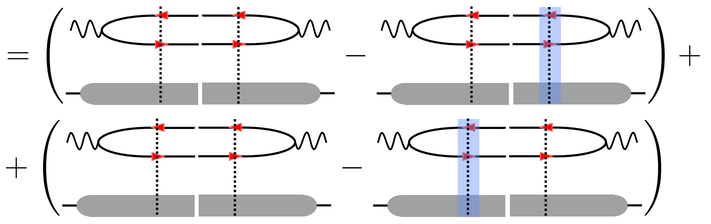
$\sigma_{DIS}(Y) =$

(2.3)

We adopt the convention that by multiplying these diagrams we imply integration over phase space, which identifies the final state coordinates, as well as summing over colour, so that we contract the Wilson lines from both diagrams. The target wave function factors (–◖ · · · ◗–) encode a path integral over possible Wilson line configurations with Y-dependent weighting. The integral over group elements is performed with a Haar measure[34, 40] and has the effect of producing a rapidity-dependent average of the target field.

We can simplify Equation 2.3 by noting that


(2.4)

The closed fermion loop gives the trace [39] $\text{tr}(U_{\boldsymbol{y}}^\dagger U_{\boldsymbol{x}} U_{\boldsymbol{x}}^\dagger U_{\boldsymbol{y}}) = \text{tr}(1)$; the diagrams are otherwise identical. Equation 2.3 becomes

$\sigma_{DIS}(Y) =$

(2.5)

Now we can write down the photon line as the $q\bar{q}$ Fock component of the virtual photon. Information about the target and the dipole is contained in the traces of Wilson lines, which are averaged in some Y-dependent fashion, to be determined by the JIMWLK (Jalilian-Marian, Iancu, McLerran, Weigert, Leonodiv and Kovner, pronounced "gym walk") equation[12–20] (or equivalently by the Balitsky hierarchy [29]) and, to leading order, by the BK equation[34]. This is described in greater detail in Section 2.2. With this in mind, we write down the dipole cross-section in the form

$$\sigma_{DIS}(Y) = \int_0^1 d\alpha \int d^2r |\psi|^2(\alpha, r^2Q^2) \int d^2b \left\langle \frac{\mathrm{tr}(1 - U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y + \left\langle \frac{\mathrm{tr}(1 - U_{\boldsymbol{y}} U_{\boldsymbol{x}}^\dagger)}{N_c} \right\rangle_Y . \tag{2.6}$$

Here $\alpha \in [0,1]$ is the longitudinal momentum fraction; $r = |\boldsymbol{x} - \boldsymbol{y}|$ is the dipole separation; and Q^2 sets the transverse resolution scale. The impact parameter is defined by $\boldsymbol{b} = \alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}$. [1]

This equation amounts to a convolution of the wavefunction (squared) with the dipole cross-section, which can be separated out as

$$\sigma_{dipole}(Y, r^2) = 2\mathrm{Re} \left(\int d^2b \left\langle \frac{\mathrm{tr}(1 - U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y \right) \tag{2.7}$$

where the energy-dependent average becomes the focus of the expression.

Noticing the role of the average above in determining whether or not the dipole interacts with the rest of the system, we define that average as $\langle T \rangle = 1 - \langle S \rangle$. $\langle S \rangle$ is the dipole correlator or Wilson line 2-point function[11]: if the $q\bar{q}$ pair is correlated, it is indistinguishable from a gluon and does not interact with the system. in this situation, $S = 1$. If the $q\bar{q}$ pair is decorrelated, the constituents interact independently with the target and we have $S = 0$ and $T = 1$. Of course, we can only calculate expectation values for S (or equivalently T), and these values will range between 0 and 1, depending on the probability the the $q\bar{q}$ pair is decorrelated. We have the following definitions:

$$T(r) = \left\langle \frac{\mathrm{tr}(1 - U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y , \tag{2.8}$$

so that

$$S(r) = 1 - \left\langle \frac{\mathrm{tr}(1 - U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y = \left\langle \frac{\mathrm{tr}(U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger)}{N_c} \right\rangle_Y . \tag{2.9}$$

We note the the situation in which the dipole separation distance is 0 is one of the special coincidence limits discussed in Section 1.1 above. In the limit $\boldsymbol{x} \to \boldsymbol{y}$ the correlator reduces to $S = 1$, since $U_{\boldsymbol{y}} U_{\boldsymbol{y}}^\dagger = 1$. On the other hand, when $\boldsymbol{x}$ and $\boldsymbol{y}$ are far apart, $U_{\boldsymbol{x}}$ and $U_{\boldsymbol{y}}^\dagger$ are unrelated, so $U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger$ averages to zero. Since the space is N_c-dimensional, the trace of 1 is N_c and the correlator is zero. This behaviour is intuitively reasonable: coincident particles are guaranteed to

[1] This definition of the impact parameter is chosen so that b is the Fourier conjugate to momentum transfer. It is parametrised over the dipole separation be α.

be correlated, while particles at infinite separation must be decorrelated. The general shape of the dipole correlator is shown in Figure 2.3.

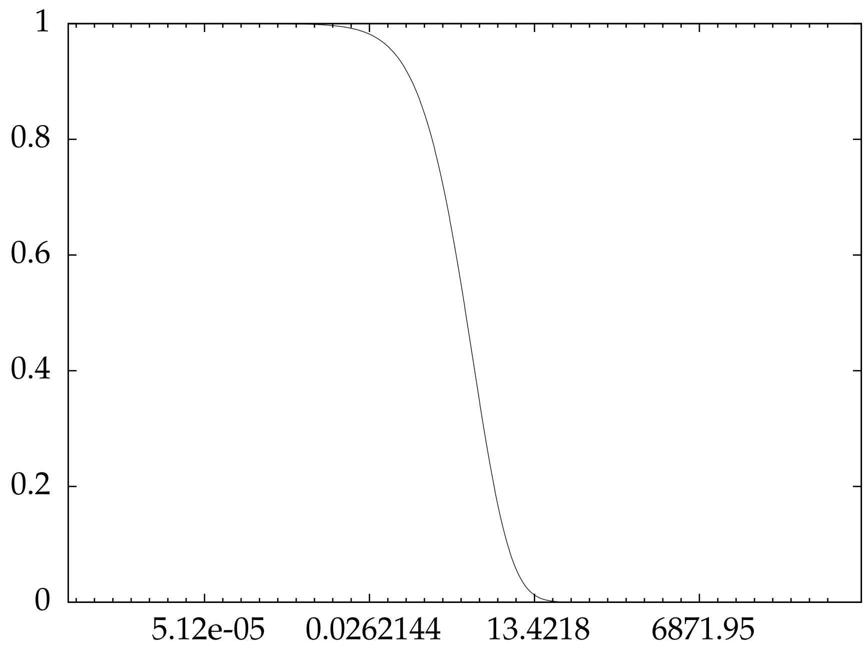


FIGURE 2.3: The dipole correlator expectation value, $\langle S(r) \rangle_Y$, decreases monotonically from 1 to 0 as r goes from 0 to ∞.

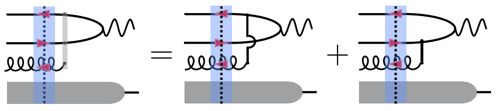
We choose from amongst several similar and equally valid definitions[2] to define the correlation length, $R_s(Y)$, as the dipole separation below which the $q\bar{q}$ pair is more likely to be correlated than decorrelated. That is, $\langle S(R_s) \rangle_Y = \frac{1}{2}$. The inverse of the correlation length is the saturation scale, $Q_s = \frac{1}{R_s}$. R_s decreases with increasing rapidity and is used to define the scaling speed when the scaling behaviour of the dipole function's evolution is quantified [41].

2.2 Energy evolution

The rapidity dependence (or equivalently energy dependence, in our context) of the dipole correlator is described by the JIMWLK equation. JIMWLK is a renormalisation group (RG) equation that tracks leading log corrections to diagrams such as Figure 1.4 [34]. These logarithmically large contributions contain factors of the form $\ln \frac{1}{x_{bj}} = Y$, which are large when x_{bj} is small and which increase with rapidity. JIMWLK thus describes the energy evolution of the Wilson line n-point functions associated with the interaction. This is expressed through

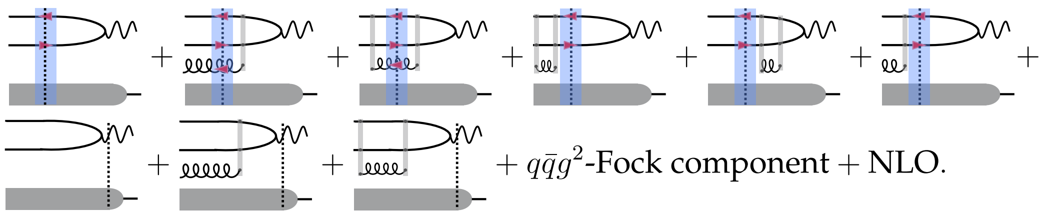
[2] of the form $\langle S(R_s) \rangle_Y = c$, where the precise value of $c \in [0, 1]$ has little impact

the action of the JIMWLK Hamiltonian. As with the DIS cross-section, before writing down the JIMWLK Hamiltonian, we will discuss its action in terms of diagrams. We do so using the fact that increasing the available energy can be expressed diagrammatically by introducing an additional gluon. Since we will always be interested in the introduction of a gluon at either the quark or the antiquark, we introduce a notation for the summation of gluon vertices, so that we can represent both of these cases in a single diagram. We also represent the adjoint Wilson line associated with the gluon as an arrow on the gluon line; if the gluon is produced in a way that precludes interaction with the target, there is no Wilson line. Then we have



(2.10)

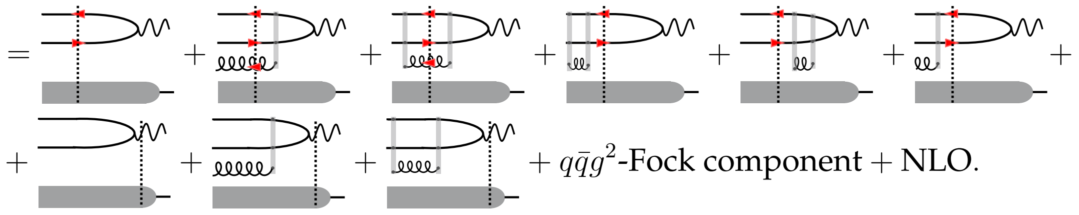
and similarly for other gluon insertions. This allows us to make a list of corrections to the $|\text{out}\rangle$ state at order $\alpha_s \ln\left(\frac{1}{x_{bj}}\right)$. Including these corrections, the $|\text{out}\rangle_1$ state is

$|\text{out}\rangle_1 =$  $+\, q\bar{q}g^2$-Fock component $+$ NLO.

(2.11)

The $q\bar{q}g^2$-Fock component is mentioned, but not explicitly drawn out, as it will not contribute to the cross-section at order α_s and thus can be neglected here.

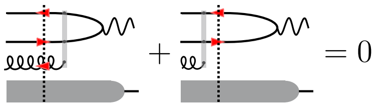
To obtain the "nothing happens" $|\text{in}\rangle$ state including corrections, we can simply consider Equation 2.11 without interactions. In cases where there could already be no interaction, this means some diagrams are unchanged. We obtain

$|\text{in}\rangle_1 =$  $+\, q\bar{q}g^2$-Fock component $+$ NLO.

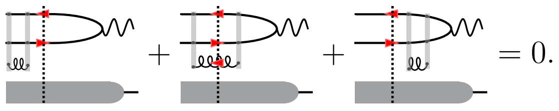
(2.12)

While we can formally write down these corrections, we should expect their contribution to be zero, since the lack of interaction is not energy-dependent.

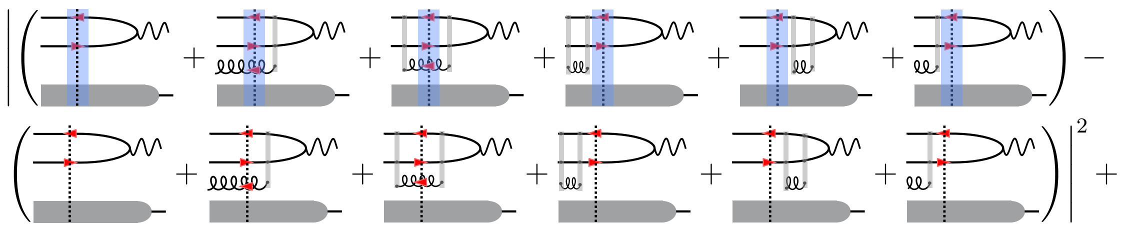
This cancellation[35] is realised through the identities


(2.13)

and


(2.14)

Thus only the first diagram on the $|\text{in}\rangle_1$ state's first line remains, while the diagrams in the second line will immediately cancel with the second line of the $|\text{out}\rangle_1$ state when we calculate the cross-section. To do so, we consider the quantity $||\text{out}\rangle_1 - |\text{in}\rangle_1|^2$ in diagrams:


+NLO. (2.15)

We note that the last five diagrams in this expression cancel amongst themselves according to Equations 2.13 and 2.14. Retaining these significantly expands the number of diagrams which we consider in the $\langle\text{in}|\text{out}\rangle$ and $\langle\text{out}|\text{in}\rangle$ states, since this cancellation removes all diagrams with three free final state coordinates. Since diagrams with different numbers of free coordinates are in different parts of the Fock space, their overlap is zero. Thus cancelling the three-coordinate $|in\rangle$ state diagrams would mean that the overlap of the $|\text{in}\rangle$ state with all three-coordinate diagrams is zero. However, since we wish to also consider other cancellation structures, we retain these terms for completeness.

As the foregoing discussion suggests, we will consider the various cross-section terms separately: the $\langle\text{in}|\text{in}\rangle$ term, the $\langle\text{out}|\text{out}\rangle$ term and the $\langle\text{in}|\text{out}\rangle$ term, with its complex conjugate $\langle\text{out}|\text{in}\rangle$. The $\langle\text{in}|\text{in}\rangle$ term is the simplest of these: it does not provide corrections, as the only apparent corrections to the $|\text{in}\rangle$ state cancel. The $\langle\text{out}|\text{out}\rangle$ state also does not provide corrections, as long as we consider the inclusive cross-section, but the cancellation bears some discussion.

Figure 2.4 gives the contributions to the $\langle\text{out}|\text{out}\rangle$ overlap in tabular form. Each of the first three rows of the table cancels through final state cancellations. These come from the fact that a $q\bar{q}g$ vertex can be moved across the final vertex at the cost of a negative sign and a symmetry factor. The symmetry factors are exactly such that the rows cancel. The last row cancels for independent reasons: each diagram in the last row only contains Wilson lines when they are directly adjacent to their conjugates. By unitarity, these diagrams cannot contribute to corrections. We observe, however, that the diagrams in the central column of Figure 2.4 only cancel because we consider an inclusive cross-section. Although one might hope for the cancellation pattern to hold in general, this is not the case. Considering exclusive production requires that the $|\text{out}\rangle$ state be filtered according to the desired event. This procedure would alter the diagrams in the central column sufficiently that these cancellation patterns would no longer hold.

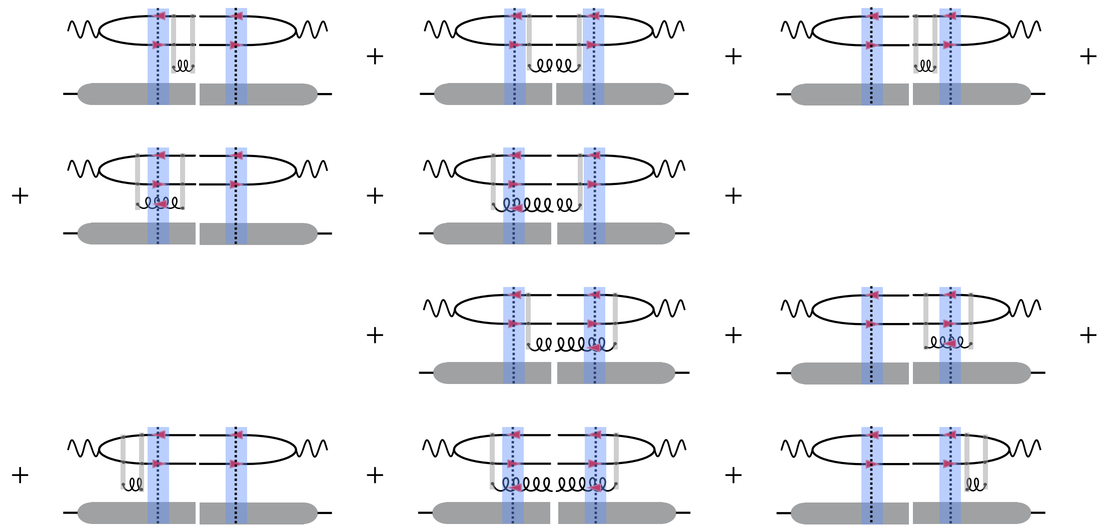


FIGURE 2.4: Contributions to the $\langle\text{out}|\text{out}\rangle$ term of the cross-section with first order corrections. Each of the first three rows cancels due to final state cancellations, while the last row cancels due to the adjacency of Wilson lines and their conjugates. Placing restrictions on the final states, as for exclusive observables would ruin the cancellation pattern of the middle column.

Having determined that neither the $\langle\text{in}|\text{in}\rangle$ nor the $\langle\text{out}|\text{out}\rangle$ term contributes to the inclusive cross-section, we turn our attention to the $\langle\text{in}|\text{out}\rangle$ term. Figure 2.5 displays the contributions in tabular form.

Since this term and its conjugate are the only ones that remain to contribute to the cross-section corrections, we expect to see terms of the type that one obtains by deriving JIMWLK through the optical theorem. These are the virtual gluon terms of the right hand column and indeed, we can cancel the first two columns. The first column cancels by Equation 2.14 while the middle column

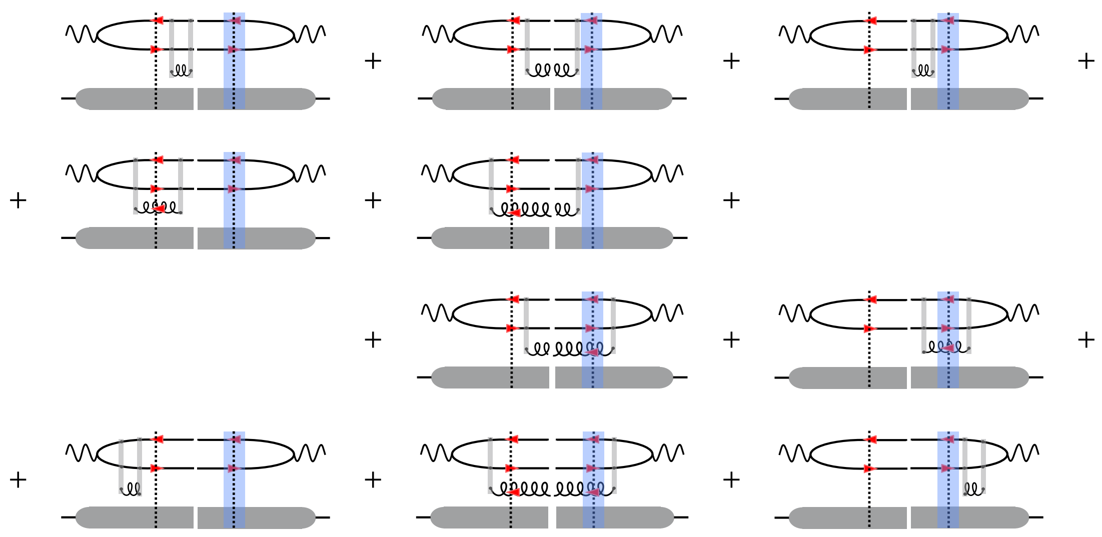
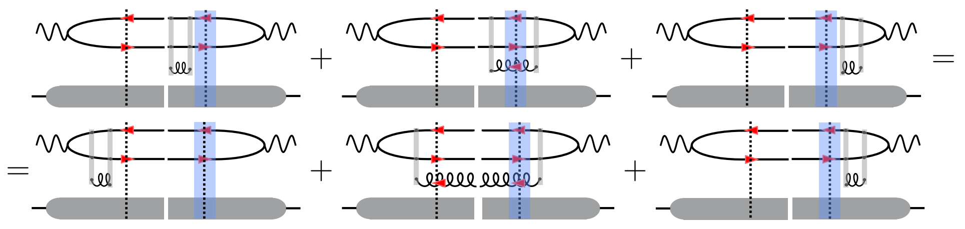


FIGURE 2.5: Contributions to the $\langle \text{in}|\text{out}\rangle$ term of the cross-section with first order corrections. The first two columns each cancel, leaving the diagrams that correspond to the virtual corrections found when deriving JIMWLK via the optical theorem. However, the top three rows also cancel, so that the bottom row must be equivalent to the right column.

cancels by application of Equation 2.13 to the upper pair and lower pair of diagrams independently. Then our corrections consist of the virtual gluon diagrams in the right column of Figure 2.5, together with their complex conjugates from the $\langle \text{out}|\text{in}\rangle$ term.

However, we can also consider final state cancellations, as we did for the $\langle \text{out}|\text{out}\rangle$ term. If we do this, the first three rows cancel, leaving only the bottom row. The bottom row is therefore required by consistency to be equal to the right column. This allows us to alternatively interpret JIMWLK evolution as consisting of gluon emission to the final state balanced by virtual gluon emission, since we have shown that it is equivalent to the virtual gluon corrections expression. We can write this equality as


(2.16)

Having this diagrammatic understanding of the higher-energy corrections, we now write down the JIMWLK Hamiltonian at leading order in $\alpha_s \ln\left(\frac{1}{x_{bj}}\right)$.

It is[34]

$$H_{JIMWLK} = -\frac{\alpha_s}{2\pi^2}\frac{(\boldsymbol{x}-\boldsymbol{z})\cdot(\boldsymbol{z}-\boldsymbol{y})}{(\boldsymbol{x}-\boldsymbol{z})^2(\boldsymbol{z}-\boldsymbol{y})^2}\left\{i\nabla_x^a i\nabla_y^a + i\bar{\nabla}_x^a i\bar{\nabla}_y^a + \tilde{U}_z^{ab}\left(i\bar{\nabla}_x^a i\nabla_y^b + i\nabla_x^a i\bar{\nabla}_y^b\right)\right\} \tag{2.17}$$

with the convention that repetition of x, y, z indices represents integration over the repeated index. $\nabla_{\boldsymbol{x}}^a$ is a functional derivative operator defined by

$$i\nabla_{\boldsymbol{x}}^a = [U_{\boldsymbol{x}}t^a]_{ij}\frac{\delta}{\delta U_{\boldsymbol{x},ij}} \tag{2.18}$$

where δ is a normal functional derivative that acts with respect to the components of the Wilson line field. $\bar{\nabla}_{\boldsymbol{x}}^a$ is defined similarly by

$$i\bar{\nabla}_{\boldsymbol{x}}^a = -[t^a U_{\boldsymbol{x}}]_{ij}\frac{\delta}{\delta U_{\boldsymbol{x},ij}}. \tag{2.19}$$

From unitarity we know that $U_{\boldsymbol{x}}U_{\boldsymbol{x}}^\dagger = 1$ and so these functional differential operators must act not only on Wilson lines, but also on their conjugates. Their actions are:

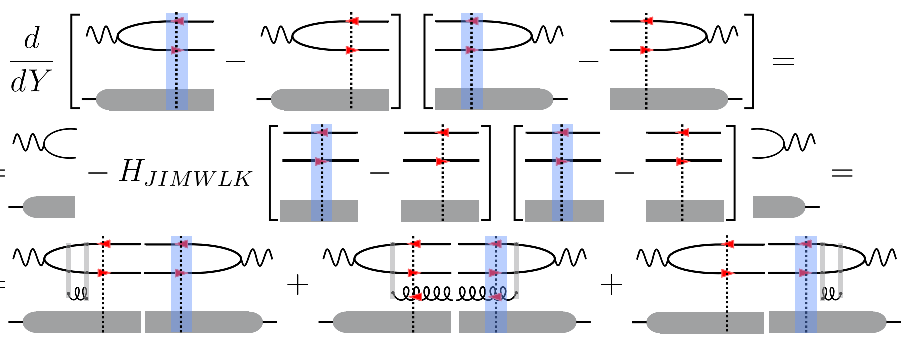
$$i\nabla_{\boldsymbol{x}}^a U_{\boldsymbol{y}} = -U_{\boldsymbol{x}}t^a\delta_{\boldsymbol{xy}}, \tag{2.20}$$

$$i\nabla_{\boldsymbol{x}}^a U_{\boldsymbol{y}}^\dagger = t^a U_{\boldsymbol{x}}^\dagger\delta_{\boldsymbol{xy}}, \tag{2.21}$$

$$i\bar{\nabla}_{\boldsymbol{x}}^a U_{\boldsymbol{y}} = t^a U_{\boldsymbol{x}}\delta_{\boldsymbol{xy}}, \tag{2.22}$$

$$i\bar{\nabla}_{\boldsymbol{x}}^a U_{\boldsymbol{y}}^\dagger = -U_{\boldsymbol{x}}^\dagger t^a\delta_{\boldsymbol{xy}}. \tag{2.23}$$

We see that the operator essentially acts by seeking out Wilson lines and introducing a t^a factor, which corresponds to a gluon insertion. The first two terms in the curly braces of Equation 2.17 act to introduce gluons away from the interaction region, without adjoint Wilson lines, while the latter terms introduce gluons that participate in the interaction, along with the requisite adjoint Wilson lines. This corresponds to the introduction of gluons that has been introduced diagrammatically. We can write this in a diagrammatic sense as



(2.24)

where the last line can be replaced according to Equation 2.16 as desired. The effect of the JIMWLK Hamiltonian is thus, by construction, to produce the corrected expression for the cross-section, as we have derived it diagrammatically.

2.3 The BK equation

Since Equations 2.5 and 2.6 for the dipole cross-section contain the dipole function $\hat{S}_{\boldsymbol{xy}} = \frac{1}{N_c}\mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{y}})$, we can separate out the JIMWLK evolution of just this component, to obtain a dipole evolution equation. This equation is[34]

$$\begin{aligned}&\frac{d}{dY}\langle\mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{y}})\rangle_Y = -H_{JIMWLK}\langle\mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{y}})\rangle_Y = \\ =&\frac{\alpha_s}{\pi^2}\int d^2z\mathcal{K}_{\boldsymbol{xzy}}\left(\langle[\tilde{U}_{\boldsymbol{z}}]^{ab}\mathrm{tr}(t^aU_{\boldsymbol{x}}t^bU^\dagger_{\boldsymbol{y}})\rangle_Y - C_f\langle\mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{y}})\rangle_Y\right).\end{aligned} \tag{2.25}$$

At leading order the kernel is

$$\mathcal{K}_{\boldsymbol{xzy}} = \frac{(\boldsymbol{x}-\boldsymbol{y})^2}{(\boldsymbol{x}-\boldsymbol{z})^2(\boldsymbol{z}-\boldsymbol{y})^2}. \tag{2.26}$$

To arrive at the BK equation, we use the (normalised) definition of the dipole correlator, $S_{\boldsymbol{xy}}$, from Equation 2.9, together with the Fierz identity, which is

$$[\tilde{U}_{\boldsymbol{z}}]^{ab}2\mathrm{tr}(t^aU_{\boldsymbol{x}}t^bU^\dagger_{\boldsymbol{y}}) = \mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{z}})\mathrm{tr}(U_{\boldsymbol{z}}U^\dagger_{\boldsymbol{y}}) - \frac{1}{N_c}\mathrm{tr}(U_{\boldsymbol{x}}U^\dagger_{\boldsymbol{y}}). \tag{2.27}$$

For our purposes we rewrite the Fierz identity as

$$[\tilde{U}_{\boldsymbol{z}}]^{ab}\mathrm{tr}(t^aU_{\boldsymbol{x}}t^bU^\dagger_{\boldsymbol{y}}) = \frac{1}{2}\left(N_c^2\hat{S}_{\boldsymbol{xz}}\hat{S}_{\boldsymbol{zy}} - \hat{S}_{\boldsymbol{xy}}\right). \tag{2.28}$$

Then substituting into Equation 2.25 and using $C_f = (N_c^2 - 1)/2N_c$, we arrive at

$$N_c \frac{d}{dY} \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y = \frac{\alpha_s}{2\pi^2} \int d^2z \mathcal{K}_{\boldsymbol{xzy}} \left(N_c^2 \langle \hat{S}_{\boldsymbol{xz}} \hat{S}_{\boldsymbol{zy}} \rangle_Y - \left(N_c^2 - 1 + 1\right) \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y \right). \tag{2.29}$$

Dividing through by N_c produces

$$\frac{d}{dY} \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y = \frac{\alpha_s N_c}{2\pi^2} \int d^2z \mathcal{K}_{\boldsymbol{xzy}} \left\langle \hat{S}_{\boldsymbol{xz}} \hat{S}_{\boldsymbol{zy}} \right\rangle_Y - \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y. \tag{2.30}$$

We note immediately that the 2-coordinate function, $\hat{S}_{\boldsymbol{xy}}$ involving $\boldsymbol{x}$ and $\boldsymbol{y}$ depends on a function of three coordinates: $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$. In turn the 3-point function depends on the four point function and so on in an infinite hierarchy. To make progress we will have to somehow truncate the hierarchy. The simplest, although not necessarily the most elegant, solution is to simply factorise the average (the so-called large-N_c approximation), so that the dependence is instead over a set of 2-point functions. This produces the BK equation[34]:

$$\frac{d}{dY} \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y = \frac{\alpha_s N_c}{2\pi^2} \int d^2z \frac{(\boldsymbol{x}-\boldsymbol{y})^2}{(\boldsymbol{x}-\boldsymbol{z})^2(\boldsymbol{z}-\boldsymbol{y})^2} \left(\left\langle \hat{S}_{\boldsymbol{xz}} \right\rangle_Y \left\langle \hat{S}_{\boldsymbol{zy}} \right\rangle_Y - \left\langle \hat{S}_{\boldsymbol{xy}} \right\rangle_Y \right). \tag{2.31}$$

This is now a solvable equation, although it does not respect all the original symmetries of the system. Particularly, the coincidence limits (as discussed in Section 1.1) are not preserved by the transformation. One might therefore wish to consider alternative truncations which do preserve these symmetries. The Gaussian Truncation (GT) has this property. We therefore also consider the result of treating Equation 2.25 with the GT. To do so, we note the following results for a 2-point function $\mathcal{G}_{Y,\boldsymbol{xy}}$ involving multi-U-correlators within the GT[9, 10, 35]:

$$\text{tr}(\langle U_{\boldsymbol{x}} U_{\boldsymbol{y}}^\dagger \rangle_Y) = N_c e^{-C_f \mathcal{G}_{Y,\boldsymbol{xy}}} \tag{2.32}$$

and

$$[\tilde{U}_{\boldsymbol{z}}]^{ab} \text{tr}(t^a U_{\boldsymbol{x}} t^b U_{\boldsymbol{y}}^\dagger) = N_c C_f e^{-\frac{N_c}{2}(\mathcal{G}_{Y,\boldsymbol{xz}} + \mathcal{G}_{Y,\boldsymbol{zy}} - \mathcal{G}_{Y,\boldsymbol{xy}}) - C_f \mathcal{G}_{Y,\boldsymbol{xy}}}. \tag{2.33}$$

Inserting these into Equation 2.25 gives

$$N_c \frac{d}{dY} e^{-C_f G_{Y,\boldsymbol{xy}}} = \frac{\alpha_s N_c C_f}{\pi^2} \int d^2z \mathcal{K}_{\boldsymbol{xzy}} \left(e^{-\frac{N_c}{2}\left(G_{Y,\boldsymbol{xz}} + G_{Y,\boldsymbol{zy}} - G_{Y,\boldsymbol{xy}}\right)} \cdot e^{-C_f G_{Y,\boldsymbol{xy}}} - e^{-C_f G_{Y,\boldsymbol{xy}}} \right). \tag{2.34}$$

After rewriting the derivative on the left hand side as

$$\frac{d}{dY}e^{-C_f G_{Y,\boldsymbol{xy}}} = -C_f e^{-C_f G_{Y,\boldsymbol{xy}}}\frac{d}{dY}G_{Y,\boldsymbol{xy}}, \tag{2.35}$$

we can divide the equation through by $-N_c C_f e^{-C_f G_{Y,\boldsymbol{xy}}}$ to obtain

$$\frac{d}{dY}\mathcal{G}_{Y,\boldsymbol{xy}} = \frac{\alpha_s}{\pi^2}\int d^2z \mathcal{K}_{\boldsymbol{xyz}}\left(1 - e^{-\frac{N_c}{2}\left(\mathcal{G}_{Y,\boldsymbol{xz}}+\mathcal{G}_{Y,\boldsymbol{yz}}-\mathcal{G}_{Y,\boldsymbol{xy}}\right)}\right). \tag{2.36}$$

At first glance, this is quite different from Equation 2.31, but there is a surprising relationship between $\mathcal{G}_{Y,\boldsymbol{xy}}$ and $\hat{S}_{\boldsymbol{xy}}(Y)$, which we can derive by multiplying through by $e^{-N_c G_{Y,\boldsymbol{xy}}}$, finding

$$\frac{d}{dY}e^{-\frac{N_c}{2}G_{Y,\boldsymbol{xy}}} = \frac{\alpha_s N_c}{2\pi^2}\int d^2z \mathcal{K}_{\boldsymbol{xzy}}\left(e^{-\frac{N_c}{2}G_{Y,\boldsymbol{xz}}}\cdot e^{-\frac{N_c}{2}G_{Y,\boldsymbol{zy}}}\cdot e^{\left(\frac{N_c}{2}-\frac{N_c}{2}\right)G_{Y,\boldsymbol{xy}}} - e^{-\frac{N_c}{2}G_{Y,\boldsymbol{xy}}}\right) \tag{2.37}$$

If we now define $S^G_{\boldsymbol{xy}} = e^{-\frac{N_c}{2}G_{Y,\boldsymbol{xy}}}$, we can write

$$\frac{d}{dY}S^G_{\boldsymbol{xy}} = \frac{\alpha_s N_c}{2\pi^2}\int d^2z \mathcal{K}_{\boldsymbol{xzy}}\left(S^G_{\boldsymbol{xz}}S^G_{\boldsymbol{zy}} - S^G_{\boldsymbol{xy}}\right) \tag{2.38}$$

which has exactly the same form as Equation 2.31. Since we have divided through by $e^{-C_f G_{Y,\boldsymbol{xy}}}$ and then multiplied through by $e^{-\frac{N_c}{2}G_{Y,\boldsymbol{xy}}}$, the procedure has been considered as letting $C_f = \frac{N_c^2-1}{2N_c} \to \frac{N_c}{2}$. The by-hand factorisation procedure can thus said to be related to the Gaussian truncation by taking the large-N_c limit in the exponent.

Solutions to the BK equation can thus be transformed to give $G_{Y,\boldsymbol{xy}}$ in the large-N_c limit, yielding improved fits to data. Because this relationship is so straightforward, we make use of the by-hand factorisation of the BK equation, which has a computationally simple form, for the calculational part of this work.

2.3.1 Divergences

Upon a brief inspection, it appears that the integral in equation 2.31 is not well defined, as the denominator $(\boldsymbol{x}-\boldsymbol{z})^2(\boldsymbol{z}-\boldsymbol{y})^2$ is 0 at both $\boldsymbol{z}=\boldsymbol{y}$ and $\boldsymbol{z}=\boldsymbol{x}$. On closer inspection one can show that the infinite parts of the integral cancel in the sense of the Cauchy principle value of the integral. The result is thus finite and usable: we demonstrate this below by considering the Taylor expansion of the integrand.

In this discussion and subsequently, we write $\langle S_{\boldsymbol{xy}} \rangle_Y$ simply as $S_{\boldsymbol{xy}}$, leaving the averaging implicit. Since we have assumed factorisation in the derivation of the BK equation, we no longer need to indicate at what point the averaging procedure is performed, so that this notation is not ambiguous, if it is understood that the $S_{\boldsymbol{xy}}$ represents only the expectation value of the Wilson line 2-point function.

The integrand which we wish to expand is then (including only those terms which cannot be factored out of the integral),

$$I = \frac{S_{\boldsymbol{xz}} S_{\boldsymbol{zy}} - S_{\boldsymbol{xy}}}{(\boldsymbol{x} - \boldsymbol{z})^2 (\boldsymbol{z} - \boldsymbol{y})^2}. \tag{2.39}$$

Since this is exactly symmetrical under exchange of $\boldsymbol{x}$ and $\boldsymbol{y}$, we need only demonstrate the behaviour at one of these points. We choose to do so at $\boldsymbol{y}$ and furthermore choose our coordinate system such that $\boldsymbol{y} = 0$. In this coordinate system, writing the dipole-separation dependence explicitly, we have

$$I = \frac{S(|\boldsymbol{x} - \boldsymbol{z}|) S(|\boldsymbol{z}|) - S(|\boldsymbol{x}|)}{(\boldsymbol{x} - \boldsymbol{z})^2 (\boldsymbol{z})^2}. \tag{2.40}$$

We can also transform into polar coordinates, replacing the two-dimensional vector $\boldsymbol{z}$ with a radial coordinate $z = |\boldsymbol{z}|$ and an angular coordinate θ. We choose the rotation of our coordinate system so that θ is the angle between $\boldsymbol{x}$ and $\boldsymbol{z}$. The transformation introduces a Jacobian factor of z, so that the integrand in the new coordinate system is

$$I' = \frac{S(\sqrt{x^2 + z^2 - 2xz\cos\theta}) S(z) - S(x)}{(x^2 + z^2 - 2xz\cos\theta)\,(z)}. \tag{2.41}$$

If we Taylor expand the numerator of I' around $z = 0$, we know that each term in the expansion will carry a factor of z^n. From the $n = 2$ term, the integrand is clearly finite. For $n = 1$, there is a factor of the form $\frac{z}{z}$. The integrand is thus not defined at the origin; however the limit as $z \to 0$ is defined and finite. Since the value at a single point will not alter the result of the integral, the denominator will not cause the integral to diverge. All that remains to be shown is that the $n = 0$ term is well behaved. This term is

$$S(x)S(0) - S(x)$$

, but recalling that $S(0) = 1$, this term must be zero and will not cause the integral to diverge.

From the symmetry of equation 2.31, we can handle the $z = x$ case in exactly the same way. Although the integrand is not defined at the $z = x$ point, the limit is well behaved and therefore the integral will not diverge.

2.3.2 Scaling of the BK equation

A peculiarity of the BK equation is its scaling behaviour: when the coupling, α_s, is fixed, dipole evolution eventually reaches a point where each evolution step is simply a logarithmic rescaling of the dipole function. This allows us to consider the dipole function without knowledge of initial conditions, as once the function reaches this scaling state, it no longer shows dependence on its initial condition. It is also useful as a test of any numerical simulation of dipole evolution. This scaling behaviour is in part a consequence of the saturation features of the CGC, although also on parameters such as the running (or lack thereof) of the coupling constant. We will demonstrate this scaling behaviour numerically in subsequent chapters, but before commencing numerical work, it is possible to determine analytically some of the properties of the scaling solution. We do so here, following the argument of [41]. In the process we define the scaling speed, λ and derive some of its properties. We expect that the scaling speed determined numerically must share these properties and will demonstrate that this is the case in Chapter 4.

We will assume for the remainder of this section that $S_{\boldsymbol{xy}}(Y)$ is a scaling solution of the BK equation in order to determine the properties of such a solution. If $S_{\boldsymbol{xy}}$ scales, it does not depend on the dipole separation r and rapidity Y independently, but rather on their combination. It is thus useful to define $\xi := \ln \frac{1}{r^2 Q_s^2(Y)} = \ln \frac{R_s^2(Y)}{r^2}$. All of the rapidity dependence is contained in this scaling of ξ, so that we can write $S(r, Y) = S(\xi)$.

Before dealing with the dipole function and the BK equation directly, we will prove two identities for arbitrary functions of ξ. Firstly, consider the Y-derivative of such a function:

$$\begin{aligned} \frac{\partial}{\partial Y} f(\xi) &= f'(\xi) \frac{\partial}{\partial Y} \left(ln \frac{1}{r^2 Q_s^2} \right) \\ &= -f'(\xi) \frac{\partial}{\partial Y} \left(\ln Q_s^2 \right) . \end{aligned} \tag{2.42}$$

We also calculate the r^2 derivative of an arbitrary function of ξ:

$$\begin{aligned}\frac{\partial}{\partial r^2} f(\xi) &= f'(\xi)\frac{\partial}{\partial r^2}\left(\ln\frac{1}{r^2 Q_S^2}\right)\\ &= -f'(\xi)\frac{1}{r^2 Q_s^2} Q_s^2 \frac{\partial r^2}{\partial r^2}\\ &= -\frac{1}{r^2} f'(\xi). \end{aligned} \tag{2.43}$$

Now consider the LHS of the BK equation (equation 2.31). We divide through by r^2 and the integrate over all possible dipole separations to obtain

$$\int d^2r \frac{1}{r^2}\frac{\partial S(r)}{\partial Y} = \int d^2r \frac{1}{r^2} S'(r) \frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right) \tag{2.44}$$

using the identity of equation 2.42. Then we can use equation 2.43 to substitute for $\frac{1}{r^2}S'(r)$, since we are supposing that S is really a function of ξ.

$$\int d^2r \frac{1}{r^2}\frac{\partial S(r)}{\partial Y} = -\frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right) \int d^2r \frac{\partial}{\partial r^2} S(r). \tag{2.45}$$

We can now transform the integration variables to r^2 and θ and apply the fundamental theorem of calculus (with $S(\infty) = 0$ and $S(0) = 1$) to find

$$\int d^2r \frac{1}{r^2}\frac{\partial S(r)}{\partial Y} = \pi \frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right). \tag{2.46}$$

If we divide the RHS of equation 2.31 by r^2 and the integrate over all possible dipole separations, the result must be equal to what we have just calculated. We therefore have

$$\pi\frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right) = -\frac{N_c\alpha_s}{2\pi^2}\int d^2r d^2z \frac{1}{(r-z)^2 z^2}(S_Y(r-z)S_Y(z) - S_Y(r)). \tag{2.47}$$

But since $S_Y(r)$ is a scaling solution, integrating out the spatial dependence, as in equation 2.47 above, must leave a Y-independent result. Then $\frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right)$ is equal to a constant in Y. This allows us to write

$$\frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right) = \alpha_s C =: \lambda \tag{2.48}$$

where C is given by

$$C = -\frac{N_c}{2\pi^3}\int d^2r d^2z \frac{1}{(r-z)^2 z^2}\left(S_Y(r-z)S_Y(z) - S_Y(r)\right). \tag{2.49}$$

Solving equation 2.48 for Q_s^2 gives

$$Q_s^2 = \Lambda^2 e^{\alpha_s C Y}. \tag{2.50}$$

The constant Λ is fixed by the initial condition of equation 2.48 and typically $\Lambda \approx \Lambda_{QCD}$. Thus we see that for the logarithmic scaling described by dependence through ξ, Q_s^2 must grow exponentially. We also see that the quantity $\lambda = \frac{\partial}{\partial Y}\left(\ln Q_s^2(Y)\right)$ becomes constant when the solution scales exactly. This value is known as the scaling speed and does not depend on the initial conditions. We will calculate the value of this constant in the numerical work to follow.

While useful, this discussion of scaling behaviour holds strictly only for fixed coupling. Allowing the coupling to vary with the energy scale is a phenomenological method of including higher-order effects [41] and slows the compression of the dipole function. The slowing of the compression disrupts scaling behaviour slightly, although regions of near-scaling (or pseudoscaling) and rapid shape change can still be distinguished, allowing a similar qualitative treatment in some respects. Computationally, the rapid dipole evolution of the fixed coupling case is a greater challenge, as it requires that dipole evolution take place over much larger scales before evolution can be considered to be in the scaling region. This is the test case for which the code in this project is written, forcing us to develop efficient code if we are to maintain a reasonable degree of accuracy without using excessive computation time. The extension to running coupling would be fairly straightforward and will significantly improve the performance, as the detail of the dipole function does not need to be considered over such varying scales in this case.

Chapter 3

Numerical methods

This chapter documents the structure and properties of the parallel code developed to quickly and accurately evolve Wilson line 2-point function in rapidity using the BK equation. A number of flow charts detailing the structure of the code are included with the discussion in this chapter; the code itself may be found in Appendix A. The chapter begins with a discussion of NVIDIA CUDA and parallel programming in general, so that the later discussion of the code's structure can be made in this context.

3.1 Parallel development and CUDA C

3.1.1 GPUs and parallel development

Traditional CPU design is based on the idea of serial processing. Instructions are executed one after another in sequence. Any instruction may or may not depend on the previous instructions, but in code development, the developer can be certain that when a given line of code executes, all previous lines have already finished executing. This is often useful: for example, an ODE stepper cannot calculate the second step in evolution until the first evolution step has been executed. Attempting to perform the steps out of order would produce nonsensical results. However, not all problems are inherently serial in nature. For example, numerical evaluation of an integral requires that the integrand be calculated at a number of grid locations. Each of these calculations is independent of the others and the calculations can be performed in any order. In fact, they might even be performed simultaneously, or in parallel. Non-traditional processor design, where many less powerful processors execute instructions in parallel can be extremely useful in these cases.

Sophisticated computer graphics area near-ubiquitous and lucrative example of a parallelisable computer process. Effects on any given pixel typically do not involve physically distant pixels. This means that such effects can be calculated on many pixels simultaneously. Massively parallel processors, called Graphical Processing Units (GPUs) were developed to perform these operations. However GPUs are not inherently restricted to use in graphical contexts. GPU developer NVIDIA has produced CUDA, an extension to the C programming language which allows the developer to write hybrid code, using the serial nature of CPU-executed code as a base from which to write parallel code which is then executed on an NVIDIA GPU. In this project, CUDA C is used to produce code that evolves the BK equation through rapidity with dramatically improved speed/accuracy performance.

Although parallel implementations have many advantages, they are not always better than serial implementations, even when the process in question is not inherently serial. This is in part due to the fact that the many processors of a GPU are individually less powerful than the typical CPU: GPUs are designed to handle huge numbers of small jobs simultaneously. It is also a result of overheads. Copying data from the host to the device and vice versa is a time-expensive process. If there is not a large gain to be made from parallelising the process, the costs in copying time are likely to outweigh the benefit of decreased computation time. It is therefore necessary to be selective in the processes chosen for parallelisation and to implement parallelised code in a fashion that minimises the data to be copied.

The code in this work is written using CUDA C, which allows a hybrid of serial and parellelised implementation. The main function of the C program is run on the traditional CPU – called the host – while special kernel functions specify code to be executed in parallel on the GPU. Time lost to overheads is therefore primarily in the tranition between serial and parallel instructions, while commnication between parallel threads is relatively fast, as all threads execute on the GPU and can even make use of shared memory.

3.1.2 Race conditions

When a number of threads are executed simultaneously from the GPU kernel, there is no guarantee that they will finish execution at exactly the same time or in any particular order. Given that each thread will perform a slightly different calculation and that timing is to some extent controlled at the driver / operating system level, the order in which various threads reach a given point can be considered to be random from the perspective of the developer. As long as the execution of the threads is entirely independent, this ordering will have no effect. However, it is frequently useful to include some minor thread interaction in the code. For example, if data is stored in shared memory, rather than per-thread (usually to improve access speeds), no thread should attempt to use that data until it has actually been loaded into shared memory. Situations in which the order of thread execution becomes relevant give rise to *race conditions*, which must be resolved by the developer to ensure that the results of the final code are deterministic[42]. There are two tools in particular which are relevant to this work in the context of race conditions.

The situation described above, where variables in shared memory must not be accessed until they have been assigned is resolved using the CUDA command `__syncthreads()`. Threads which each this command in the code to be executed are forced to halt until all threads have reached the `__syncthreads()` commands, whereupon execution resumes[42]. This results in a bottleneck, which may or may not be insignificant compared to the speed increase gained from using shared memory. When a small amount of frequently used data is stored in shared memory, the advantage of the speed up is usually more significant than the bottlenecking effect.

A race condition arises when multiple threads need to update the same variable. In this project, integrals are calculated over many threads, which each thread adding its contribution to the final integral result. The addition process consists of reading the initial result, adding some value to it, and replacing the initial result with the new result. If two threads that need to add to the result variable must therefore do so sequentially, so that each thread adds to a result including the previous thread's contribution and no contributions are overwritten. This process is known as atomic addition. It is described in flowchart form as part of Figure 3.2.

3.2 Overall algorithm and structure

At its heart, the problem of finding solutions to the BK equation is simply the problem of stepping through the evolution of a differential equation; here this is done very simply using the Euler forward step method. However, the equation has some features that require a little more thought, especially in regards to their impact on the computational time complexity of the problem. In this section we will discuss these features and the structure of a program that not only accommodates the additional features, but attempts to compensate for the time-complexity which they introduce. As we will see in Chapter 4, these compensation result in an order of magnitude performance improvement over the type of code that is often considered the norm for this problem.

The first complication that we encounter is that Equation 2.31 does not evolve a single variable, but a function. In order to evolve thus, we discretise the function $S(Y, r)$ along the r axis so that we have n variables $S(Y, r_i)$ whose Y-evolution is described by the system of n coupled equations obtained by discretising Equation 2.31 in r. The full solution $S(Y, r)$ can be regained to sufficient accuracy by interpolating between the $S(Y, r_i)$s, assuming that the initial discretisation is properly suited to the system. The problem of choosing a method of discretisation is discussed in Section 3.3.1. Beyond the choice of a discrete grid, this process is quite straightforward; however we take note that is introduces a factor of n to the time complexity of the problem.

The interpolation necessitated by the discretisation along the r axis is now a feature required by the process solving the BK equation. Interpolation is not only an in-principle consideration for regaining the continuous form of S after evolution. Instead, it is required for almost every integral in the evolution as in general the term $S(Y, \sqrt{r^2 + z^2 - 2rz\cos\theta})$ is not a term that is explicitly stored after discretisation. In theory more sophisticated interpolation routines may allow greater coarseness of discretisation. In practise the scaling behaviour of the solution means that the more immediate danger is that the function evolves into an interval where extrapolation would be required. In guarding against this, this discretisation is automatically such that linear interpolation suffices; although future work might investigate means by which more sophisticated interpolation and discretisation techniques might reduce the size of the problem, we do not expect such effects to be significant.

The evolution is additionally complicated by the fact that the right hand side of Equation 2.31 is not a particularly simple function. There is a two dimensional integral that must be evaluated for each of the coupled equations, which moreover, per Section 2.3.1, has an undefined integrand at certain points. This requires some care in the selection of points to be evaluated numerically.

Both the coupled evolution of a large number of equations and the existence of complex integrals within the equation make this problem a good candidate for parallelisation to reduce running time. As the incremental change $dS(r_i)$ that each dipole value experiences at each time step is independent of every other $dS(r_j)$, there is no reason (beyond processor capabilities) to calculate these values sequentially. As discussed in Section 3.1, most integrals are inherently amenable to parallelisation. We therefore may consider *a priori* that BK evolution is a good candidate for parallelisation and expect significant performance gains from a parallel implementation.

While these processes are not the only parts of the code that could in principle be parallelised, they are the only parts for which we expect significant gains from parallelisation and therefore the only parts of the code which are parallelised in this implementation. This is because of the computation-copy time tradeoff discussed in Section 3.1 above. If we lose as much time transferring data between the host and device as we gain from the parallel implementation, we would do better to turn our attention to other work. Therefore only the computation of the $dS(r_i)$s is carried out on the GPU device; this is, after all, the bulk of the work.

The overall structure of the code, focussing on the serial portions thereof, is described in Figures 3.1 and 3.2. These flowcharts describe the code executed on the host in detail: this is primarily the initialisation stage of the code and various looping structures. The Y-evolution updates are of course required to be sequential and could not be parallelised. Similarly the output processes are not at all suitable to parallelisation. The rest of the code executed sequentially and described in these flowcharts might in principle be to some degree parallelised, but the gains would not outweigh the losses or would be marginal.

The parallel integral calculation procedure referenced in Figure 3.1 is described in further detail in Figure 3.3. The integrand calculation referenced in that flowchart is in turn described in Figure 3.4. As described above, the interpolation of the dipole function is done through simple linear interpolation. The updates of the correlation length $R_s(Y)$ are done similarly. Since the correlation length is defined by $S(Y, R_s) = \frac{1}{2}$, R_s can be calculated by a kind of 'reverse interpolation'. The points at which $S(Y, r)$ is just greater and just less than $\frac{1}{2}$ are found, and the point at which it would be the desired $\frac{1}{2}$ is then linearly interpolated. Two previous values of R_s are always stored so that the scaling speed can be calculated using symmetric difference quotient numerical differentiation. This means that the scaling speed outputs are at slightly different rapidities from the dipole outputs if the two are output simultaneously in real time, but the difference has no particular impact beyond basic testing, so this does not in general prevent a single output step dealing with both the dipole and the scaling speed. This is only particularly relevant when, as is useful for most long runs, results are only output after an interval of a number of rapidity steps. Otherwise each value will be output at every rapidity value at which it

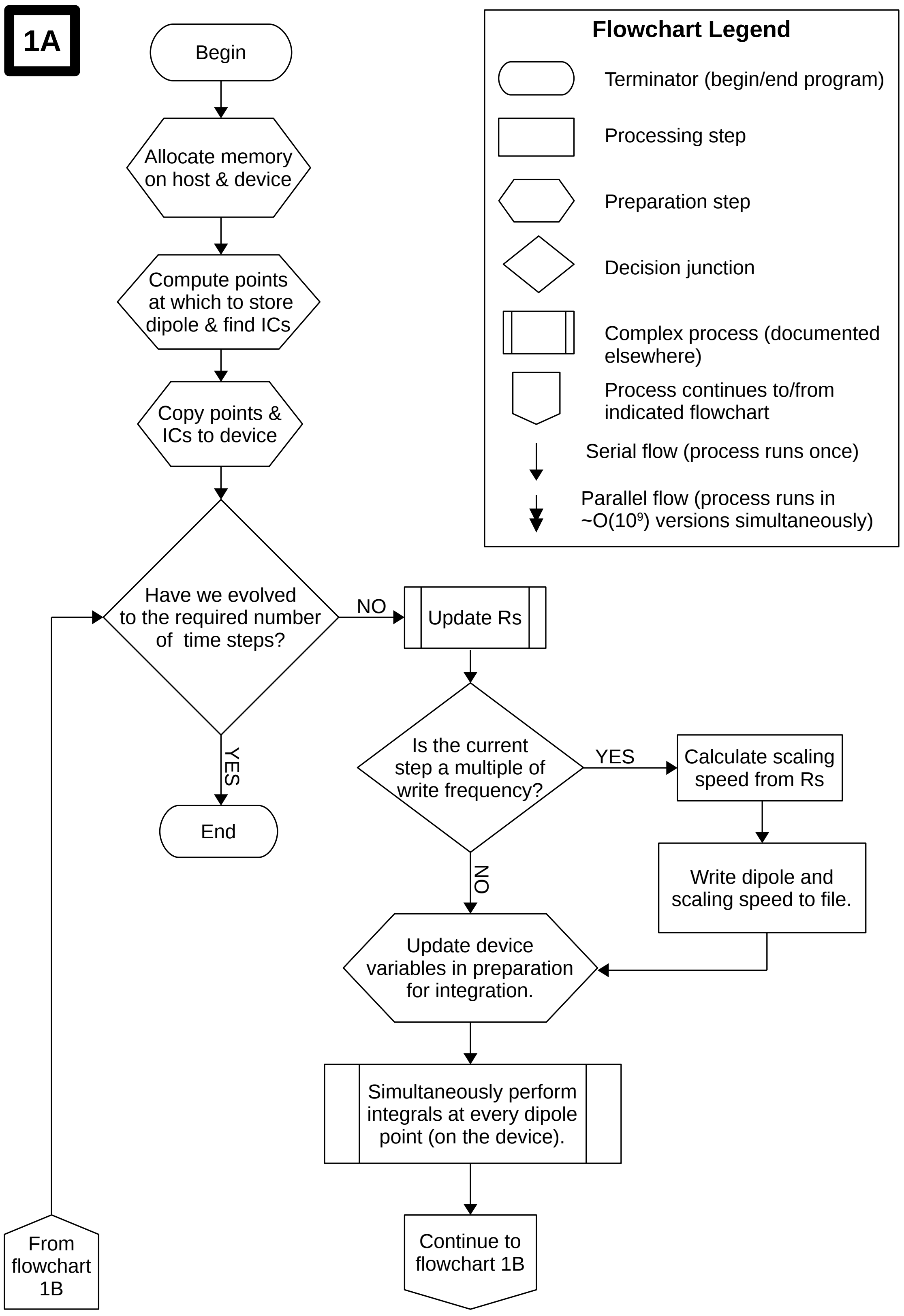

1A
Begin
Allocate memory on host & device
Compute points at which to store dipole & find ICs
Copy points & ICs to device
Have we evolved to the required number of time steps?
NO
Update Rs
YES
End
Is the current step a multiple of write frequency?
YES
Calculate scaling speed from Rs
Write dipole and scaling speed to file.
NO
Update device variables in preparation for integration.
Simultaneously perform integrals at every dipole point (on the device).
Continue to flowchart 1B
From flowchart 1B
Flowchart Legend
Terminator (begin/end program)
Processing step
Preparation step
Decision junction
Complex process (documented elsewhere)
Process continues to/from indicated flowchart
Serial flow (process runs once)
Parallel flow (process runs in ~O(10⁹) versions simultaneously)

FIGURE 3.1: Flowchart 1A illustrates the structural flow of the code that evolves the Wilson line 2-point function using the BK equation.It is continued in Figure 3.2.

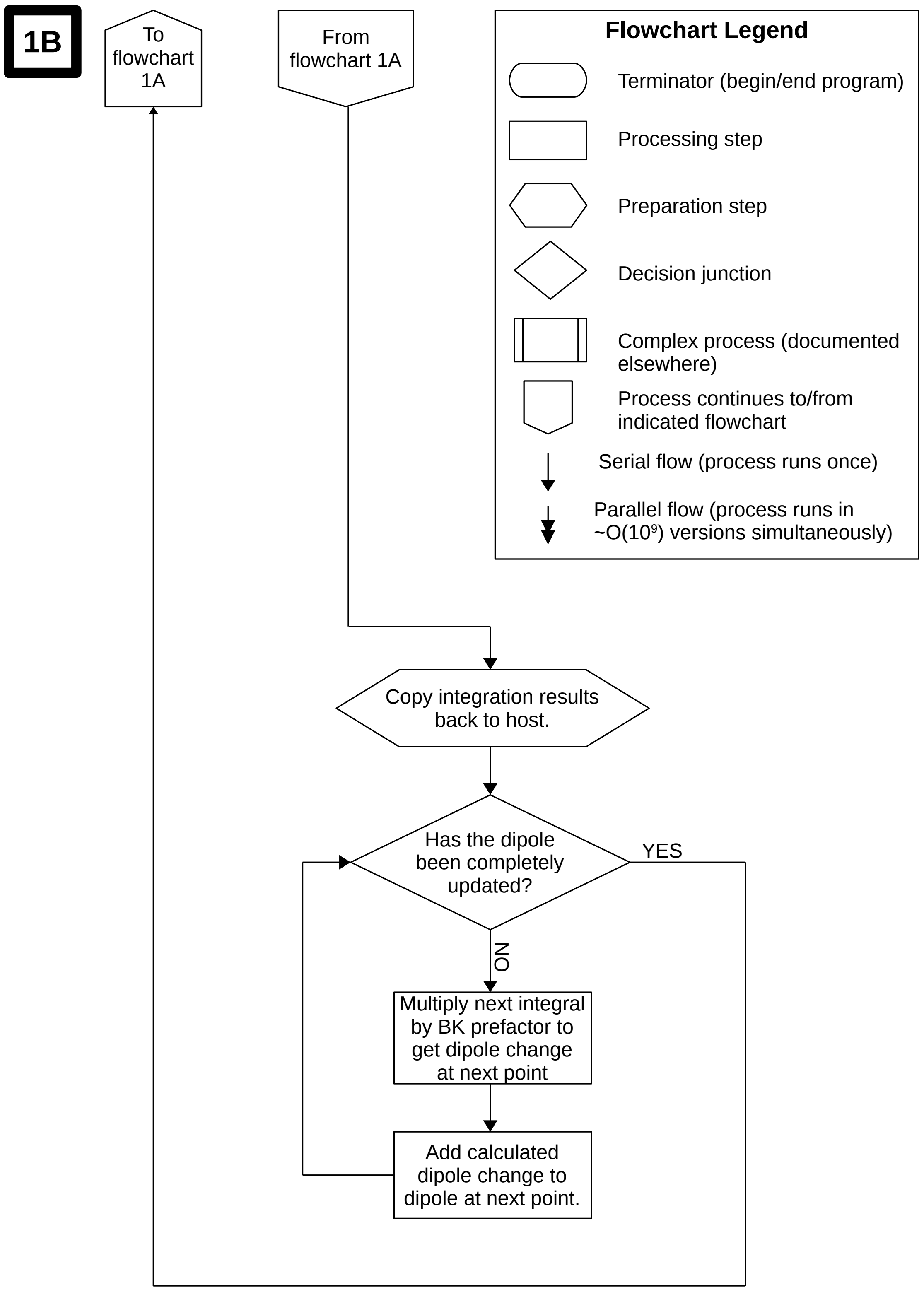

1B
To flowchart 1A
From flowchart 1A
Flowchart Legend
Terminator (begin/end program)
Processing step
Preparation step
Decision junction
Complex process (documented elsewhere)
Process continues to/from indicated flowchart
Serial flow (process runs once)
Parallel flow (process runs in ~O(10⁹) versions simultaneously)
Copy integration results back to host.
Has the dipole been completely updated?
YES
NO
Multiply next integral by BK prefactor to get dipole change at next point
Add calculated dipole change to dipole at next point.

FIGURE 3.2: Flowchart 1B illustrates the structural flow of the code that evolves the Wilson line 2-point function using the BK equation.It is continued from Figure 3.1.

is calculated in any case.

3.3 Numerical analysis techniques

3.3.1 Discretisation and integral limits

The task of storing and calculating $\hat{S}_{\boldsymbol{xy}}$ comes with inherent limitations. The first of these has been discussed above: the function must be discretised and calculated at discrete points. Given the shape of the function (as in Figure 4.1), and because we expect it to scale logarithmically, we choose to distribute these discrete points logarithmically. Since $\hat{S}_{\boldsymbol{xy}}$ depends only the dipole separation distance, we do not need to take a second dimension, such as a radial direction, into account at this point. However, a second limitation of the computer has already become apparent: even after discretising, we cannot store the function on an infinite interval. We must thus ensure that our finite interval is such that it is reasonable within our working accuracy to extrapolate that the function is unity between the smallest point and the origin and zero beyond the largest point.

As well as representing the dipole function, we need to calculate the integrals which are used to evolve it. These integrals are performed over a two-dimensional region. In principle this region is the infinite plane, but once the integrand becomes sufficiently small it is reasonable to approximate it by zero and thus reduce the region of integration. Additionally, the $\boldsymbol{x} \leftrightarrow \boldsymbol{y}$ symmetry of the integrand means that even in principle it need only be integrated over half the plane vertically and again, to a quarter, horizontally, with coordinates chosen as is illustrated in Figure 3.5. This means that only one apparent singularity need be integrated over, but substantially complicates the limits of integration. The integration region is essentially broken into two parts, described as regions A and B in Figure 3.5. For sufficiently small radial distance r, the integration region is simply a half-disk and the integral can be described as $\int_0^{|\boldsymbol{x}|/2} dr \int_0^{\pi} d\theta$. Beyond this disk lies a second region: here r extends out until infinity – or at least as far as the integration grid on the computer – and θ is chosen to go from one symmetry line to the other. Alternatively, one can think of θ as extending from 0 to π and the extent of r being constrained so that it does not cross the symmetry line. In either case, the equality at the symmetry line is $r\cos\theta = \frac{|\boldsymbol{x}|}{2}$. The integral can be written in the form $\int_{\cos^{-1}(|\boldsymbol{x}|/2r)}^{\pi} d\theta \int_{|\boldsymbol{x}|/2}^{\infty} dr$. Such a limit is cumbersome and inefficient to implement computationally and so instead the

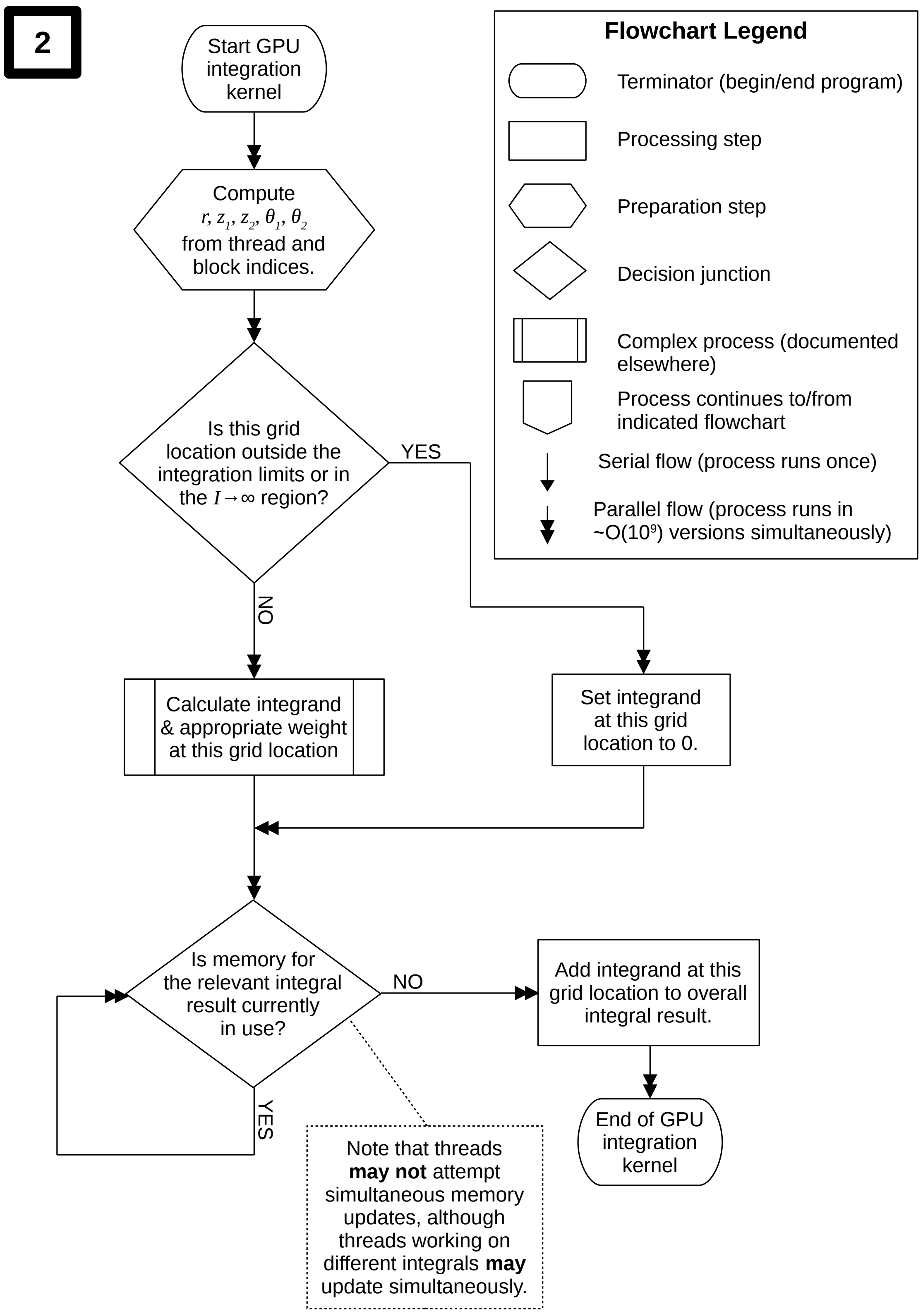
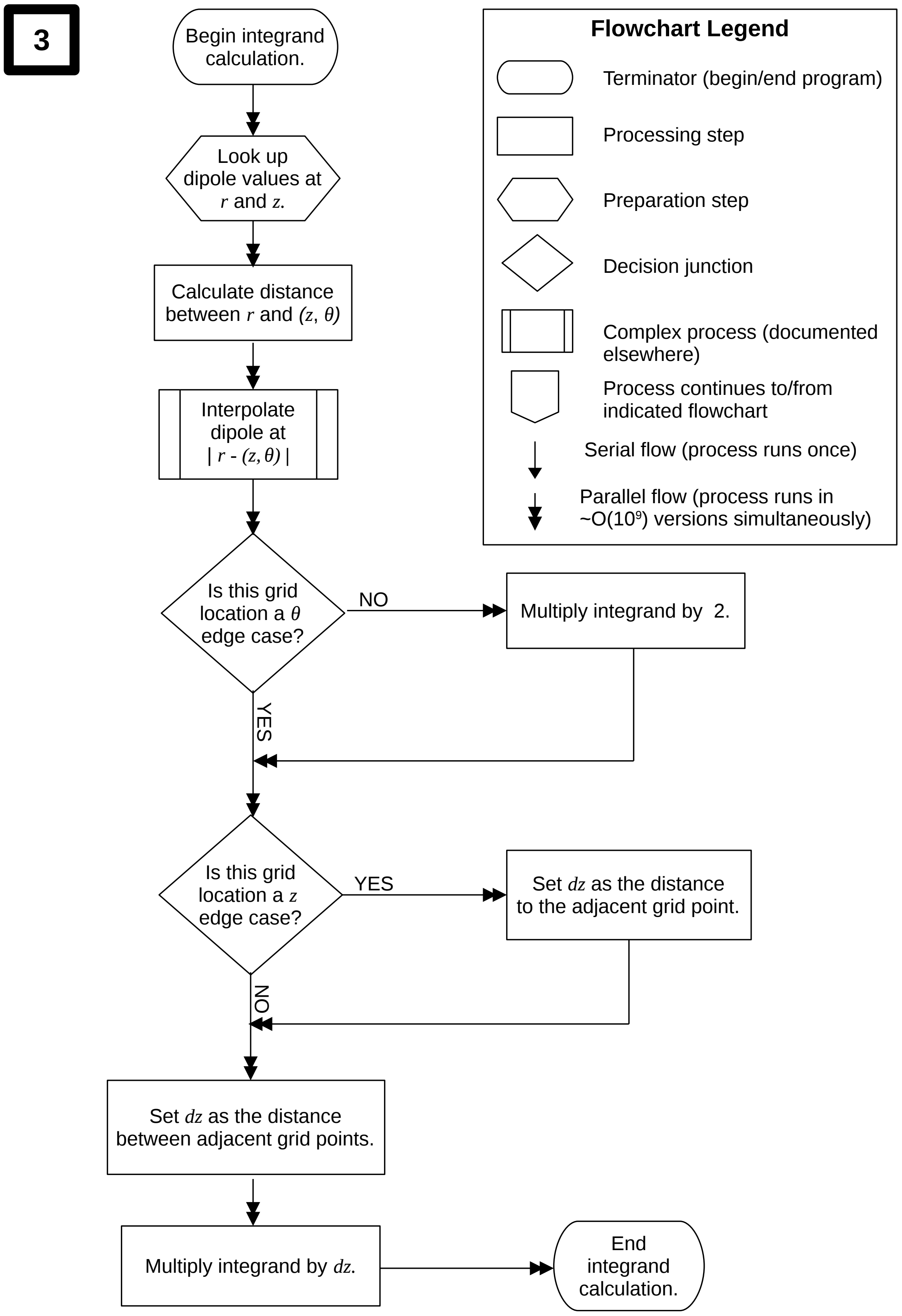

2
Start GPU integration kernel
Compute $r, z_1, z_2, \theta_1, \theta_2$ from thread and block indices.
Is this grid location outside the integration limits or in the $I \rightarrow \infty$ region?
YES
NO
Calculate integrand & appropriate weight at this grid location
Set integrand at this grid location to 0.
Is memory for the relevant integral result currently in use?
NO
YES
Add integrand at this grid location to overall integral result.
End of GPU integration kernel
Note that threads **may not** attempt simultaneous memory updates, although threads working on different integrals **may** update simultaneously.
Flowchart Legend
Terminator (begin/end program)
Processing step
Preparation step
Decision junction
Complex process (documented elsewhere)
Process continues to/from indicated flowchart
Serial flow (process runs once)
Parallel flow (process runs in ~O(10^9) versions simultaneously)

FIGURE 3.3: Flowchart 2 illustrates the structure of the code executed in parallel on the GPU device, using CUDA C. Since many steps of the evolution integrals are independent, the calculations can be performed simultaneously.



3
Begin integrand calculation.
Look up dipole values at r and z.
Calculate distance between r and (z, θ)
Interpolate dipole at | r - (z, θ) |
Is this grid location a θ edge case?
NO
Multiply integrand by 2.
YES
Is this grid location a z edge case?
YES
Set dz as the distance to the adjacent grid point.
NO
Set dz as the distance between adjacent grid points.
Multiply integrand by dz.
End integrand calculation.
Flowchart Legend
Terminator (begin/end program)
Processing step
Preparation step
Decision junction
Complex process (documented elsewhere)
Process continues to/from indicated flowchart
Serial flow (process runs once)
Parallel flow (process runs in ~O(10^9) versions simultaneously)

FIGURE 3.4: Flowchart 3 illustrates the flow of the integration process performed on the GPU device in parallel: since the integrand at a given point does not depend on the integrand at any other point, the integrand can be calculated at all points simultaneously.

function is redefined piecewise over the upper half plane: we simply check whether or not the point under consideration falls beyond the symmetry line. If it does so, the integrand there is set to zero without further computation.

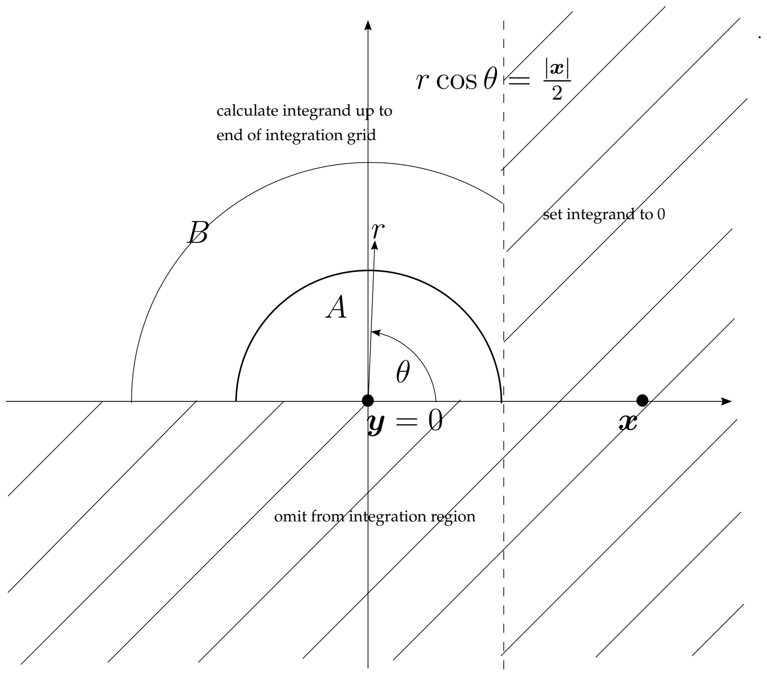

$r\cos\theta = \frac{|\boldsymbol{x}|}{2}$
calculate integrand up to end of integration grid
set integrand to 0
B
r
A
θ
$\boldsymbol{y} = 0$
$\boldsymbol{x}$
omit from integration region

FIGURE 3.5: The $\boldsymbol{x} \leftrightarrow \boldsymbol{y}$ symmetry of the integrand allows us to integrate over only half the plane, while the symmetry about the horizontal axis reduces this fraction further to a quarter. Additionally, we do not integrate over regions where the integrand may be reasonably approximated as zero, so that we can obtain a finite integration region.

After choosing the region of integration from symmetry constraints (although the limits at which the integrand becomes negligible are still to be addressed) we can consider the discretisation of the grid and the coordinate system in which to perform the integral. Following the discussion of Section 2.3.1 we choose to use polar coordinates within the integral. Given that there is a lot of detail near the origin, as witnessed by the apparent divergences there, it would be convenient to choose a radially logarithmic coordinate system for the integral. While transforming the integral introduces a number of unwanted complications, as discussed in Section 3.3.2, we can gain many of the same benefits by choosing to distribute our discretised points logarithmically. There are no particular features to induce us to choose our angular discretisation to be other than uniform, so we choose a polar grid with uniformly separated angular points and logarithmically space radial points. This is illustrated in Figure 3.6.

Having chosen the form of the distribution of the grid points, it remains to decide on the region over which they should be distributed. However, it is not entirely trivial to choose a point at which we begin to approximate the the integrand as zero. A first point to consider may be the point at which we begin to extrapolate the function as zero. Beyond this point we will certainly have $\hat{S}(z) = 0$, so that one term in the integrand is necessarily neglected, but the other term is of the form $\frac{1}{z(\boldsymbol{z}-\boldsymbol{x})^2}\hat{S}(r)$ and need not be especially small, recalling that we may have $\hat{S}(r) = 1$. The size of contributions at this limit is therefore a side effect of the choice of the scale on which the dipole function is stored. In practice, we have found that it is simplest to run the code for various integration regions so that the distance from the origin at which the integrand no longer makes a significant contribution can be gauged by considering the changes in the evolution results. This distance is typically around three times the distance at which the dipole correlator is extrapolated to zero, depending on the desired accuracy and other parameters of the solver.

3.3.2 Integration

Integral transformations and divergences

In Section 2.3.1 we transformed the BK integral to polar coordinates, so that it had form

$$\int dz d\theta \frac{1}{\left(r^2+z^2-2rz\cos\theta\right)z^2}\left(S(\sqrt{r^2+z^2-2rz\cos\theta})S(z)-S(r)\right), \qquad (3.1)$$

where we have factored out everything that we can. In this work, following older work, we have not further transformed the integrand, instead using a grid point spacing tailored to the nature of the integrand.

Considering the logarithmic nature of the scaling behaviour and the logarithmic choice of discretisation made in Section 3.3.1 above, it does make sense to further transform the integral so that $z \to \zeta = \ln(z)$. Then ζ would range from $-\infty$ to ∞ with the negative half line representing the region into which we expect the interesting part of the dipole correlator to evolve. Doing do would produce an integral of the form

$$\int d\zeta d\theta \frac{1}{\left(r^2+e^{2\zeta}-2re^{\zeta}\cos\theta\right)}\left(S(\sqrt{r^2+z^2-2rz\cos\theta})S(z)-S(r)\right). \qquad (3.2)$$

The evaluation of this integral on a uniform grid is equivalent to our evaluation of the untransformed integral on a logarithmic grid. However, since we have not transformed the integral, we will reformulate the trapezoidal rule to account for the non-uniform grid.

The trapezoidal rule with a non-uniform grid

We choose our integration grid to be logarithmic in the radial direction, as the most important features of the dipole are found logarithmically near to the origin, as discussed in Section 3.3.1. Such a grid is schematically illustrated in Figure 3.6, both in the 'circular' sense usually associated with a radial grid and in the rectangular sense in which we may deal with the grid after applying the Jacobian factor. We will use this second image as a visual aid in deriving the two-dimensional trapezoidal rule on a radially logarithmic grid. Three points which will make different contributions to the integral are picked out: a point in the middle of the grid (B), a point at a θ-extreme of the grid (A) and a point at a z-extreme of the grid(C). We do not specifically include a corner point, as this is merely the special case where a point is on both the θ-extreme and z-extreme of the grid simultaneously and should follow both sets of special rules.

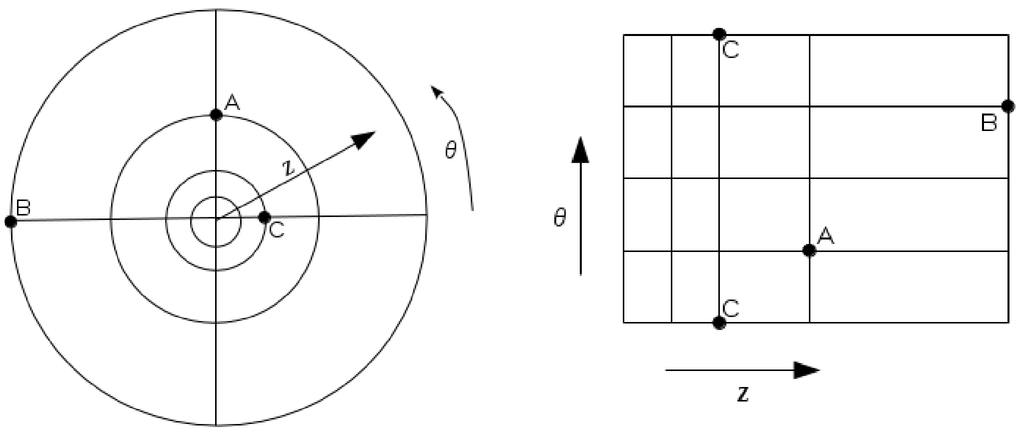


FIGURE 3.6: The radially logarithmic integration grid is shown schematically, both before and after the effect of the Jacobian factor is taken into account. Points A, B and C will each contribute differently to the integral result.

Point A is the simplest and most common case. The integrand here will contribute to the average in all four surrounding blocks. In the θ part of the integral, we simply weight this point by 2 before the measure is calculated (as in the one-dimensional trapezoidal rule). In the z part of the integral we must take into account the non-uniformity of the grid. This can be done by treating the measure dz as dz_i where i references the grid block in question. The measure must then be included in the weight for the integrand at each points. For point A the weight is just $dz_{left} + dz_{right}$.

Point B again is multiplied by 2 in the θ integral. However dz_{right} is not defined here: the point is only part of two blocks. The z weighting of this point is therefore just dz_{left}. Similarly a point at the left extreme of the z values we consider would be weighted by dz_{right}.

Point C is perhaps the strangest point: in the circular diagram it appears once, participating in four blocks, while in the rectangular diagram it appears twice, contributing to two different blocks in each case. This is because the circular diagram explicitly incorporates the periodicity of the integrand in θ, while the rectangular grid does not. One might consider that point C contributes to four blocks or that it contributes to two blocks at $\theta = 0$ and another two blocks at $\theta = 2\pi$. We will adopt the latter approach, as this generalises much better. For example, if we integrate from $\theta = 0$ to $\theta = \pi$, over the upper semicircle of the circular diagram, we would only want point C to contribute twice. The θ weighting of points on θ extremes is thus 1 (with $d\theta$ to be considered later), while the z weighting is determined as for point A or B, whichever is appropriate.

Algebraically, the rule is

$$\begin{aligned}
4I =& 2\sum_{i=1}^{n-1}\sum_{j=1}^{m-1} f(z_i,\theta_j)(z_{i+1}-z_{i-1})d\theta + \sum_{i=1}^{n-1} f(z_i,\theta_0)(z_{i+1}-z_{i-1})d\theta + \\
&+ \sum_{i=1}^{n-1} f(z_i,\theta_m)(z_{i+1}-z_{i-1})d\theta + \sum_{j=1}^{m-1} f(z_0,\theta_j)(z_1-z_0)d\theta + \\
&+ \sum_{j=1}^{m-1} f(z_n,\theta_j)(z_n-z_{n-1})d\theta + f(z_0,\theta_0)(z_1-z_0)d\theta + f(z_0,\theta_m)(z_1-z_0)d\theta + \\
&+ f(z_n,\theta_0)(z_n-z_{n-1})d\theta + f(z_n,\theta_m)(z_n-z_{n-1})d\theta
\end{aligned} \tag{3.3}$$

This process is described, perhaps more efficiently, as a flowchart in Figure 3.4

3.4 Testing

Beyond comparing our results with values obtained elsewhere, we can perform various other tests on the code to ensure that it behaves as we expect. The key features of this testing and results thereof are outlined in this section.

One of the most obvious candidates for unit testing – that is, testing of a single code segment independently of the rest of the code – is the interpolation routine. Even an apparently minor edge case error in interpolation can dramatically change the results of the evolution, as the function we are integrating now differs significantly from the dipole we intend to integrate. We test the interpolator by printing both the function at the stored points and the interpolated function at various different points. There are a few test cases of interest. Does the interpolator correctly interpolate the function at points where the actual value is stored? Does it correctly interpolate the function halfway between two such points? Does it correctly interpolate the function at an arbitrary point between two stored points, but closer to one than the other? Does the interpolation routine handle cases where it is asked to extrapolate? Plotting the results of interpolation at the relevant points against the stored values at the stored points allows us to quickly diagnose problems with the interpolation routine. An example is given in Figure 3.7.

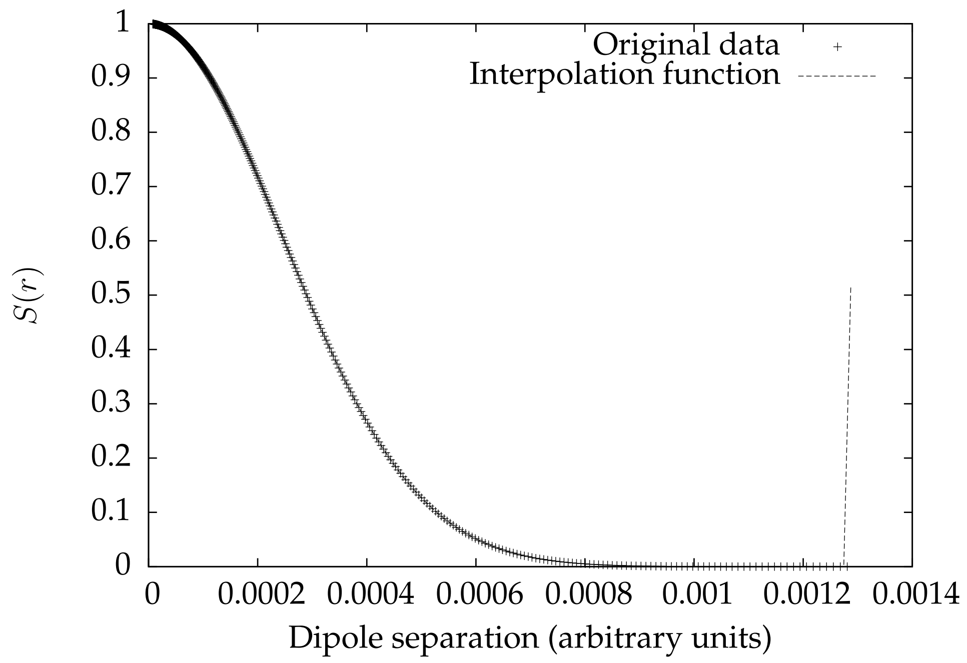


FIGURE 3.7: Plotting the original data and results of interpolating points near it makes it visually obvious when the interpolator has misfunctioned, which is invaluable in development. Here, as an example, is shown a case where the interpolator has failed to make a reasonable extrapolation when asked for a point beyond the range of the original data.

As discussed in Section 3.3.1, it is necessary to test the bounds used for the various integrals to ensure that these are not artificially restricting the RHS produced through the integrals in question. In Figure 3.8 we plot the right-hand-side of the BK equation at initial rapidity for various choices of the integration boundaries. This allows us to find a point beyond which increasing the integration region does not affect the result, thereby ensuring that the result is not artificially diminished. This plot can also be used to confirm that the initial right-hand-side has the expected shape and behaviour.

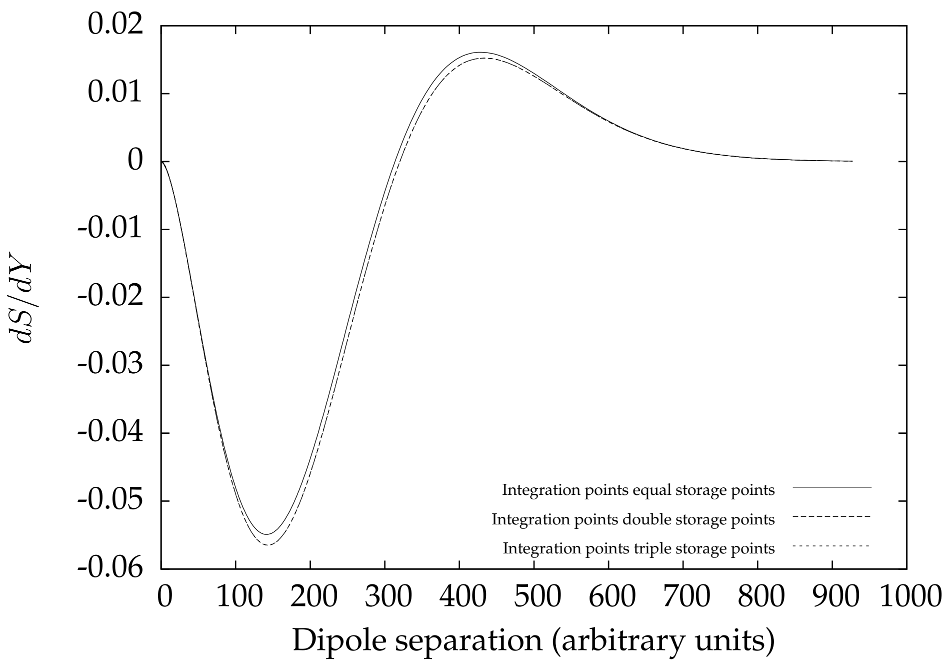


FIGURE 3.8: Comparison of the initial BK equation right-hand-side, where $S(r)$ is Gaussian, under different conditions. This allows us to choose numeric boundaries for the integration region that do not affect the result of the integration. We observe that the dashed lines which represent doubling and tripling the number of points in the integration grid, respectively, lie exactly on top of one another. Increasing the number of points considered beyond doubling them has no effect on the result and would be a waste of computational resource. However, there is a significant change from the case where the integration grid is no larger than the storage grid. It should be noted that since the grid is logarithmic, doubling the number of points considered does more than double the radial distance considered – rather, it squares it.

If the dipole evolution occurs correctly, it should be possible to calculate the correlation length for each rapidity and hence the change in the scaling speed with rapidity. From Section 2.3.2, we expect that the scaling speed must reach some asymptotic value if the coupling is fixed, since the scaling behaviour becomes exact. This behaviour is demonstrated in Chapter 4. A contributor to the scaling behaviour that must also be tested is the correlation length calculator. This can be tested visually by plotting the dipole function at various rapidities, in each case scaled by the appropriate correlation length. This should result in the coincidence of the midpoints of all the dipole functions and eventually, when plotted on a logarithmic scale, the dipole functions at high rapidities should be indistinguishable, since the only distinguishing feature, the scaling, has been removed. This behaviour can be observed in Figure 4.3 of the following chapter.

3.5 Alternative techniques and potential future work

There are a number of numerical methods and techniques which have not been directly utilised in this project, but which would bear further investigation. For example, one dimensional work (in which the angular part of the integral has been factored out) has made use of fast Fourier transforms (FFTs) to compute the integrals. This can in principle also be done in the two dimensional case. The integral can be written in terms of convolutions as

$$\int d^2z \frac{S(x-z)}{(x-z)^2}\frac{S(z)}{z^2} - S(r)\int d^2zK(x,z) = (f*f)(x) - S(r)\int d^2zK(x,z) \tag{3.4}$$

where $f(x) = \frac{S(x)}{x^2}$ and $K(x,z) = \frac{1}{(x-z)^2z^2}$. After Fourier transforming the convolution, one need only multiply, instead of integrating: thus the use of FFTs significantly boosts efficiency. However the technique is generally less useful in the two-dimensional case, as it becomes impractical to store $f(x)$ on a sufficiently fine uniform grid and FFTs traditionally require that the function to be transformed is stored on a uniform grid. However, work in numerical analysis such as [43, 44] has introduced methods for performing Fourier transforms on non-uniform grids at speeds comparable to those of FFTs. These methods may make it plausible to solve the two-dimensional equation in Fourier space, although further investigation would be required.

Alternatively, if the equation is solved using direct integration, there is scope to investigate more sophisticated quadrature rules. While the two-dimensional trapezoidal rule described above is sufficient to demonstrate the utility of parallelising the evolution calculation, more sophisticated quadrature rules should improve the efficiency of the code, perhaps significantly. In particular, the transformation discussed in Section 3.3.2 should make the implementation of integration rules such as Gaussian or Curtis-Clenshaw quadrature relatively simple.

Another conventional technique which may have a significant impact on the efficiency of the code is the choice of ODE stepper. The code in this project simply uses an Euler forward step algorithm; an algorithm such as fourth order Runge-Kutta should improve the efficiency dramatically if it can be done without introducing new overheads into the parallelisation process. As the code is presently structured, only the integration phase of evolution is parallelised. The update phase is not performed on the GPU, but on the host device. Since Runge-Kutta methods require several integrations and updates in each time step, implementing such a method would introduce a large overhead as the results of various integrals are copied back and forth between the host and the device. However, if the code was structured such that the dipole update phase occurred on the device, there would be no such overhead and the speed increase possible by using fourth order Runge Kutta ODE stepping could be investigated.

Much of the programme's resource-hungry nature is controlled by the requirement for the dipole to be stored on a huge grid encompassing many scales. Since the dipole does not occupy all of these scales at once, it may be possible to make use of a dynamic grid, which stores only the parts of the dipole that it does not make sense to extrapolate at a given rapidity and changes with the rapidity under consideration. Initial experimentation with such a grid system has shown that it is less obviously useful than it might seem, because the bulk of the code executes while the dipole is changing shape. During this shape change, the curve tends to become shallower (for the ICs considered) so that the relevant part of the dipole function effectively becomes longer. Thus for a significant period of evolution, the grid would not be able to shift, or would shift very little, making the dynamic property less useful than it initially appears. However, it may be worth investigating the effect of applying such a grid in the later stages of evolution, which has not been done comprehensively in this project.

Chapter 4

Results and discussion

4.1 Solutions to the BK equation

In this section we present the results of evolving the dipole function from Gaussian initial conditions using the BK equation. The width of the Gaussian initial condition is chosen so that the dipole function is 10^{-6} at the end of the dipole storage grid. Changing the width of this Gaussian is essentially a rescaling of the units, since it changes the value of $R_s(Y_0)$ without changing the shape of the dipole function. The coupling constant is set to $\alpha_s = 0.217$. The function is stored at 2750 points, allowing ample room for the evolution from 0 to 38 rapidity. Evolution proceeds in steps of size 0.01, for a total of 4000 steps taken. Integration is perform over 128 angular increments and 5500 radial increments (a total grid size of somewhat more than 700,000 points). Significantly smaller grid sizes resulted in numerical artifacts, such as oscillations, bad extrapolation or the inconsistency demonstrated in Figure 3.8. With these parameters, on a commercially available GPU (see Table 4.2), the code finished execution in about 31 hours, as the dipole function began to scale.

In Figure 4.1 we plot the dipole function over its evolution using uniformly scaled axes. It is evident from the plot that it is more natural to display the dipole separation on a logarithmically spaced grid to show the relevant detail of the function and we do this in Figure 4.2. The scaling behaviour also becomes evident in this plot. Since we have derived the scaling to be logarithmic in Section 2.3.2, it is unsurprising that this is seen on a logarithmic grid.

We can examine the scaling behaviour more closely by scaling the dipole function at each rapidity by the corresponding correlation length. This is done in Figure 4.3. Rapidity evolution of the curve in this sense is from less dense to more dense, as the scaling speed increases with rapidity. As the solution begins to scale, the curves in Figure 4.3 lie directly on top of one another. This is further elucidated in Figure 4.4, which shows $\frac{dS}{dY}(r/R_s(Y), Y)$ over a range of rapidities,

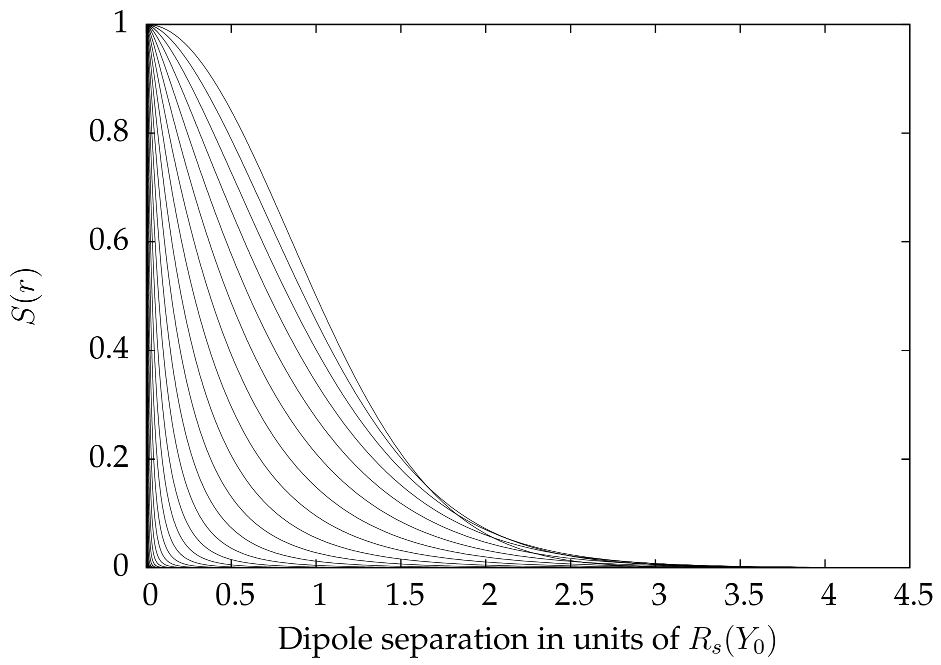


FIGURE 4.1: The evolution of the dipole from a Gaussian initial condition is plotted on a uniform scale. It is clear that much of the detail will be better displayed using a logarithmic scale; this will also demonstrate the behaviour. Such a plot is shown in Figure 4.2.

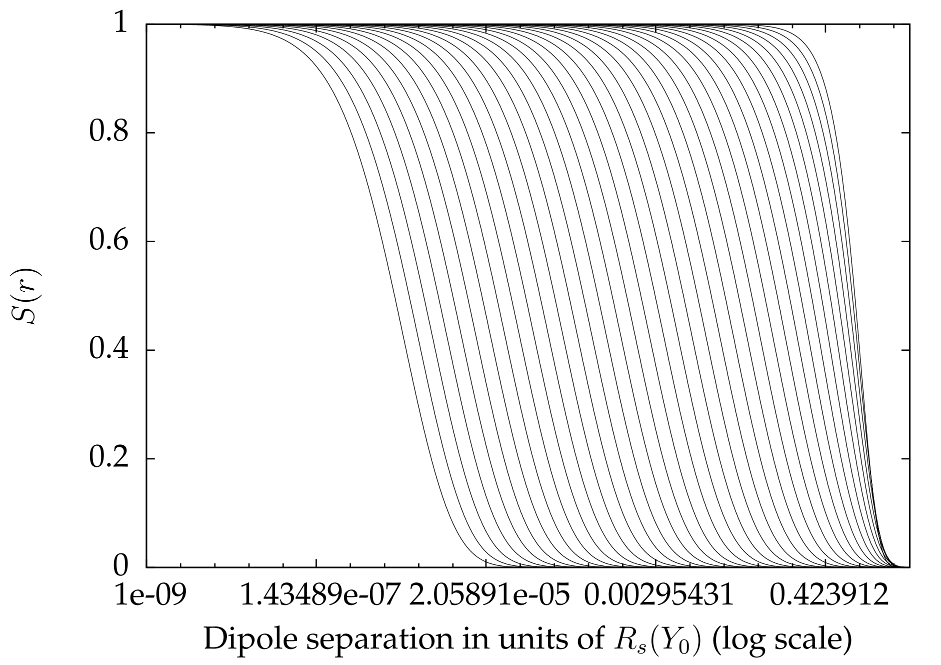


FIGURE 4.2: The evolution of the dipole function from Gaussian initial conditions through 39 units of rapidity is plotted. Evolution used the BK equation with fixed coupling. Although the function initially changes shape, by the end of the evolution it scales with no shape change.

in each case scaled by the appropriate correlation length. Initially the shape is relatively complex as the dipole function undergoes its most significant shape changes. However, it can be seen that, as one might expect, the $\frac{dS}{dY}(r/R_s(Y), Y)$ term tends towards a final shape, with ever smaller changes (recalling that in this form scaling effects are removed, as in Figure 4.4) until it reaches a constant form in which scaling effects are the only change. This is consistent with the fashion in which the trajectories in Figure 4.3 become denser as they approach scaling form.

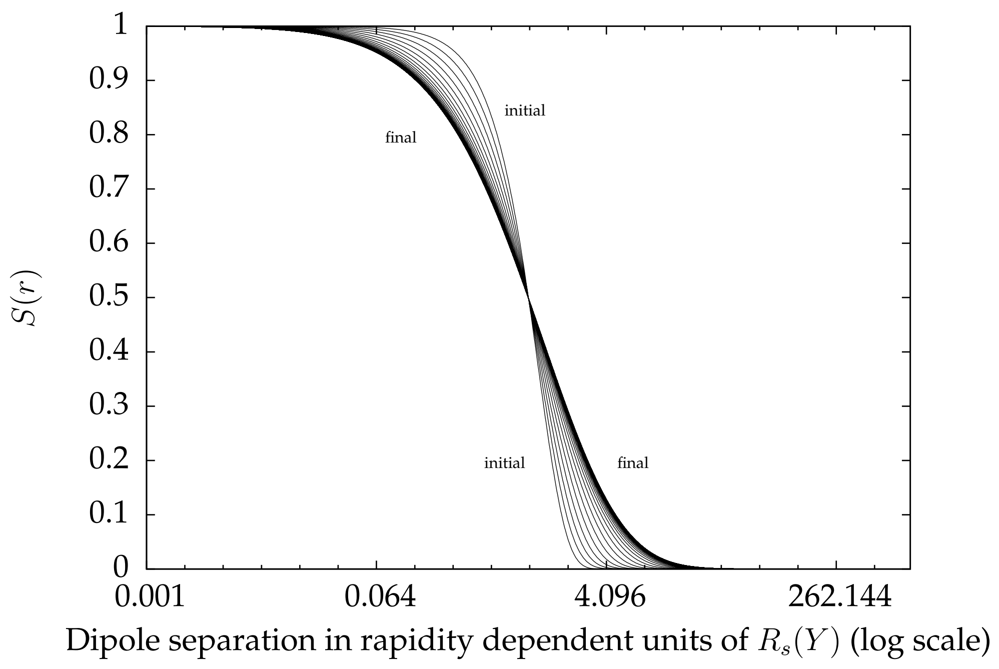


FIGURE 4.3: Here the dipole function is plotted at various rapidities. In each case the dipole separation is expressed in units of the correlation length of the rapidity in question. This rescaling means that the scaling effects of evolution become invisible in this plot and only the shape-change effects remain. Because eventually all effects are scaling effects, the curves for the dipole function at the last several rapidities plotted lie directly on top of one another. Beyond observing that this scaling behaviour is indeed evident, we can also note the nature of the change of shape during the initial stage before the solution begins to scale. We see that the curve of the Gaussian initial condition (marked in in the plot) becomes shallower over the course of evolution as it tends toward the scaling shape. We also note that while the scaling speed increases with rapidity (see Figure 4.5), the change of shape begins rapidly and slows as the final shape is produced, resulting in the denser set of curves near to the final state.

We also examine the scaling behaviour via the scaling speed in Figure 4.5. This shows the scaling speed calculated at various rapidities, plotted against the corresponding correlation lengths. The data corresponding to this plot is shown in Table 4.1 and allows us to confirm that λ has become constant to

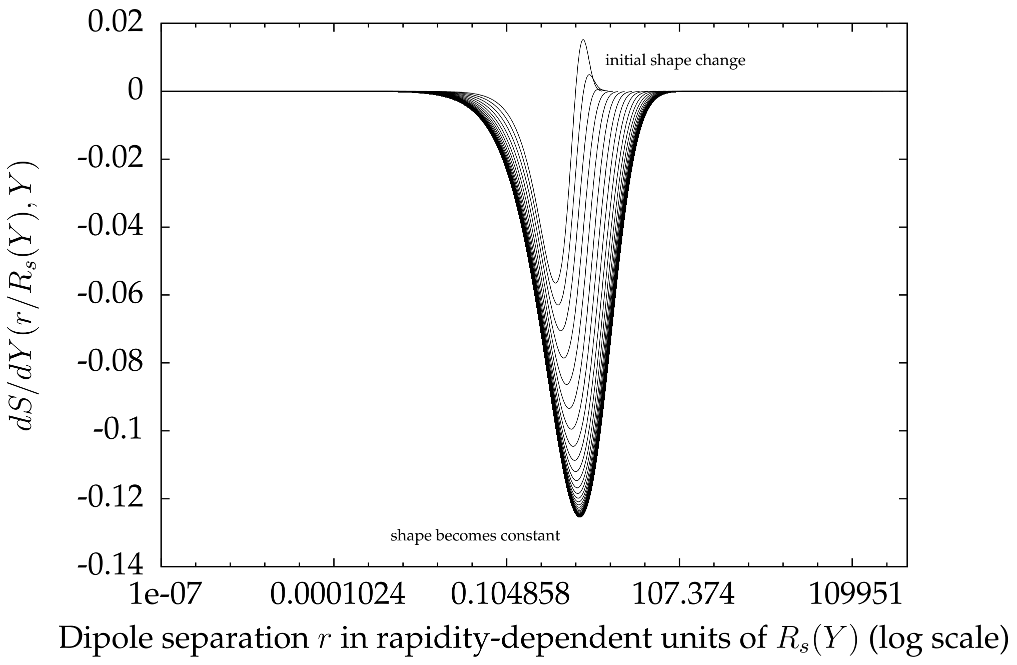


FIGURE 4.4: This plot shows a scaled version of the right-hand-side of the BK equation, giving the change $\frac{dS}{dY}(r/R_s(Y),Y)$ for various rapidities, beginning with a Gaussian initial condition. The scale is rapidity-dependent and chosen to minimise scaling effects in the plot, so that changes of shape become evident. Initially we see that $\frac{dS}{dY}(r/R_s(Y),Y)$ has a relatively complex form, as the dipole function undergoes its most dramatic shape changes. This rescaling of the right-hand-side of the BK equation then tends towards its final state. The shifts are ever smaller, until it becomes constant, signifying that the only change which the dipole function undergoes is logarithmic scaling, with no change to its shape. The fashion in which these shifts become increasingly small is consistent with the fact that in Figure 4.3 the scaled trajectories of the dipole function become increasingly dense before finally becoming constant. We observe that the dip in the change function becomes deeper as the scaling speed increases (with increasing rapidity) and moves to the right as the function assumes its final form.

within numerical noise by the end of the evolution period. This satisfies the scaling properties of Section 2.3.2 and is an important test of the code. It is also one of the most difficult tests to perform, since the dipole function does not begin to scale long before the rapidity reaches 40, although R_s and the region in which the dipole function has the most detail varies a great deal over this range. The computational resources required to deal with this are substantial. Including next-to-leading order corrections by allowing the coupling to run would make this test impossible, since true scaling is never achieved when the coupling runs, but also significantly slow down the evolution of the dipole function, so that it is not necessary to reach such a large range of R_s values before the function reaches the scaling or pseudoscaling region of evolution.

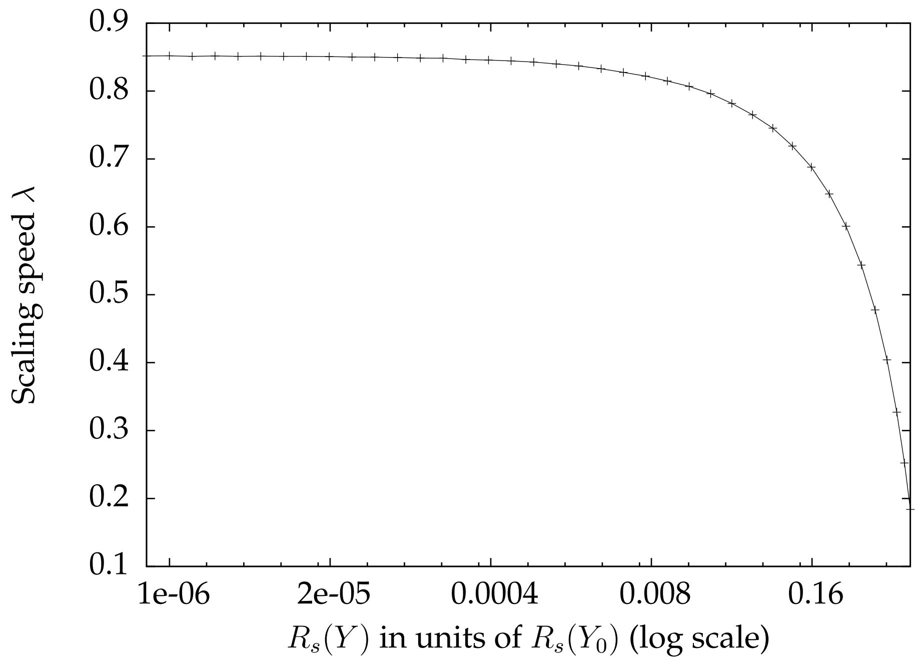


FIGURE 4.5: The evolution of the scaling speed through 40 units of rapidity is plotted, where the dipole initial condition was Gaussian. Evolution made use of the BK equation with fixed coupling. Initially the scaling speed changes rapidly, but by the end of the evolution period the solution is scaling and λ reaches a final value of 0.851. Notice that the scaling speed only flattens out over the last few time steps, although a visual inspection of the dipole function might have suggested that true scaling had been reached much earlier. By consulting Table 4.1 we see that just before 40 rapidity steps, the scaling speed truly begins to stabilise to within the accuracy of the numerics.

Our results here can be compared to those of Kuokannen, Rummukainen and Weigert in [37], where they find similar results for fixed coupling evolution. For illustration purposes, we also include their plot for the running coupling

Y	$R_s(Y)$	λ
1	192.489	0.183868
2	172.635	0.252199
3	149.372	0.327253
4	124.428	0.404215
5	99.8093	0.477695
6	77.2925	0.543785
7	58.0264	0.600815
8	42.436	0.648499
9	30.3714	0.688052
10	21.3589	0.719047
11	14.8094	0.745344
12	10.1512	0.765085
13	6.89398	0.781667
14	4.64698	0.796068
15	3.11339	0.806856
16	2.07571	0.814718
17	1.37838	0.822076
18	0.912378	0.827465
19	0.602345	0.832838
20	0.396835	0.836915
21	0.261000	0.83977
22	0.171425	0.842571
23	0.112467	0.844312
24	0.073721	0.845599
25	0.0482896	0.846314
26	0.0316133	0.848246
27	0.0206860	0.848603
28	0.0135310	0.84939
29	0.00884799	0.849867
30	0.00578459	0.850105
31	0.00378100	0.85063
32	0.00247105	0.851154
33	0.00161473	0.851011
34	0.00105508	0.851297
35	0.000689341	0.851059
36	0.000450357	0.851727
37	0.000294211	0.851154
38	0.000192195	0.851822
39	0.000125553	0.851631

TABLE 4.1: The numerical values obtained for the scaling speed at various rapidities. These are plotted in Figure 4.5. Inspecting the last few values in the table, we see that the scaling speed has stablised to within numerical accuracy.

case in Figure 4.6. We anticipate that similar results would be obtained upon implementing running coupling in the code produced here.

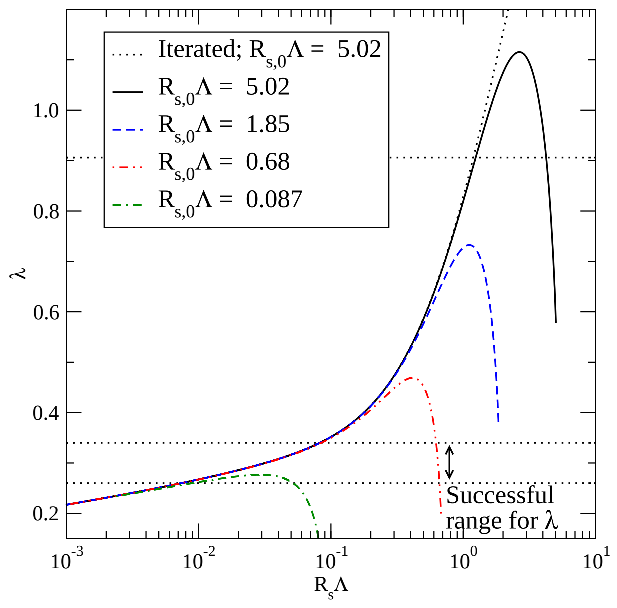


FIGURE 4.6: The results of Kuokannen, Rummukainen and Weigert in [37] for BK evolution with running coupling are displayed for illustrative purposes. This behaviour can be anticipated as the result of extending the code developed here to include running coupling effects.

4.2 Computational performance and benchmarking

Details in the literature on the computational performance of this type of code are scarce. However, from informal discussions and comparisons, we expect this code to have about an order of magnitude improvement in accuracy/-time performance over typical serial implementations. The statistics presented here are for evolution to full scaling behaviour with fixed coupling. It should be remembered that if the coupling was allowed to run, the evolution would become simpler in the sense of scales to be considered (since the evolution speed would be slowed significantly[41]) and computation would be significantly faster. Nonetheless, calculations of interest, such as odderon evolution, require increasing large amounts of storage and computation. In the odderon case, evolution also encompasses an imaginary part and the dipole function must be stored in two dimensions, with full angular dependence. The performance increases realised in this situation will be equally applicable in those, with the added advantage of the possibility of rigorous testing in the fixed coupling case.

GeForce GTX 970	
CUDA Driver Version / Runtime Version	7.5 / 7.5
CUDA Capability Major/Minor version number:	5.2
Total amount of global memory:	4096 MBytes (4294967296 bytes)
(13) Multiprocessors, (128) CUDA Cores/MP:	1664 CUDA Cores
GPU Max Clock rate:	1253 MHz (1.25 GHz)
Memory Clock rate:	3505 Mhz
Memory Bus Width:	256-bit
L2 Cache Size:	1835008 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes, with 2 copy engines
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Disabled
CUDA Device Driver Mode (TCC or WDDM):	WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA):	Yes
Device PCI Domain ID / Bus ID / location ID:	0 / 1 / 0
Compute Mode:	Default

TABLE 4.2: This table displays the full specifications of the GPU used to evolve the dipole function in this project.

We present the detailed specifications of the GPU used for evolution in Table 4.2. Table 4.3 shows a profile of the code's execution via NVIDIA tool `nvprof`. We see that the overwhelming majority of computation time is devoted to integral calculations, which is to be expected when an effort is made to minimise the overheads of the programme.

Profile:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	2e+05s	4000	42.1068s	42.0905s	42.6120s	integrationKernel
0.00%	15.747ms	8001	1.9680us	1.9190us	8.1280us	[CUDA memcpy HtoD]
0.00%	12.500ms	4000	3.1240us	3.0710us	8.8000us	[CUDA memcpy DtoH]

API calls:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	2e+05s	12001	14.0345s	6.1990us	42.6120s	cudaMemcpy
0.00%	150.16ms	3	50.052ms	7.5400us	150.14ms	cudaMalloc
0.00%	43.501ms	4000	10.875us	9.9120us	155.08us	cudaLaunch
0.00%	2.4437ms	12000	203ns	134ns	7.9270us	cudaSetupArgument
0.00%	1.5086ms	4000	377ns	306ns	3.5070us	cudaConfigureCall
0.00%	288.64us	83	3.4770us	319ns	118.89us	cuDeviceGetAttribute
0.00%	37.546us	1	37.546us	37.546us	37.546us	cuDeviceTotalMem
0.00%	31.256us	1	31.256us	31.256us	31.256us	cuDeviceGetName
0.00%	2.0070us	2	1.0030us	587ns	1.4200us	cuDeviceGetCount
0.00%	1.1140us	2	557ns	399ns	715ns	cuDeviceGet

TABLE 4.3: The analysis of NVIDIA profiler `nvprof` of the CUDA code used to evolve the BK equation is shown here for a sample run of 4000 time steps. Only the integration procedure and the copying of data between the GPU and the host computer are significant contributers to computation time, but each of these makes a large contribution.

Chapter 5

Conclusion

In this project we have presented a performance-oriented implementation of numerical BK evolution. We have made use of parallelisation and GPU programming techniques to enhance the speed of the code, allowing for greater accuracy within a given time frame for computation. The performance improvements over typical serial code are estimated to around an order of magnitude. Possibilities to further improve the numerical performance of the code have been discussed in Chapter 3. These include implementing superior ODE stepping, such as the fourth order Runge Kutta method and investigating better quadrature rules for the integrals which are so frequently performed in BK evolution.

Additionally, there are a number of theoretical extensions and applications to which the code is well suited and which could be implemented straightforwardly. In this sense the code has been designed as a cornerstone on which more complex evolution programmes can be built, making use of the same numerical framework. The simplest of these extensions would be to introduce NLO correction in the form of running coupling, which would require only minimal extensions. Further work might go on to include higher order corrections to the 2-point function, ,making use of the appropriate resummations. It would also be possible to include higher order corrections to the 2-point function, as in [1], where the increase in available accuracy may be helpful in addressing questions of instability discussed there and in [2, 46]. It would also be possible to extend the code to consider 3-point and 4-point functions. Another option made available for exploration is that of evolving the dipole function in the full Gaussian truncation, rather than the large-N_c limit of the BK equation. This extensibility makes the code well suited to applications in topics such as single transverse spin asymmetries and transverse momentum distributions, as in the context of [6–8].

Overall, then, we have introduced the ideas of the CGC and JIMWLK evolution. Considering the evolution of the Wilson line 2-point correlator, we have also introduced the BK equation and considered its relation to the Gaussian truncation of JIMWLK evolution for the 2-point function. Considering the properties of the BK equation, we have implemented a parallel version of evolution of the dipole correlator, with the result of significant performance increases. We have used this code to evolve the dipole under conditions of fixed coupling. There are numerous potential extensions to the code, many of them straightforward, which would allow application to interesting issues in modern CGC physics. Possibilities for further work on this topic are broad indeed.

Appendix A

Code

```
/* --------------------- BALITSKY-KOVCHEKOV EQUATION ON A GPU -------------------- *\
|This code depends on CUDA (see developer.nvidia.com/cuda-toolkit for installation).|
|It solves the BK equation with fixed coupling and outputs to the files            |
|       dipole.dat      scaling-speed.dat       rhs.dat                             |
|                                                                                   |
|With the CUDA toolkit installed, compile with                                      |
|       nvcc parallel-BK.cu -o <output file name>                                   |
|                                                                                   |
|Charlotte Hillebrand-Viljoen 2015-2016                                             |
\*---------------------------------------------------------------------------------*/


//we include libraries for IO and basic maths
#include <math.h>
#include <iostream>
#include <fstream>

//the std namespace makes IO less clumsy
using namespace std;


//parameters have been done via preprocessor constants to avoid memory allocation issues
//replacing array declarations with pointers and allocating memory by hand would mean these
//definitions could be read into variable from a parameters file instead

#define dipoleSize 2750         //this is the number of points at which the dipole is stored & governs the level of detail near 0
#define thetaSize 128           //the number of angular increments in the integral. must be <1024 due to CUDA thread # limits
#define extraZ 2                //how far beyond the storage limit do we extend the integration limit?
                                //the total grid must not exceed GPU memory! (2GB in my case)
                                //dipoleSize^2*extraZ*thetaSize*sizeof(float)

#define dY 1e-2                 //time step
#define totSteps 4000           //number of steps to evolve in total
#define stepsPerWrite 100       //number of steps to evolve before writing to file


//these variable are declared as global constants, to the dismay of good practitioners everywhere
//h_ and d_ prefixes indicate host and device variables respectively. this will become apparent when allocate memory
//the Rs variables have no prefixes, because the device never deals with that sort of information
//you will notice that we use floats and not doubles -- partly to keep memory usage down, but mostly because cuda support for floats is better

float Rs, oldRs, newRs;
float * h_radialPoints;
float * d_radialPoints;
float * d_dipole;
float * d_resultArray;


//we declare both host and device functions here
//__global__ functions run on the device, but can be called from the host
//__device__ functions are invisible to the host
//functions without __prefixes__ are ordinary C++ functions

//this is the kernel that makes the GPU do the integration and return the results
__global__ void integrationKernel(float *dipole, float *resultArray, float * pts);
```

```

//a function to test the interpolator (must run on the device, since the interpolator is on the device)
__global__ void testInterpolator(float *dipole, float *resultArray, float * pts);

//this is a function that handles all the interpolation, on the GPU since that's where we need it!
__device__ float interpolate(float values[dipoleSize], float posn, float * pts);

//this is a function that calculate the scaling speed and writes the relevant info to file
void writeState(float rapidity, float dipole[dipoleSize], float points[dipoleSize*extraZ], float rhs[dipoleSize],
      float pf);

//this is a function to update the Rs cohort at every step, so that we have good lambda calculations when we want
      them
void calcRs(float dipole[dipoleSize], float points[dipoleSize*extraZ]);


//preamble & declarations are complete, so we move on to function definitions
//
      ------------------------------------------------------------------------------------------------------------------//

int main(){

//set up the files for data output, including some description
//this clears their content, so that they can just be appended to in the evolution loop

ofstream dipoleFile;
ofstream speedFile;
ofstream rhsFile;
dipoleFile.open("dipole.dat", ios::out);
speedFile.open("scaling-speed.dat", ios::out);
dipoleFile.open("rhs.dat", ios::out);
dipoleFile << "#this file describes the dipole function S(r) at various rapidities. \n #Rapidity \t Separation \t S
      (r)"<<endl;
//we comment out the first scaling speed in advance, since it does not calculate well due to fencepost issues and
      isn't particularly needed anyway
speedFile << "#this file gives the scaling speed and correlation length R_s at various rapidities. \n #Rapidity \t
      Rs \t lambda"<<endl<<"#";
rhsFile << "#this file describes the RHS of teh BK equation at various rapidities. \n #Rapidity \t Separation \t dS
      /dY"<<endl;
dipoleFile.close();
speedFile.close();
rhsFile.close();

//here we declare a number of pointers and allocate memory to them, either on the device or the host
//recall that h_ and d_ prefixes are used to distinguish host and device variables
//the cudaMalloc function manages memory allocation on the device, but host memory is allocated with malloc as
      normal

h_radialPoints = (float *) malloc(extraZ*dipoleSize*sizeof(float));
cudaMalloc(&d_dipole, dipoleSize*sizeof(float));//the dipole values
cudaMalloc(&d_resultArray, dipoleSize*sizeof(float));//with an entry for each r - not quite dS(r) yet
cudaMalloc(&d_radialPoints, extraZ*dipoleSize*sizeof(float));//those points
float * h_dipole;
h_dipole = (float *) malloc(dipoleSize*sizeof(float));//the dipole values
float * h_resultArray;
h_resultArray = (float *) malloc(dipoleSize*sizeof(float));//with an entry for each r - not quite dS(r) yet, but we
      'll operate on it after copying from the device


//as part of the setup,  we declare variables that describe the thread & block breakdown with which we want the
      kernel to run
//we include thetaSize-1 threads in each block, since each thread corresponds to an angular discretisation
//the grid is two-dimensional, with the first dimension of blocks corresponding to the radial discretisation and
      the second to the various integrals to be performed
dim3 grid(dipoleSize*extraZ, dipoleSize);
dim3 block(thetaSize-1);


//with the computational setup complete, we can move on the more mathematical side of things and set up initial
      conditions

//first we set up the points at which the dipole will be stored and the integral performed -- these are logarithmic
//it makes sense to reuse the first few radial integrator points as storage points for the dipole function
//note that these are NOT the dipole values -- just the points for which we will store the dipole values [r, not S(
      r)]
for (int i=0; i<dipoleSize*extraZ; i++)
```

```
        h_radialPoints[i] = 1e-3*exp(1e-2*(i - dipoleSize/2));


//once we have the points as which to store the dipole, we can calculate the values we want to store
//our IC is a Gaussian, scaled to fit onto the grid we chose above (but its relation to the Rs(Y_0) units doesn't change)
//to this end we calculate a width parameter -- not the literal width, but a related value -- based on the grid
float width = log(1e-6)/pow(h_radialPoints[dipoleSize-1], 2);
for (int i =0; i<dipoleSize; i++)
        h_dipole[i] = exp(width*h_radialPoints[i]*h_radialPoints[i]);

//we calculate the initial correlation length now so that the scaling speed calculations kick in ASAP
Rs=sqrt(log(0.5)/width);//we can try

//we have populated the set of radial points in host memory, but it will also be needed in device memory, so we copy it there too
cudaMemcpy(d_radialPoints, h_radialPoints, dipoleSize*extraZ*sizeof(float), cudaMemcpyHostToDevice);


//we factor everything we can out of the integral calculation, so that it can just be done once at the end this leaves us with a prefactor that we value here
//dS = r^2*timestep*(alpha*Nc/(2pi^2))*integral
//we also need a factor of d\theta, which we factored out of the quadrature rule. this cancels a factor of 2pi
//we only perform a quarter of the integral, but the factor of 4 in the quadrature rule exactly cancels this

float prefactor = 0.5 * dY/thetaSize * (0.217*3/M_PI);


//START OF TESTING SECTION

//once initialsation is complete, there are a number of tests that can be run by uncommenting the appropriate section below
//I would skip this section on a first readthrough of the code
//most of these output to stdout, so you might want to run './a.out > testdata' or similar
//it generally makes sense to set totsteps to 0, since actual evolution is generally not relevant if you're doing these tests
//these tests will need tweaking, but they should be a useful start to any testing regimen

// ----------------------------------------------------------------------------------------------------------------------------


        //TEST ICs
/*
        writeState(0, h_dipole, h_radialPoints);
*/
        //RHS TESTER
/*
        //update dipole and resultArray on device so we can calculate there
        for (int i=0; i<dipoleSize; i++) h_resultArray[i]=0;
        cudaMemcpy(d_resultArray, h_resultArray, dipoleSize*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_dipole, h_dipole, dipoleSize*sizeof(float), cudaMemcpyHostToDevice);
*/
/*
        //call the GPU integration routine
        integrationKernel<<<grid, block>>>(d_dipole, d_resultArray, d_radialPoints);//<<<blocks, threadsPerBlock>>>

        //copy results of integration (in d_resultArray) back to the host
        cudaMemcpy(h_resultArray, d_resultArray, dipoleSize*sizeof(float), cudaMemcpyDeviceToHost);

        for (int i=0; i<dipoleSize; i++) cout<<h_radialPoints[i]<<"\t"<<h_resultArray[i]*prefactor*h_radialPoints[i]*h_radialPoints[i]<<endl;
*/

/*
        //INTERPOLATION TESTER
        cudaMemcpy(d_dipole, h_dipole, dipoleSize*sizeof(float), cudaMemcpyHostToDevice);

        //call the interpolation test routine
        testInterpolator<<<1, dipoleSize>>>(d_dipole, d_resultArray, d_radialPoints);//<<<blocks, threadsPerBlock>>>
        //cudaMemcpy(h_resultArray, d_resultArray, dipoleSize*sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(h_resultArray, d_resultArray, dipoleSize*sizeof(float), cudaMemcpyDeviceToHost);

        for (int i=0; i<dipoleSize; i++) cout<<h_radialPoints[i]<<"\t"<<h_dipole[i]<< "\t"<<(h_radialPoints[i]+h_radialPoints[i+1])/2<<"\t"<<h_resultArray[i]<<endl;
```

```
*/

//  ----------------------------------------------------------------------------------------------------------------------------------------

//END OF TESTING SECTION


//if we aren't running tests, we just need to loop through evolution steps now that initialisation is complete

for (int step=0; step<totSteps; step++)
        {
        //update the correlation length at every step, so that we have data to calculate the scaling speed locally when we need it
        calcRs(h_dipole, h_radialPoints);



        //the GPU needs to be intialised before calculation are carried out
        //the sums must all start at zero and the dipole values must be up to date, so we copy those values across from the device

        for (int i=0; i<dipoleSize; i++) h_resultArray[i]=0;//this is exceedingly hacky, but right now I'm just satisfied that it works

        cudaMemcpy(d_resultArray, h_resultArray, dipoleSize*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_dipole, h_dipole, dipoleSize*sizeof(float), cudaMemcpyHostToDevice);


        //with the GPU set up, we can call the integration routine on the device
        //notice that the kernel reads in parameter from device memory, not host memory
        //the <<<...>>> allows us to assign the number of blocks and threads per block, using the variable we defined earlier

        integrationKernel<<<grid, block>>>(d_dipole, d_resultArray, d_radialPoints);


        //once the integrator has run, copy the results that have been calculated and stored on device memory back to the host
        //if something is going wrong, it tends to crop up here, so proper error checking is useful
        //IME most errors are caused by memory allocation problem. Timeout errors probably mean the GPU needs restarting, because this kernel is quick to execute

        cudaError_t copyReturn = cudaMemcpy(h_resultArray, d_resultArray, dipoleSize*sizeof(float), cudaMemcpyDeviceToHost);
        if (copyReturn) cerr<<"Error copying from GPU device: "<<copyReturn<<"."<<endl;

        //check how far we are through the evolution steps and write to file if appropriate
        //writing at this point lets us output S and the dS/dY it produces together
        if (step%stepsPerWrite==0) writeState(step*dY, h_dipole, h_radialPoints, h_resultArray, pf/dY);

        //once the results are copied across, they can be used to update the dipole
        //arguments for moving this step onto the GPU to enable RK are made in my MSc thesis
        //if you did that, you would only need to copy back to the host when you wanted to write to file, reducing copying time even further (up to Rs calculations)
        //(don't try to write to file from the GPU unless you are willing to think very hard about race conditions. it tends to be unpredicatable)
        for (int i=0; i<dipoleSize; i++)
                {
                //multiply in everything we factored out of the integral this allows an error checking statement on the dS quantity
                float addend =prefactor*h_radialPoints[i]*h_radialPoints[i]*h_resultArray[i];
                //we could also use this space to output the RHS, which might be useful
                //cout<<"Error! dS = "<<addend<<", result = "<< h_resultArray[i]<< " @ " << h_radialPoints[i] <<". Copy return was "<<copyReturn<<"."<<endl;

                //finally, actually add the increment to the function
                h_dipole[i] += addend;
                }//end dipole update loop


        }//end iteration loop

}//end of main
```

```
// -----------------------------------------------------------------------------------------------------------------------------------

//We're done! Apart from the details of all the calculations . . .



__global__ void integrationKernel(float *dipole, float *resultArray, float * pts){
//this is the kernel that makes the GPU do the integration and return the results
//it takes in the dipole array and a place to spit out the results of the integration
//(which depend on the dipole array and some constant stuff that we can precalculate


//each thread needs to know what theta, z, r it is working on
//since z and r values are shared across blocks, they only need to be looked up once per block
//this won't make much difference in the scheme of things, but it's a nice proof of concept

float theta;
__shared__ float z;
__shared__ float r;
theta = M_PI * threadIdx.x/thetaSize;
if (threadIdx.x==0)
{
        z = pts[blockIdx.x];
        r = pts[blockIdx.y];
}//end if master thread

//make sure we're all caught up with the parameters
__syncthreads();

//now we calculate the integrand, given these parameters

//first some set up
float integrand;
float diff = z*z + (r)*(r) - 2*z*(r)*cos(theta);//this is the the square of the vector difference


//skip the cancelling infinity at the origin and anything on the far side of the symmetry line
if ((diff < 1e-18) || (z < 1e-10) ||(z*cos(theta)>r/2) )
        integrand =0;


//if we're in the proper integration region, calculate the integrand
else
        {
        //calculate the value of the integrand at this grid location

        //we have to interpolate one of the relevant dipole values, but can look the others up directly because of the way our grid is structured
        float Sd = interpolate(dipole, sqrt(diff), pts);
        float Sz = dipole[blockIdx.x];
        float Sr = dipole[blockIdx.y];

        //once we have all the components, we stick them together to get the integrand at this grid point
        integrand = ((Sd*Sz -Sr)/(z*diff));

        //now we use out irregular trap quadrature rule (described in the thesis) to weight this appropriately
        if (!(threadIdx.x == 0 || threadIdx.x == thetaSize-1))
                integrand *= 2;
        if (blockIdx.x == 0)
                integrand *= pts[blockIdx.x + 1] - z;
        else if (blockIdx.x == extraZ*dipoleSize -1)
                integrand *= z - pts[blockIdx.x - 1];
        else
                integrand *= pts[blockIdx.x + 1] - pts[blockIdx.x - 1];

        //remember that we have factored everything we possibly can out of the quadrature rule -- we don't even have a d\theta term here, since that's a constant

        }//end else calculate integrand

//we sum the weighted integrand terms to get the integral itself
//(up to factored out pieces, which are large -- so this is orders of magnitude away from the actual result)
//atomicAdd avoids race conditions by making all the thread belonging to one integral take turns accessing memory

atomicAdd(&(resultArray[blockIdx.y]), integrand);
```

```
}//end of integrationKernel

//
     ---------------------------------------------------------------------------------------------------------------------------------------------


__device__ float interpolate(float * values, float posn, float * pts)
{
//this is a function that handles all the interpolation

//first check if we're being asked to extrapolate
//if we are, return an asymptotic value (if that's inappropriate, something has gone very wrong)

if (posn > pts[dipoleSize-1])
        return 0;
if (posn<= pts[0])
        return 1;

//otherwise, look for the stored point closest to the one we're interested in

int i;
for(i=0; i<dipoleSize; i++)
        if (pts[i]>=posn) break;

//we're going to linearly interpolate by drawing a straight line between the points neighbouring the one we want
//we calculate weights based on how far the neighbouring stored points are from the desired point

float normalisation = pts[i]-pts[i-1];
float lowerWeight = (pts[i] - posn)/normalisation;
float upperWeight = (posn - pts[i-1])/normalisation;

//use the weights to linearly interpolate the value at the desired point

return (upperWeight*values[i]+lowerWeight*values[i-1]);

}//end of interpolate

//
     ---------------------------------------------------------------------------------------------------------------------------------------------


void writeState(float rapidity, float dipole[dipoleSize], float points[dipoleSize*extraZ], float rhs[dipoleSize],
     float pf){
//a nice simple subroutine with nothing to do with parallelisation

//open up the files we're outputting to
//we've prepped these in main and may have already output a previous state, so we want to append now

ofstream dipoleFile;
ofstream speedFile;
ofstream rhsFile;
dipoleFile.open("dipole.dat", ios::out | ios::app);
speedFile.open("scaling-speed.dat", ios::out | ios::app);
rhsFile.open("rhs.dat", ios::out | ios::app);

//set the precision of the writes to be reasonably high
//(this gives us enough detail to see the noise -- we don't have a full six sig. fig.s of accuracy!)
dipoleFile.precision(6);
speedFile.precision(6);
rhsFile.precision(6);

//write the dipole & rhs data to file

for (int i=0; i<dipoleSize; i++)
        {
        dipoleFile << rapidity << "\t" << points[i] << "\t" << dipole[i] << endl;
        rhsFile << rapidity << "\t" << points[i] << "\t" << h_resultArray[i]*pf*points[i]*points[i] << endl;
        }
//we include a blank line at the end of the trajectory to simplify plotting later on
dipoleFile << endl;
rhsFile << endl;


//we calculate the scaling speed from the automatically updated Rs values and write it to file
```

```
//we could get fancy and try to avoid the fencepost problem for the initial state, but it doesn’t really change
     much
//in general the scaling speed is one baby time step behind the dipole trajectory, but it’s a very small
     difference and they aren’t directly compared anyway
//we don’t need a newline, since there’s a single scaling speed trajectory for the entire evolution

speedFile << rapidity-dY << "\t" <<Rs << "\t" << -(log(newRs) - log(oldRs))/dY<<endl;

//close the files (until we need them again)

dipoleFile.close();
speedFile.close();
rhsFile.close();

}//end of writeState

//
     ------------------------------------------------------------------------------------------------------------------------------------------


void calcRs(float dipole[dipoleSize], float points[dipoleSize*extraZ])
//this calculates the correlation length by simple linear interpolation
//we keep three values of the correlation length around so the we can do symmetric fintie difference calculations
     for the scaling speed

{
//update the old values
oldRs = Rs;
Rs = newRs;

//calculate the new value

//find the stored point closest to 1/2
//(this is very similar to the interpolation procedure)
int i;
for(i=0; i<dipoleSize; i++)
        if (dipole[i]<=0.5) break;


//we will linearly interpolate the point where S(r) is actually 0.5
//calculate the weights to do so
float upperWeight = (dipole[i-1] - 0.5)/(dipole[i-1]-dipole[i]);
float lowerWeight = (0.5 - dipole[i])/(dipole[i-1]-dipole[i]);

//use the weights to interpolate and update the new Rs value
newRs = (upperWeight*h_radialPoints[i]+lowerWeight*h_radialPoints[i-1]);

}//end of calcRs

//
     ------------------------------------------------------------------------------------------------------------------------------------------


__global__ void testInterpolator(float *dipole, float *resultArray, float * pts)
//a simple GPU kernel to test the interpolator
//puts an interpolator-shifted version of dipole into resultArray

{

int i = threadIdx.x;
resultArray[i] =interpolate(dipole, (pts[i]+pts[i+1])/2, pts);

}//end of testInterpolator
//
     ------------------------------------------------------------------------------------------------------------------------------------------


/*

There’s a dog at the end of the world
with a sad smile and a bow tie,
with a mile-high wagging tail
waving a flag that says
YOU MADE IT!
on one side in bouncing yellow letters, and
```

```
THAT'S ALL FOLKS :(
on the other side, in wistful blue.

--Helen Harvey
        from 'Dog at the End of the World' (March Hamilton Media, 2012)

*/
```

Bibliography

[1] T. Lappi and H. Mäntysaari. “Next-to-leading order Balitsky-Kovchegov equation with resummation”. In: (2016). arXiv: 1601.06598 [hep-ph].

[2] E. Iancu et al. “Resumming double logarithms in the QCD evolution of color dipoles”. In: *Physics Letters B* 744 (2015), pp. 293 –302. ISSN: 0370-2693. DOI: http://dx.doi.org/10.1016/j.physletb.2015.03.068. URL: http://www.sciencedirect.com/science/article/pii/S0370269315002439.

[3] I. Balitsky and G. A. Chirilli. “Next-to-leading order evolution of color dipoles”. In: *Phys. Rev.* D77 (2008), p. 014019. DOI: 10.1103/PhysRevD.77.014019. arXiv: 0710.4330 [hep-ph].

[4] I. Balitsky and A. V. Grabovsky. “NLO evolution of 3-quark Wilson loop operator”. In: *JHEP* 01 (2015), p. 009. DOI: 10.1007/JHEP01(2015)009. arXiv: 1405.0443 [hep-ph].

[5] A. Kovner, M. Lublinsky, and Y. Mulian. “Jalilian-Marian, Iancu, McLerran, Weigert, Leonidov, Kovner evolution at next to leading order”. In: *Phys. Rev.* D89.6 (2014), p. 061704. DOI: 10.1103/PhysRevD.89.061704. arXiv: 1310.0378 [hep-ph].

[6] M. D. Sievert. “Transverse Spin and Classical Gluon Fields: Combining Two Perspectives on Hadronic Structure”. In: (2014). arXiv: 1407.4047 [hep-ph].

[7] S. J. Brodsky et al. “Single-Spin Asymmetries in Semi-inclusive Deep Inelastic Scattering and Drell-Yan Processes”. In: *Phys. Rev.* D88.1 (2013), p. 014032. DOI: 10.1103/PhysRevD.88.014032. arXiv: 1304.5237 [hep-ph].

[8] Y. V. Kovchegov and M. D. Sievert. “A New Mechanism for Generating a Single Transverse Spin Asymmetry”. In: *Phys. Rev.* D86 (2012). [Erratum: Phys. Rev.D86,079906(2012)], p. 034028. DOI: 10.1103/PhysRevD.86.034028, 10.1103/PhysRevD.86.079906. arXiv: 1201.5890 [hep-ph].

[9] Y. V. Kovchegov et al. "Subleading-N(c) corrections in non-linear small-x evolution". In: *Nucl. Phys.* A823 (2009), pp. 47–82. DOI: 10.1016/j.nuclphysa.2009.03.006. arXiv: 0812.3238 [hep-ph].

[10] H. Weigert et al. "JIMWLK: From concepts to observables". In: *POETIC VI, Ecole Polytechnique, Palaiseau, France* (2015).

[11] F. Gelis et al. "The Color Glass Condensate". In: *Ann. Rev. Nucl. Part. Sci.* 60 (2010), pp. 463–489. DOI: 10.1146/annurev.nucl.010909.083629. arXiv: 1002.0333 [hep-ph].

[12] J. Jalilian-Marian et al. "The Intrinsic glue distribution at very small x". In: *Phys. Rev.* D55 (1997), pp. 5414–5428. DOI: 10.1103/PhysRevD.55.5414. arXiv: hep-ph/9606337 [hep-ph].

[13] J. Jalilian-Marian et al. "The BFKL equation from the Wilson renormalization group". In: *Nucl. Phys.* B504 (1997), pp. 415–431. DOI: 10.1016/S0550-3213(97)00440-9. arXiv: hep-ph/9701284 [hep-ph].

[14] J. Jalilian-Marian et al. "The Wilson renormalization group for low x physics: Towards the high density regime". In: *Phys. Rev.* D59 (1998), p. 014014. DOI: 10.1103/PhysRevD.59.014014. arXiv: hep-ph/9706377 [hep-ph].

[15] J. Jalilian-Marian, A. Kovner, and H. Weigert. "The Wilson renormalization group for low x physics: Gluon evolution at finite parton density". In: *Phys. Rev.* D59 (1998), p. 014015. DOI: 10.1103/PhysRevD.59.014015. arXiv: hep-ph/9709432 [hep-ph].

[16] J. Jalilian-Marian et al. "Unitarization of gluon distribution in the doubly logarithmic regime at high density". In: *Phys. Rev.* D59 (1999). [Erratum: Phys. Rev.D59,099903(1999)], p. 034007. DOI: 10.1103/PhysRevD.59.034007,10.1103/PhysRevD.59.099903. arXiv: hep-ph/9807462 [hep-ph].

[17] A. Kovner, J. G. Milhano, and H. Weigert. "Relating different approaches to nonlinear QCD evolution at finite gluon density". In: *Phys. Rev.* D62 (2000), p. 114005. DOI: 10.1103/PhysRevD.62.114005. arXiv: hep-ph/0004014 [hep-ph].

[18] H. Weigert. "Unitarity at small Bjorken x". In: *Nucl. Phys.* A703 (2002), pp. 823–860. DOI: 10.1016/S0375-9474(01)01668-2. arXiv: hep-ph/0004044 [hep-ph].

[19] E. Iancu, A. Leonidov, and L. D. McLerran. "Nonlinear gluon evolution in the color glass condensate. 1." In: *Nucl. Phys.* A692 (2001), pp. 583–645. DOI: 10.1016/S0375-9474(01)00642-X. arXiv: hep-ph/0011241 [hep-ph].

[20] E. Ferreiro et al. "Nonlinear gluon evolution in the color glass condensate. 2." In: *Nucl. Phys.* A703 (2002), pp. 489–538. DOI: 10.1016/S0375-9474(01)01329-X. arXiv: hep-ph/0109115 [hep-ph].

[21] A. H. Mueller and J.-w. Qiu. "Gluon Recombination and Shadowing at Small Values of x". In: *Nucl. Phys.* B268 (1986), p. 427. DOI: 10.1016/0550-3213(86)90164-1.

[22] A. H. Mueller. "Soft gluons in the infinite momentum wave function and the BFKL pomeron". In: *Nucl. Phys.* B415 (1994), pp. 373–385. DOI: 10.1016/0550-3213(94)90116-3.

[23] A. H. Mueller and B. Patel. "Single and double BFKL pomeron exchange and a dipole picture of high-energy hard processes". In: *Nucl. Phys.* B425 (1994), pp. 471–488. DOI: 10.1016/0550-3213(94)90284-4. arXiv: hep-ph/9403256 [hep-ph].

[24] L. D. McLerran and R. Venugopalan. "Computing quark and gluon distribution functions for very large nuclei". In: *Phys. Rev.* D49 (1994), pp. 2233–2241. DOI: 10.1103/PhysRevD.49.2233. arXiv: hep-ph/9309289 [hep-ph].

[25] L. D. McLerran and R. Venugopalan. "Gluon distribution functions for very large nuclei at small transverse momentum". In: *Phys. Rev.* D49 (1994), pp. 3352–3355. DOI: 10.1103/PhysRevD.49.3352. arXiv: hep-ph/9311205 [hep-ph].

[26] L. D. McLerran and R. Venugopalan. "Green's functions in the color field of a large nucleus". In: *Phys. Rev.* D50 (1994), pp. 2225–2233. DOI: 10.1103/PhysRevD.50.2225. arXiv: hep-ph/9402335 [hep-ph].

[27] A. Ayala et al. "The Gluon propagator in nonAbelian Weizsacker-Williams fields". In: *Phys. Rev.* D52 (1995), pp. 2935–2943. DOI: 10.1103/PhysRevD.52.2935. arXiv: hep-ph/9501324 [hep-ph].

[28] A. Kovner, L. D. McLerran, and H. Weigert. "Gluon production from nonAbelian Weizsacker-Williams fields in nucleus-nucleus collisions". In: *Phys. Rev.* D52 (1995), pp. 6231–6237. DOI: 10.1103/PhysRevD.52.6231. arXiv: hep-ph/9502289 [hep-ph].

[29] I. Balitsky. “Operator expansion for high-energy scattering”. In: *Nucl. Phys.* B463 (1996), pp. 99–160. DOI: 10.1016/0550-3213(95)00638-9. arXiv: hep-ph/9509348 [hep-ph].

[30] Y. V. Kovchegov. “Quantum structure of the nonAbelian Weizsacker-Williams field for a very large nucleus”. In: *Phys. Rev.* D55 (1997), pp. 5445–5455. DOI: 10.1103/PhysRevD.55.5445. arXiv: hep-ph/9701229 [hep-ph].

[31] A. H. Mueller. “Parton saturation at small x and in large nuclei”. In: *Nucl. Phys.* B558 (1999), pp. 285–303. DOI: 10.1016/S0550-3213(99)00394-6. arXiv: hep-ph/9904404 [hep-ph].

[32] Y. V. Kovchegov. “Unitarization of the BFKL pomeron on a nucleus”. In: *Phys. Rev.* D61 (2000), p. 074018. DOI: 10.1103/PhysRevD.61.074018. arXiv: hep-ph/9905214 [hep-ph].

[33] A. Kovner and J. G. Milhano. “Vector potential versus color charge density in low x evolution”. In: *Phys. Rev.* D61 (2000), p. 014012. DOI: 10.1103/PhysRevD.61.014012. arXiv: hep-ph/9904420 [hep-ph].

[34] H. Weigert. “Evolution at small x(bj): The Color glass condensate”. In: *Prog. Part. Nucl. Phys.* 55 (2005), pp. 461–565. DOI: 10.1016/j.ppnp.2005.01.029. arXiv: hep-ph/0501087 [hep-ph].

[35] C. Marquet and H. Weigert. “New observables to test the Color Glass Condensate beyond the large-N_c limit”. In: *Nucl. Phys.* A843 (2010), pp. 68–97. DOI: 10.1016/j.nuclphysa.2010.05.056. arXiv: 1003.0813 [hep-ph].

[36] M. E. Peskin and D. V. Schroeder. *An introduction to quantum field theory.* Advanced book program. Autre tirage : 1997. Boulder (Colo.): Westview Press Reading (Mass.), 1995. ISBN: 0-201-50934-2. URL: http://opac.inria.fr/record=b1131978.

[37] J. Kuokkanen, K. Rummukainen, and H. Weigert. “HERA-Data in the Light of Small x Evolution with State of the Art NLO Input”. In: *Nucl. Phys.* A875 (2012), pp. 29–93. DOI: 10.1016/j.nuclphysa.2011.10.006. arXiv: 1108.1867 [hep-ph].

[38] J. L. Albacete. “Resummation of double collinear logs in BK evolution versus HERA data”. In: (2015). arXiv: 1507.07120 [hep-ph].

[39] P. Cvitanovic. “Group theory for Feynman diagrams in non-Abelian gauge theories”. In: *Phys. Rev.* D14 (1976), pp. 1536–1553. DOI: 10.1103/PhysRevD.14.1536.

[40] P. Cvitanovic. *Birdtracks, Lie's and Exceptional Groups*. Princeton, New Jersey: Princeton University Press, 2008.

[41] E. Iancu, K. Itakura, and L. McLerran. "Geometric scaling above the saturation scale". In: *Nucl. Phys.* A708 (2002), pp. 327–352. DOI: 10.1016/S0375-9474(02)01010-2. arXiv: hep-ph/0203137 [hep-ph].

[42] NVIDIA Corporation. *CUDA C Programming Guide*. 2015.

[43] A. Dutt and V. Rokhlin. "Fast Fourier Transforms for Nonequispaced Data". In: *SIAM J. Sci. Comput.* 14.6 (June 1993), pp. 1368–1393. URL: http://dx.doi.org/10.1137/0914081.

[44] L. Greengard and J. Lee. "Accelerating the Nonuniform Fast Fourier Transform". In: *SIAM Review* 46.3 (2004), pp. 443–454. URL: http://dx.doi.org/10.1137/S003614450343200X.

[45] M. Galassi et al. *GNU Scientific Library Reference Manual (3rd Ed)*. 2013. URL: http://www.gnu.org/software/gsl/.

[46] T. Lappi and H. Mäntysaari. "Direct numerical solution of the coordinate space Balitsky-Kovchegov equation at next-to-leading order". In: *Phys. Rev. D* 91 (7 2015), p. 074016. DOI: 10.1103/PhysRevD.91.074016. URL: http://link.aps.org/doi/10.1103/PhysRevD.91.074016.