SLAC-R-646

Computer Simulation of Electron Positron Annihilation Processes*

Yue Chen

Stanford Linear Accelerator Center
Stanford University
Stanford, CA 94309

SLAC-Report-646
August 2003

Prepared for the Department of Energy
under contract number DE-AC03-76SF00515

Printed in the United States of America. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

* Ph.D. thesis, Stanford University, Stanford, CA 94309.

COMPUTER SIMULATION OF ELECTRON POSITRON ANNIHILATION PROCESSES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF PHYSICS
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yue Chen
August 2003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Peskin
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Lance Dixon

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Joanne Hewett

Approved for the University Committee on Graduate Studies.

Abstract

With the launching of the Next Linear Collider coming closer and closer, there is a pressing need for physicists to develop a fully–integrated computer simulation of e^+e^- anniilation process at center–of–mass energy of 1TeV. A simulation program acts as the template for future experiments. Either new physics will be discovered, or current theoretical uncertainties will shrink due to more accurate higher–order radiative correction calculations. The existence of an efficient and accurate simulation will help us understand the new data and validate (or veto) some of the theoretical models developed to explain new physics. It should handle well interfaces between different sectors of physics, e.g., interactions happening at parton levels well above the QCD scale which are described by perturbative QCD, and interactions happening at much lower energy scale, which combine partons into hadrons. Also it should achieve competitive speed in real time when the complexity of the simulation increases.

This thesis contributes some tools that will be useful for the development of such simulation programs. We begin our study by the development of a new Monte Carlo algorithm intended to perform efficiently in selecting weight-1 events when multiple parameter dimensions are strongly correlated. The algorithm first seeks to model the peaks of the distribution by features, adapting these features to the function using the EM algorithm. The representation of the distribution provided by these features is then improved using the VEGAS algorithm for the Monte Carlo integration. The two strategies mesh neatly into an effective multi–channel adaptive representation.

We then present a new algorithm for the simulation of parton shower processes in high energy QCD. We want to find an algorithm which is free of negative weights, produces its output as a set of exclusive events, and whose total rate exactly matches the full Feynman amplitude calculation. Our strategy is to create the whole QCD

shower as a tree structure generated by a multiple Poisson process. Working with the whole shower allows us to include correlations between gluon emissions from different sources. QCD destructive interference is controlled by the implementation of "angular–ordering", as in the HERWIG Monte Carlo program. We discuss methods for systematic improvement of the approach to include higher order QCD effects.

Acknowledgements

Firstly, my deepest thanks go to my advisor Michael Peskin. During my four years of study with him, I have not only benefited from his tremendous knowledge in physics, but also from his endless new ideas and strategies in tackling the same problem. He has lead me to the result of persistent work, and his sole tenacity and optimism will remain to be my guiding force in facing my future challenges. Above all this, he has shown me a wonderful personality of patience, modesty and objectiveness. His gracefulness has uplifted my pressure from being a graduate student into sheer enlightenment of making contributions to the eternal journey of knowledge acquisition.

Secondly, my gratitude and love go to my husband Boris, who has given up his opportunity to return to Germany — his homeland, and chosen to stay here accompanying me to finish my study. He has helped to establish my emotional stability, which lead to my single-minded concentration in the research topic. Some of my work benefited from the inspiration originated from my frequent conversations with him on the work.

When I look back, my sincere gratefulness goes to the SLAC Theory Group, which through its calsualness yet exactness, has provided me with a healthy environment to think and to mature. I thank Prof. Lance Dixon, JoAnne Hewett, Tom Rizzo, Helen Quinn, Stan Brodsky, Shamit Kachru, Marvin Weinstein, Paul Tsai and other senior members, for caring to share their knowledge and wisdom with me, for helping me grow to be fair and think in a wide angle. For my work in MAVEN, my discussions with Prof. Stanislaw Jadach were tremendously helpful and insightful. During his two months short visit to SLAC, he has shared with me a lot of valuable experience from his working in the field for so many years. He has given me a unique and long-lasting impression of a humbly quiet yet indomitable scholar towards his goal

of perfection.

I also thank theory group's past and current postdocs — Kirill Melnikov, Thomas Becher, Gudrun Hiller and all others — for teaching me physics and sharing their physics and personal experience with me, which is a necessary ingredient of my life in SLAC. I thank its graduate students — Frank Petrellio, Yasaman Farzan and all others — for all the help that I need in finishing up my thesis and for a lot of lengthy discussions which provide me with the insight into many critical issues in the modern world. My gratitude also extends to my closest friends — Jun Ren, Boaz Nash, Bruno and Hiroko Strulovici. Without their friendship and accompanionship I couldn't have survived through my graduate school years. The width and depth that my fellow friends have reached is the precious soil for my nourishment.

Finally I thank my family for urging me to start my adventure in America. Particular thanks go to my father, who through my upbringing, has infused me with the principle of freedom, equality and justice, who keeps encouraging me to dream and strive to be a woman who can be resourceful and determined enough to break the gender difference still existing in people's ideology.

The work in Chapter 2 is based upon the preprint "MAVEN: A New Algorithm for Multi–Grid Adaptive Simulation" to appear, written in collaboration with Michael Peskin. The work in Chapter 3 is based upon the preprint "A New Algorithm for Parton Shower and Matching with Finite-Order QCD Calculation" to appear, also written in collaboration with Michael Peskin.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

2 MAVEN: A New Algorithm for Multi–Grid Adaptive Simulation **7**

2.1 Introduction 7
2.2 Basics of event selection 8
2.3 Feature analysis 11
2.4 Improvement of features with VEGAS 15
2.5 Results for two–dimensional PDFs 17
2.6 Results for realistic PDFs 22
2.7 Conclusions 23

3 Parton Showering Simulation with Perturbative QCD **25**

3.1 Introduction 25
3.2 Theory of QCD Parton Showering 26
3.3 PYTHIA and HERWIG 31
3.4 Algorithmic construction of Parton Shower 33
3.5 Leading-Order perturbative QCD treatments 36
3.5.1 Angular Ordering 37
3.5.2 Next–to–Leading–Order Reweighting 37
3.6 Physical Simulation of QCD at Q $= m_Z$ 39
3.6.1 Jet Rate at O(α_s) 39

3.6.2 Thrust Simulation at O(α_s) 40
3.7 Conclusion and Outlook 41

4 Conclusions 47

A Notes for the use of the MAVEN Software 49
A.1 Bayesian statistics applied to neuron evolution 49
A.2 Structure of the software distribution 51
A.3 Monte Carlo interface 52
A.4 Initialization and control of `MavenMC` 52
A.5 Use of vector and matrix classes 55
A.6 Implementation of Feature Adaptation 55
A.7 Implementation of Grid Adpatation in Each Feature 57
A.8 A More Systematic Estimation of `Maxweight` 60

B Derivations of Shower Simulation Equations 64
B.1 Monte Carlo based on Poisson Distribution 64
B.2 Suppressing back–to–back Events 66
B.3 Calculation of the Reweighting Factor 68
B.4 Accompanying software 70
B.4.1 Variable documentation of a shower event 72
B.4.2 Construction of a shower event 74
B.4.3 Boosting of a shower event 74
B.4.4 Angular Ordering 80
B.4.5 Jet Algorithms 80
B.4.6 reweighting 80

Bibliography 89

List of Tables

2.1 Comparison of VEGAS and MAVEN on 2–dimensional functions . . . 18
2.2 Comparison of VEGAS and MAVEN on double Gaussians 22
2.3 Comparison of VEGAS and MAVEN on e^+e^- annihilation processes . 24

A.1 Test run on controlling `Maxweight` 62

List of Figures

2.1 Vegas optimization example 10
2.2 Transformation map of one feature 13
2.3 Adapted features for Bird Function 19
2.4 Adapted configuration for $e^+e^- \rightarrow \mu^+\mu^-$ 21

3.1 Collinear Emission 27
3.2 Multiple Emission 30
3.3 Gluon Showering 33
3.4 Gluon Branching 34
3.5 Dynamic check 36
3.6 Clustering of Jets 39
3.7 Jetrates compared with PYTHIA 42
3.8 Jetrates compared with HERWIG 43
3.9 Zoomed-in version of k_T Jetrate 44
3.10 Thrust Plot 45
3.11 special configuration 46

A.1 Public methods of the class `MonteCarlo`. 53
A.2 Public methods of the class `MavenMC`. 54
A.3 prepare function of the class `MavenMC`. 56
A.4 `adaptstep` function of the class `MavenMC`. 57
A.5 `update` function of the class `MavenMC`. 58
A.6 grid adaption of the function `prepare`. 59
A.7 selectively update one feature using one data point 60
A.8 point selection mechanism 61

B.1 suppression mechanism . . . 66
B.2 Gluon radiation . . . 68
B.3 Public methods of the class `process`. . . . 71
B.4 Public methods of the class `shower`. . . . 73
B.5 Initialization . . . 75
B.6 gluon insertion . . . 76
B.7 gluon instantiation . . . 77
B.8 color ordering . . . 78
B.9 boost . . . 80
B.10 angular ordering . . . 81
B.11 cross section . . . 82
B.12 k_T algorithm . . . 83
B.13 JADE algorithm . . . 84
B.14 E0 algorithm . . . 85
B.15 jet counting algorithm . . . 86
B.16 clustering algorithm . . . 87
B.17 reweighting algorithm . . . 88

Chapter 1

Introduction

With the launching of the Next Linear Collider coming closer and closer, there is a pressing need for physicists to develop a fully–integrated computer simulation of e^+e^- anniilation process at center–of–mass energy of 1TeV. A simulation program acts as the template for future experiments. Either new physics will be discovered, or current theoretical uncertainties will shrink due to more accurate higher–order radiative correction calculations. The existence of an efficient and accurate simulation will help us understand the new data and validate (or veto) some of the theoretical models developed to explain new physics. It should handle well interfaces between different sectors of physics, e.g., interactions happening at parton levels well above the QCD scale which are described by perturbative QCD, and interactions happening at much lower energy scale, which combine partons into hadrons. Also it should achieve competitive speed in real time when the complexity of the simulation increases.

The core of a simulation program for particle physics events at high energy is a program that generates paton-level events. In this thesis, we will present some new methods for parton-level event generator. We will develop a new algorithm in Chapter 2 which has higher efficiency in generating weight-1 events when some parameter dimensions exhibit strong correlation. In Chapter 3 we will apply the philosophy of Chapter 2 to write a parton shower simulation program with matching to finite order perturbative QCD results.

In Chapter 2, we discuss a general problem at the heart of constructing an event generator. We are given a process whose possible outcomes are predicted by a known

probability distribution. To simulate such a process, one selects points in the phase space ('events') according to the given probability. For example, in scattering processes, the differential cross sections are known, and the scattering process can be simulated by selecting events with probability proportional to this cross section. This issue is trivial in one dimension, but typically one must work with a cross section formula that depends on many phase space variables and is highly peaked in specific regions of the multi–dimensional space.

We refer to this problem as that of 'event sampling'. A number of different methods for event sampling have been proposed in the high–energy physics literature. One strategy is to base a method for event sampling on a general–purpose method for multi–dimensional integration. For example, one might begin from Lepage's very effective multi–dimensional integration algorithm VEGAS [1,2], which works in a fixed coordinate system in the N–dimensional space and adapts a grid with cell boundaries parallel to the coordinate axes. The adapted grid provides a simple model of the probability distribution, and this model can be used as a first approximation for sampling the distribution. This strategy is used in Kawabata's general–purpose sampling algorithm BASES and SPRING [3,4], which is used for event simulations in high–energy reactions.

VEGAS is well–known to have difficulty with probability distributions that have peaks that run diagonally to the coordinate axes. To overcome this problem, it is useful to choose new variables so that each singularity is parallel to some axis. However, this is often impossible to do. As an example, consider a process at center of mass energy $E_{\rm CM}$ that leads to a final state with three massless particles. The cross section might be resonant at a specific value of any of the mass combinations

$$m_{12}^2 = (p_1 + p_2)^2 \ , \qquad m_{23}^2 = (p_2 + p_3)^2 \ , \qquad m_{31}^2 = (p_3 + p_1)^2 \ . \tag{1.1}$$

However, the three masses are constrained by the relation $m_{12}^2 + m_{23}^2 + m_{31}^2 = E_{\rm CM}^2$, so one set of resonances is always on the diagonal relative to the others. An effective solution to this problem is multi–channel integration, writing a model for the probability distribution which is a sum of terms, each of which peaks in a different relevant kinematic variable. The EXCALIBUR event generator of Berends, Kleiss,

and Pittau adapts the weights of the various terms to generate the best representation of the original distribution [5,6]. Systems such as GRACE and MADGRAPH that automatically generate cross section formulae from Feynman diagrams have also been programmed to pick out the relevant phase space variable sets that should appear in different channels [7,8]. Ohl has developed this approach one step further by associating a VEGAS grid with the variables of each channel and adapting the set of grid along with the relative weights of channels. This algorithm, called VAMP [9], provides the simulation engine for the event generator WHIZARD [10,11].

The difficulty with the usual approach to multichannel integration is that one must know the correct variables sets to use for a given problem and the mappings that connect them. It would be preferable, if it were possible, to simply write down the probability distribution and ask the integration program itself to find the best variables to use. Perhaps for problems in which one constructs the cross section as an explicit function, this might be thought as an unnecessary luxury. But even in this case, there are applications for which the burden should be put on the integrator to find the best coordinates. For example, in the design of an event generator for high–energy physics applications using object–oriented programming, it is natural to construct separate classes to produce the different elements of a scattering process — the initial–state beam distributions, the core scattering cross sections, and the decay distributions for each final particle. It is advantageous to make this structure truly modular, so that different classes might have different authors who need not care about the internal structure of the other elements. But then the convolution of the modules will inevitably lead to peaks in the cross section that depend simultaneously on integration variables from several modules. To represent such peaks well, it is necessary either to look inside the modules, violating the 'encapsulation' of their methods, or to ask the overall event selection program to find the best optimization.

We have been engaged in constructing a general–purpose event generator for reactions of high–energy electrons, positrons, and photons, called `pandora` [12]. The program is written in `C++` with a modular construction of the kind just discussed. Currently, the event selection algorithm in `pandora` is VEGAS, but many processes exhibit very low efficiency for event sampling due to conflicts of the type just discussed. It would be advantageous to find a multichannel event selection algorithm

compatible with our modular approach. This motivates some of the specific design choices made in Chapter 2. Typically, we are interested in integrals with 5–40 variables, of which 2–10 form combinations that have peaking structure.

Our algorithm is inspired by neural networks [13] and began as an attempt to adapt a neural network to carry out the task of event selection. The final algorithm can be thought of as a two–layer neural network, with the features as the neurons in the first layer and the VEGAS grids as the neurons in the second layer. We call the final algorithm MAVEN: Multichannel Adaptive integration with Vegas Enhancement.

A very different approach to adaptive multidimensional event selection is the FOAM algorithm recently proposed by Jadach [14,15]. The idea of this algorithm is to recursively divide the integration region into smaller cells until the integrand is roughly constant over each cell. The algorithm has an advantage that one is guaranteed to obtain an accurate representation of the integrand, but it also has the difficulty that this exacts a very large price in function calls and in the size of the data structure that it requires.

In Chapter 3, we discuss another issue that is important for simulations involving strong-interacting particles. Quantum field theory predictions are based on perturbation theory, which works with finite observables of elementary particle processes. However, even in quantum electrodynamics (QED), it is typical that many hard photons are radiated in high–energy processes. In quantum chromodynamics (QCD), high–energy reactions of quarks entail the radiation of many gluons, and nonperturbative dynamics then converts the state of quarks and gluons to one of mesons and baryons. Thus, there are two stages of evolution that require separate treatment.

We will be concerned with the simulation of multiple gluon radiation in e^+e^- annihilation to hadrons. This is the first segment of the evolution described above for the case of QCD. Our methods will also suggest generalization to problems of simulating initial– and final–state radiation in QED and the weak interactions. There are of course highly developed and carefully tuned codes — in particular, PYTHIA [16–19] and HERWIG [20–22] — that simulate QCD showers with gluon radiation and also include the stage of transition from quarks and gluons to hadrons. These codes have been used as the basis for extensive, and extremely sucessful, testing of

QCD predictions against the data on final states in e^+e^- annihilation [23,24]. So, why is a new code needed, and what can it hope to improve?

Existing algorithms for the generation of QCD showers are based on the idea of independent evolution and radiation from the various partons produced in the collision. This idea comes from the analysis of the most collinear regions of the radiation pattern, in which, for example, the successive gluon emissions from each quark line can be viewed as an independent Poisson process. The parameters of this process are given, for the extreme collinear limit, by the Altarelli–Parisi equations [25]. It is not difficult to write a prescription that includes wide–angle gluon emission that reproduces the known result in the collinear region and generalizes it to the rest of phase space. However, such prescriptions are always, to some extent, ad hoc. It would be excellent if there were a strategy to tune these prescriptions systematically to the results of QCD perturbation theory in higher orders.

A first step in this direction was the introduction into the HERWIG event generator of a restriction to "angular ordering" in parton shower evolution [23]. Here, one takes account of the angular relation of radiated gluons to quarks and gluons radiated at previous stages and removes gluons radiated into regions suppressed by QCD destructive interference.

Recently, in a very beautiful series of papers [26], Frixione and Webber improved this idea by showing how to construct a QCD shower simulation that could be systematically adjusted to agree to order α_s with the result of a next–to–leading order QCD computation. Remarkably, the correction factors written by Frixione and Webber contain no soft or collinear singularities; these are absorbed naturally by the equations of the leading–order QCD shower evolution. However, the Frixione–Webber prescription still requires that configurations with little gluon radiation are produced with negative weights, so that cancellations between position and negative weight configurations are carried through the whole process of Monte Carlo event generation. Thus, we feel that there is room for improvement of this prescription. We hope to achieve this by taking a more integral view of QCD shower. As in the Frixione–Webber analysis, soft and collinear singularities cancel in this comparision. However, in our approach, we can view the modification as a consistent reweighting with positive weights, so that the simulation code can be arranged to produce final

events with uniform relative weight $+1$.

Chapter 2

MAVEN: A New Algorithm for Multi–Grid Adaptive Simulation

2.1 Introduction

In this chapter, we are engaged in the core of a simulation program for particle physics event generation. A useful problem to solve is : How to improve the efficiency of current event generators?

Our approach to multichannel event selection begins by representing the peaks of the given probability distribution by features of a canonical form. The locations, shapes, and relative weights of the features are chosen initially at some random values and then adapted to fit the probability distribution. We have found that the Expectation Maximization (EM) algorithm [27,13] is a very effective method for adapting these features. Our algorithm also uses features of `C++` to allow features to be created and destroyed according to their importance.

One can think of one feature as one local coordinate. The adapted combination of such local coordinates (feature map) does a good job of representing the peaks of the probability distribution, but it often fails to account correctly for the regions away from the peaks. To ameliorate this problem, we adjust the shapes of the features by using the VEGAS algorithm. That is, we construct each feature so that it explicitly is a coordinate system for the integration. Each feature is treated as one channel of a multi–channel integrator, which means the total integration is a weighted sum of all

the channels. The construction of the feature through the EM algorithm chooses an appropriate set of coordinate axes for this part of the function. We can then adjust the shape of the feature by selecting points in the feature using an adapted grid as in VEGAS. We call the final algorithm MAVEN: Multichannel Adaptive Integration with Vegas Enhancement.

In this chapter, we will present our algorithm as follows: Section 2 will present our general framework for event selection algorithms and review some aspects of the VEGAS algorithm. Section 3 will discuss our algorithm for representation of a probability distribution by features. Section 4 will discuss the enhancement of this algorithm for event selection by the inclusion of VEGAS grids. Section 5 will apply our algorithm to some two–dimensional and other simple probability distributions. Section 6 will describe our experience using MAVEN in practical simulation problems in high–energy physics described by the `pandora` event generator. Section 7 gives our conclusions.

We have submitted with the eprint version of this chapter a `C++` program that implements the MAVEN algorithm and some example programs to test it. We discuss the use of this software in the Appendix. The software also includes a general parent class for the use of event selection algorithms in `C++` code.

2.2 Basics of event selection

We begin by defining our basic framework for event selection. Let x be an $N-$dimensional vector whose components satisfy $0 < x^i < 1$. Let $f(x)$ be a positive function of x. Define

$$\mathcal{I} = \int d^N x f(x) \ . \tag{2.1}$$

Then $f(x)/\mathcal{I}$ is a normalized probability distribution function (PDF). Our problem is to efficiently select points x with this probability distribution. In this chapter, we will always take x to be an element of the unit cube, $[0,1]^N$.

Let $p(x)$ be a positive function that approximates $f(x)$. We will construct $p(x)$ so that it is normalized and so that there is a definite algorithm for choosing points with probability $p(x)$. We will refer to $p(x)$ as our 'model' for $f(x)$. Write the integral in

(2.1) as

$$\mathcal{I} = \int d^N x \, f(x) = \int d^N x \, p(x) \, \frac{f(x)}{p(x)} \; . \tag{2.2}$$

Define the weight of x as

$$w(x) = \frac{f(x)}{p(x)} \; . \tag{2.3}$$

Let W be an upper bound for $w(x)$ (the 'maxweight'). Now we can choose points according to $f(x)$ from the distribution of $p(x)$ by the 'hit–or–miss' algorithm: Select a point x_a with the PDF $p(x)$, compute $w(x_a)$, and retain the point x_a with probability $w(x_a)/W$. The points retained have the probability distribution of the PDF derived from $f(x)$.

The 'hit–or–miss' algorithm is extremely unsophisticated. Yet we prefer it for our purposes to other possible Monte Carlo methods such as the Metropolis algorithm [28]. The reason is that we would like to provide our event selection routine as a black box to users of our event generator. The 'hit–or–miss' algorithm is guaranteed to eventually produce the correct probability distribution. Its difficulty is that it might do this extremely inefficiently. The efficiency of the algorithm is controlled by the quality of the model $p(x)$. If $f(x)/p(x)$ is never large, the event selection will be efficient. However, if $p(x)$ underrepresents $f(x)$ even in a small region, the maxweight will be much larger than the weight of typical points and so the efficiency of event selection will be low. Our choice is to put our effort into constructing a $p(x)$ that follows $f(x)$ as closely as possible.

The VEGAS algorithm gives a simple model for $p(x)$: For each dimension i, $1 \leq i \leq N$, divide the interval [0,1] into N_g segments at the points x^i_a. We set $x^i_0 = 0$, $x^i_{N_g} = 1$. Choose x^i_a from a random cell, and then randomly within the cell. This gives the model

$$p(x) = \prod_i \frac{1}{x^i_{a_i} - x^i_{a_i - 1}} \equiv 1/V(x) \tag{2.4}$$

where a_i is the cell in the ith dimension that contains x and $V(x)$ is the volume of the N–dimensional cell that contains x.

We now adapt the grid to meet an appropriate criterion. In Lepage's original

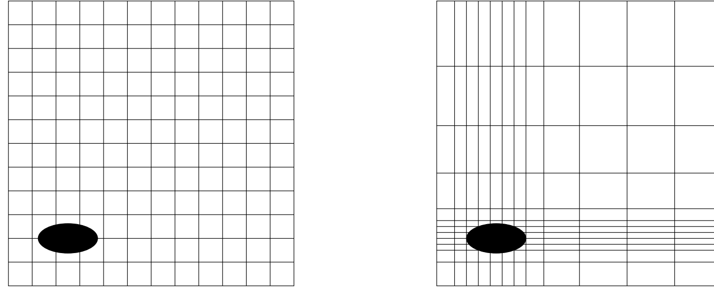


Figure 2.1: Example of optimization of a VEGAS grid for a PDF with one peak, indicated by the dark ellipse.

construction, the grid was adjusted in such a way as to minimize the variance

$$\sigma^2 = \int d^N x p(x) \left| \frac{f(x)}{p(x)} - I \right|^2 . \tag{2.5}$$

In addition, to obtain the best estimate of the integral, stratified sampling was used to produce an ensemble of values x_a [1,2]. Our goal is to construct the most efficient selection procedure for single events. In pursuit of this goal, we record, for each dimension, the maximum value of the weight $w(x)$ obtained in each cell and adjust the grid to equalize these maximum weights as much as possible over the grid.

An example of the use of the VEGAS algorithm is shown in Fig. 2.1. Evolution from the grid on the left to the grid on the right equalizes the maximum weight as much as possible for the cells in each row and in each column. Notice the small volumes in the upper left and lower right corners, which probably imply that the weight is underestimated and the event selection efficiency is low in these regions.

It is not difficult to devise PDFs that give uniformly low efficiency. For example. a PDF that has a peak of uniform height at the values of (x^1, x^2) such that $x^1+x^2 = 1$ gains no advantage from readjustment of a grid aligned with x^1 and x^2. On the other hand, such a peak would be treated very well by a grid aligned with the coordinates $(x^1 \pm x^2)$. It would be attractive to improve the VEGAS algorithm in such a way as to allow the algorithm to find the best axes for its grid.

The VEGAS algorithm also has important advantages that we should not overlook as we attempt to generalize it. First, the grid adaptation is very rapid, requiring only

about 100,000 function calls in our typical high–energy physics applications. Event selection is also very rapid. On a Sun2200 workstation, the adaptation step requires only a few tenths of a second, and after adaptation a point can be chosen on the grid in 10 μsec. This gives a reasonable rate for generating unweighted events even if the efficiency of the hit–or–miss selection is 1% or even 0.1%. Finally, the time required for adaptation has a very mild dependence on the number of dimensions N of the parameter space, growing more slowly than linearly with N as long as the new variables added do not have compex peaking structure.

2.3 Feature analysis

As an alternative to modelling $f(x)$ with a grid, we might model $f(x)$ as a sum of component functions by writing

$$p(x) = \sum_a p_a p(x|A_a) \ , \tag{2.6}$$

where p_a is a probability ($\sum_a p_a = 1$) and $p(x|A)$ is a PDF of a fixed functional form that depends on parameters A. As an example, $p(x|A)$ might be a Gaussian distribution for which A parametrizes the mean and the covariance matrix. In our algorithm, we will make another choice for $p(x|A)$, explained below. We will refer to the components in this equation as 'features', indexed over $a = 1, \ldots, n_f$.

An effective method for adapting the parameters of the features in (2.6) is the Expectation Maximization (EM) algorithm [13]: We first produce an ensemble of weighted points for each component. Assume that there is a method for selecting random points according to the PDF $p(x|A)$. With probability p_b, choose the distribution $a = b$ and select a point x from the PDF. Then add x to the ensemble for each of the features. For the feature $a = c$, assign x the weight

$$w_c(x) = \frac{f(x)}{p(x)} \cdot \frac{p_c p(x|A_c)}{p(x)} \ . \tag{2.7}$$

That is, reweight x to have the correct weight for the function we wish to model, and then divide this weight among the n_f features according to the contribution that

each feature makes to $p(x)$. For each feature, we can now compute expectation values according to

$$\langle F(x)\rangle_c = \sum_x w_c(x)F(x)/\sum_x w_c(x) \tag{2.8}$$

Then assign new probabilities for the features

$$p_c = \sum_x w_c(x)/\sum_{x,a} w_a(x) \tag{2.9}$$

and assign new parameters so that the mean and covariance matrix of $p(x|A_c)$ lie at the mean of the ensemble $\langle x\rangle_c$ and the covariance matrix of the ensemble. After several iterations, features chosen initially with arbitrary positions and orientations move to positions and orientations at which they form a good approximation to the PDF derived from $f(x)$.

For a reason that will become clear in the next section, we choose our features to be coordinate systems on the unit cube. We choose the following form for the coordinate transformation: Consider the mapping between $[0,1]$ and the real line

$$x^i = g(z^i) = \frac{1}{1+\exp(-z^i)} \ , \qquad z^i = g^{-1}(x^i) = \log\frac{x^i}{1-x^i} \ . \tag{2.10}$$

We will use these mappings component by component to transform a point x in $[0,1]$ to a point z in R^N. Represent these transformations as

$$x = g(z) \ , \qquad z = g^{-1}(x) \ . \tag{2.11}$$

Now define the form of the feature to be given by the mapping $[0,1]^N \to [0,1]^N$

$$G(x|A,B) = g(Az+B) \ , \quad \text{with} \quad z = g^{-1}(x) \ , \tag{2.12}$$

where A is a positive $N \times N$ matrix and B is an N–dimensional vector. For each feature a, a uniform distribution of x_a over $[0,1]^N$ gives a distribution for $x_* = G(x_a|A_a,B_a)$ that is centered at $x_* = g(B_a)$ and is concentrated in a small volume around this point if the matrix A_a has small eigenvalues. We give a graphical view

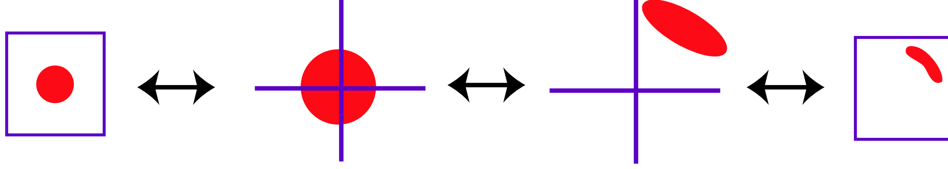


Figure 2.2: The transformation (2.12) from $[0,1]^N$ to $[0,1]^N$ that defines one feature in (2.6).

of the transformation $G(x_a|A_a, B_a)$ in Fig. 2.2. Since

$$\int dx_a = \int dx_* \left| \frac{\partial x}{\partial x_*} \right| \ , \tag{2.13}$$

the feature $p(x|A, B)$ is the inverse Jacobian of the transformation,

$$p(x_*|A_a, B_a) = \left| \frac{\partial x_a}{\partial x_*} \right| = \prod_i \frac{x_a^i(1-x_a^i)}{x_*^i(1-x_*^i)} \frac{1}{\det A_a} \ . \tag{2.14}$$

Note that, to compute $p(x_*) = \sum_c p_c p(x_*|A_c, B_c)$ it is necessary to map the chosen point x_* back to the original x variable in each of the coordinate systems. This is also a property of Ohl's VAMP algorithm; the formalism of these mappings is discussed further in [9].

To evolve the features (2.14), we work in the z_* space. After taking some data and creating the weighted ensembles described above, we set the new probabilities for the feature c equal to (2.9). We assign the new value of B_c by

$$B_c^i = \left\langle z_*^i \right\rangle \ . \tag{2.15}$$

To assign the new value of A_c, we first write

$$\mathcal{A}_c^{ij} = (A_c A_c^T)^{ij} = \left\langle z_*^i z_*^j \right\rangle - \left\langle z_c^i \right\rangle \left\langle z_c^j \right\rangle \ . \tag{2.16}$$

The matrix $\mathcal{A}_c$ can be diagonalized in the form

$$\mathcal{A}_c = R\sigma^2 R^T \tag{2.17}$$

where R is a rotation in N dimensions and σ^2 is a diagonal matrix with positive eigevalues σ_i^2. Then we set the new A_c to

$$A_c = R\sigma \ . \tag{2.18}$$

This means that the standard coordinates in x_c map into the principal axes of the covariance matrix in (2.16). Note that the fact that the algorithm includes a diagonalization allows the sampling of the PDF itself to find the best local set of coordinate axes.

In the implementation of the algorithm, we store the rotation matrix R and the eigenvalues of σ for each feature c. The determinant in (2.14) is readily computed as $\prod_i \sigma_i$. To choose a point, we pick a feature c with probability p_c, select x_c randomly in the unit cube, map to z_c, and then map to z_* using (2.12),

$$z_* = R_c \sigma_c z_c + B_c \ . \tag{2.19}$$

The chosen z_* can be mapped to x_*. This value can also be mapped backwards through the other coordinate systems to produce the corresponding values

$$z_b = \sigma_b^{-1} R_b^T (z_* - B_b) \ . \tag{2.20}$$

These can then be converted to the x_b. As we have already noted, all of the x_b associated with z_* are needed to compute $p(x_*)$.

To implement the algorithm, it is also necessary to specify the number of features and the initial positions and sizes of the features. We choose the initial positions of features randomly in the unit cube, with the initial size fixed to $\sigma_i = 0.3$ for all i. The user specifies a number `nfpd`, and the program starts by laying down $N \cdot$ `nfpd` features. (That is, we take the initial number of feature to be proportional to the dimensionality of the phase space.) We also include a 0th feature that is a uniform mapping of the unit cube into itself, with p_0 fixed at the value 10%. This insures that the representation $p(x)$ cannot collapse onto some peaks of the original function $f(x)$ while ignoring other peaks of $f(x)$ that were not close to the random positions of the original features. For the functions that arise in our application, each

adaptation step can be done with about 20,000 function calls. After each adaptation step, features with probability $p_c < 0.01$ are deleted, and a new set of $N \cdot \texttt{nfpd}$ features in random positions is added. We observe that after about ten steps, the total number of features equilibrates. We then continue to adapt with the same set of features for another five steps. The result is a fair representation of the target PDF with 5–20 features.

2.4 Improvement of features with VEGAS

While the feature map does represent the peaks of the the PDF based on $f(x)$, it is not necessarily a good representation from the point of view of event generation. The hit–or–miss algorithm corrects for the difference between the target PDF and the model $p(x)$ in a manner that is efficient only if the model provided by $p(x)$ is uniformly of reasonable quality. If the weight (2.3) is exceptionally large in some small region, the value of the weight in this region will determine the maxweight W. The efficiency of the hit–or–miss event selection will be then be small, of the order of I/W. We might say that, while the feature map is determined by the bird's eye view of the target function $f(x)$, the efficiency of the event selection is determined by the worm's eye view.

If we choose $p(x)$ to be the feature map described in the previous section, the points of maximum weight are typically points in the regions between features. What we need to do now is to stretch the individual features into these regions in such a way as to pick up the regions where the weight is large.

One possible way to do this is to generate second–order features for each feature. However, in our experience, this leads to modifications that are too highly restricted to the vicinity of the feature, and also produces a complex and time–consuming procedure for event selection. Instead, we have chosen to improve the features using VEGAS.

It is at this point that it becomes important that the features are constructed from mappings $G(x|A,B)$ from $[0,1]^N$ to $[0,1]^N$. In the previous section, we generated ponts x_* by choosing points randomly in the original cube. As an alternative, we can attach a grid to each mapping and choose points in each cube according to the

VEGAS algorithm. The model $p(x)$ is made somewhat more complex. We now have

$$p(x_*) = \sum_{c=0}^{n_f} p_c g_c(x_*) \ , \tag{2.21}$$

with

$$p(x_*|A_c, B_c) = \left|\frac{\partial x_a}{\partial x_*}\right| = \prod_i \left(\frac{x_c^i(1-x_c^i)}{x_*^i(1-x_*^i)}\frac{1}{\sigma_c^i}\right) \cdot \frac{1}{V_C(x_c)} \tag{2.22}$$

where the last term is defined by (2.4) using the grid of feature c. Note that the sum over features runs from 0 to n_f; as explained above, the 0th feature is a uniform mapping from $[0,1]^N$ to $[0,1]^N$.

The final model is thus a multigrid implementation of VEGAS, similar to the VAMP algorithm [9] except that the multiple coordinate systems are not chosen *a priori* but rather are fixed by sampling. A key requirement for multigrid integration is that the coordinate systems should be chosen so that the coordinate axes line up with the peaks of the target function $f(x)$. We will see in the examples that the feature map does succeed, to a great extent, in producing sets of principal axes lined up along the ridges of $f(x)$.

There are many possible choices to be made in fixing the criterion for updating the grids. We have found the best results with the following procedure: We first fix the probabilities, locations, and shapes of the features using the algorithm described in the previous section. We then adapt the grids in a separate stage. For each chosen point, we determine which coordinate system c, excluding $c = 0$, makes the largest contribution to the weight (2.21). We then assign the weight $w(x_*) = f(x_*)/p(x_*)$ to the cell of the grid c that contains x_c, and also to the cell of the 0th feature that contains x_*. We then adapt the grids, as explained in Section 2, to equalize as much as possible the maximum weight in each grid. The updating is done in 8 steps, which in our practical examples are given about 10,000 function calls each. Features of low probability have relative few points assigned to them, so, for each grid, we do not update until 1000 points have been assigned to that grid.

This completes our discussion of the MAVEN algorithm. However, there is a modification of this algorithm that we have found useful for working with functions on phase spaces of high dimensionality, $N \geq 10$. In Section 2, we recognized the

useful feature of the VEGAS algorithm for event selection that its performance deteriorates not significantly as the dimensionality increases. MAVEN does not have this property. When N is larger, it is difficult to collect enough data to determine the covariance matrix in (2.16) well enough. More importantly, the matrix multiplications with the $N \times N$ matrix R make event selection prohibitively time–consuming. To deal with this problem, it is very useful to follow [3,4] in separating the coordinates into 'fast' dimensions with strong peaking and 'slow' dimensions with more moderate behavior. Our implementation of the MAVEN algorithm requires the user to mark the fast dimensions explicitly. (The procedure for doing this is described in the usage notes in the Appendix.) Then the MAVEN algorithm is applied in the fast dimensions, while simple VEGAS event selection — with a single grid — is used in the slow dimensions. With this modification, one can add as many slow dimensions as one wishes to the target function $f(x)$ without substantially slowing down the event selection.

2.5 Results for two–dimensional PDFs

To understand how the MAVEN algorithm works, it is useful to visualize its behavior for two–dimensional functions. In this section, we will present three examples, two chosen artificially to present problems for VEGAS, and one chosen as a realistic example from high–energy physics. We will also present results for Lepage's example of a double Gaussian on the diagonal in a large–dimensional phase space [1].

The two artificially chosen examples are the 'V function'

$$f_V(x_1, x_2) = \frac{b}{b^2 + (x_1 - 0.4x_2 - 0.1)^2} + \frac{b}{b^2 + (2x_1 + x_2 - 1.5)^2} \tag{2.23}$$

and the 'bird function'

$$f_B(x_1, x_2) = \frac{b}{b^2 + (x_1^2 + x_2^2 - 1)^2} + \frac{b}{b^2 + (x_1^2 + x_2^2 - \frac{1}{2})^2} + \frac{b}{b^2 + (x_1 - x_2)^2} \ . \tag{2.24}$$

In both cases, we take $b = 10^{-3}$ for the results given below.

The comparison of VEGAS and MAVEN for these two functions is shown in

	integral	W/I	σ/I	efficiency
V function				
VEGAS	4.71	541.8	8.0	1 : 1077.7
MAVEN	4.70	124.3	3.1	1 : 253.3
bird function				
VEGAS	10.274	216.833	6.881	1 : 452.8
MAVEN	10.26	37.89	1.408	1 : 73.9
$e^+e^- \to \mu^+\mu^-$				
VEGAS	1072.24	66.31	2.31	1 : 130.5
MAVEN	1072.93	30.45	0.70	1 : 63.6

Table 2.1: Comparison of the VEGAS and MAVEN event selection algorithms for three two–dimensional functions describe in the text. For both algorithms, the adaptation used 10^6 function calls. The columns give the final value of the integral I, the maxweight normalized to I, the square root of the variance (2.5) normalized to I, the selection efficiency for unweighted events, i.e., 1 event get selected out of 1077.7 for the first row

Table 2.1. MAVEN makes a significant improvement in all aspects of the function modelling, and in particular in the efficiency of the final Monte Carlo event selection. It should be noted that VEGAS still wins out in terms of the time needed to select one event. This is because the model $p(x)$ is more complex in the case of MAVEN, while the function $f(x)$ itself is trivial to compute. In practical examples, the time for one function call should be dominated by the time to compute $f(x)$ and $p(x)$, and the time required for both VEGAS and MAVEN should be inversely proportional to the efficiency with the same constant of proportionality.

In Fig. 2.3, we show a visualization of the working of MAVEN for the example of the bird function. In Fig. 2.3(a), we show the positions of the 13 features: For each feature, the interior of the square $0.2 < x_1, x_2 < 0.8$ is mapped into the interior of the box shown. The light lines are the features given by the algorithm of Section 3; the dark lines are the final features after VEGAS adaptation. In Fig. 2.3(b), we show points generated by these features before hit–or–miss event selection. The points with lighter shading have $w(x)/I > 3$. The problematic points with $w(x)/I > 10$ are shown as stars.

To the simple model functions in (2.23), (2.24), we add the two–dimensional PDF that arises in the problem of computing the cross section for the high–energy physics

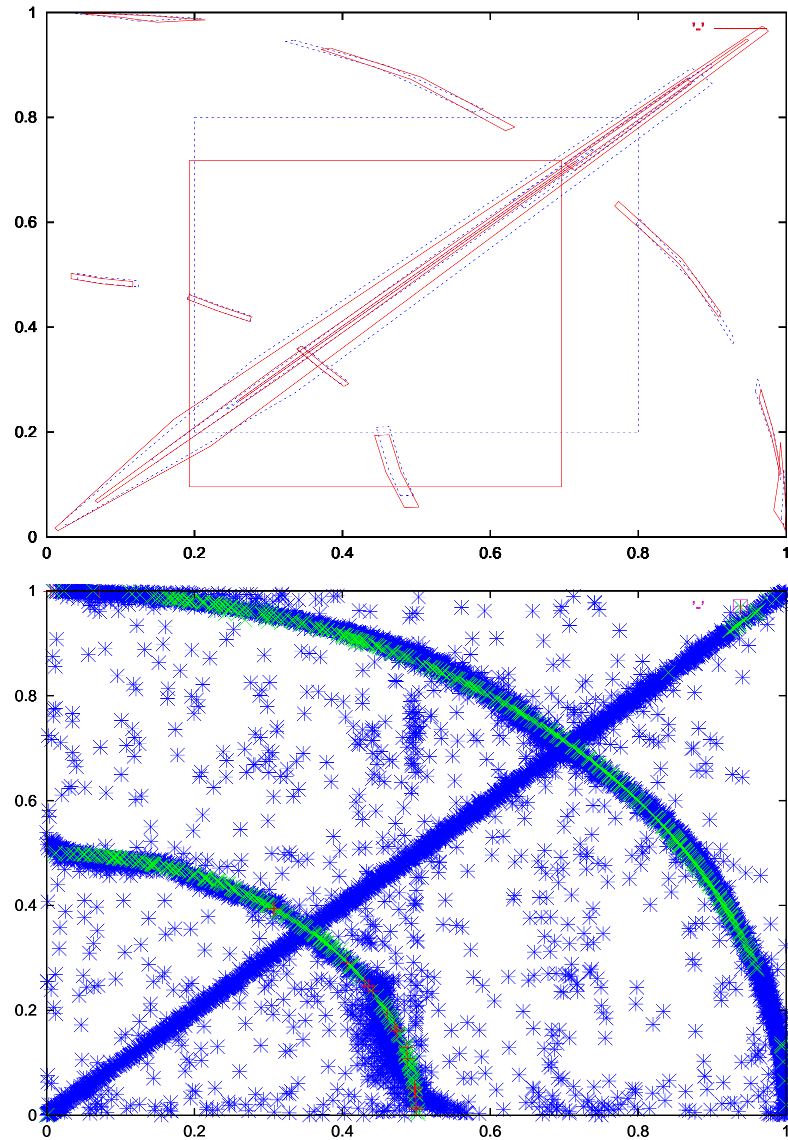


Figure 2.3: Adapted configuration of MAVEN for the bird function (2.24): (a.) features of MAVEN before (light, dotted) and after (dark) VEGAS grid adaptation; (b) points chosen by the model $p(x)$, and their associated weights: light crosses have $w(x)/I > 3$, dark stars have $w(x)/I > 10$.

process $e^+e^- \to \mu^+\mu^-$ at a center of mass energy $E_{\rm CM} = 500$ GeV, allowing the initial electron and positron to radiate photons. The function $f(x)$ is the integrand in the cross section formula

$$\sigma = \int_0^1 dz_1 \int_0^1 dz_2 F(z_1)F(z_2)\sigma_0(z_1, z_2, s) \ . \tag{2.25}$$

In this formula, $\sigma_0(s)$ is the zeroth–order cross section at a lower center of mass energy $E_{\rm CM} = \sqrt{s}$, z_i is the fracton of the electron or positron energy that enters the actually annihilation rather than being radiated away, $F(z)$ is the 'structure function' for giving the probability of radiating a given fraction $(1-z)$, from [29]. Since $F(z)$ is strongly peaked at $z = 1$ and also has a long tail extending to $z = 0$, it is useful to map z into a new variable x in which the structure function is relatively flat. Our choice of $x(z)$ has a discontinuous derivative at $x = \frac{1}{2}$. Flattening $F(z)$ does not make the integration (2.25) trivial because $\sigma_0(s)$ has a strong peak at the Z boson mass, $s = m_Z^2$, and m_Z is well below 500 GeV. This creates a peak in $f(x)$ that is not aligned with the x_1, x_2 coordinate axes.

The comparison of VEGAS and MAVEN for this PDF is shown in Table 2.1. The features and event weights for the MAVEN adaptation are shown in Fig. 2.4. The discontinuous behaviour seen at $x_1 = \frac{1}{2}$ and at $x_2 = \frac{1}{2}$ correctly reflects the discontinuity of the mapping from z to x. The peak at $s = m_Z^2$ is seen in the upper left and lower right; it is the features in these regions that are doing the work of improving the event selection model. Note that the integrand of (2.25) is symmetric under $x_1 \leftrightarrow x_2$. However, the features chosen by MAVEN begin at random positions, and so do not necessarily respect this symmetry. The generated points are more symmetric and the final accepted points are symmetric to a high degree.

As a final example, we study the behavior of MAVEN for the double Gaussian function suggested as a test function in [1] and [14,15]:

$$f(x_i) = \frac{1}{2}\frac{1}{(\sqrt{\pi}a)^N}\left\{\exp\left[-\sum_{i=1}^{N}\frac{(x_i-\frac{1}{3})^2}{a^2}\right] + \exp\left[-\sum_{i=1}^{N}\frac{(x_i-\frac{2}{3})^2}{a^2}\right]\right\} \ . \tag{2.26}$$

The properties of the event selection with VEGAS and MAVEN are compared in Table 2.2. Unlike the previous three examples, here we see that as dimensionality of

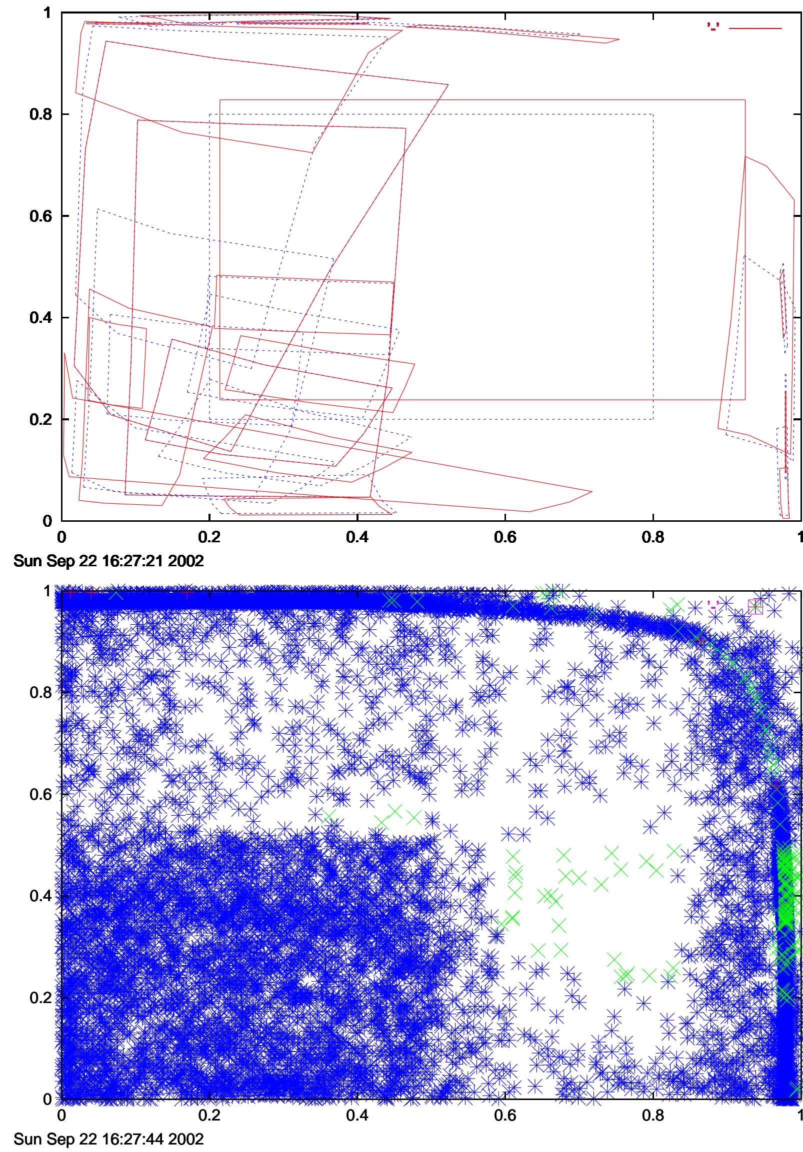


Figure 2.4: Adapted configuration of MAVEN for the function (2.25) associated with the physics process $e^+e^- \rightarrow \mu^+\mu^-$: (a.) features of MAVEN before (light, dotted) and after (dark) VEGAS grid adaptation; (b) points chosen by the model $p(x)$, and their associated weights: light crosses have $w(x)/I > 3$, dark stars have $w(x)/I > 10$.

	integral	W/I	σ/I	efficiency	t(select)
$N = 3$					
VEGAS	0.031	6.7	1.8	1:14.2	0.25
MAVEN	0.031	3.3	0.4	1:6.8	1.06
$N = 6$					
VEGAS	1.75e-4	276.0	6.9	1 : 555.1	5.7
MAVEN	1.74e-4	21.3	0.5	1 : 43.3	7.5
$N = 8$					
VEGAS	5.3e-6	5146.5	6.9	1 : 9993.4	93.0
MAVEN	5.5e-6	68.6	0.5	1 : 135.3	8.4

Table 2.2: Comparison of the VEGAS and MAVEN event selection algorithms for the double Gaussian (2.26). For both algorithms, the adaptation used 10^6 function calls. t(select) is the amount of time it takes each method to generate 1,000 events after adaption, in terms of second. The program is being run on Sun 2200 workstation. The other columns are as in Table 2.1.

strongest correlation increases, Maven starts to show its advantage in the speed of event selection. It starts to suggest to us that Maven's advantage lies in handling problems with large degree of complexity.

2.6 Results for realistic PDFs

As we pointed out in the introduction, we were motivated to consider the problem of Monte Carlo event selection in order to improve the performance of the event selection in a particular application, the simulation of scattering events in high energy electron–positron collisions. We have constructed a general–purpose simulation program, `pandora`, which is written in `C++` and uses the modular construction that is natural in that language to separate the treatment of initial beams, particle reactions, and particle decays into different `C++` classes linked by well–defined interfaces. The general structure of `pandora` is described in [12]. Some physics studies that have made practical use of this program can be found in [30,31]. The current version of the `pandora` code and documentation can be found at [32].

The number of dimensions of the phase space for `pandora` integrands is typically very large. The cross section formula given by `pandora` includes integration over three

variables per initial beam (parametrizing intrinsic energy spread from the acclerator, energy spread due to 'beamstrahlung' [33], and energy spread due to initial state radiation), plus explicit integration over all relevant final state phase space variables. To use MAVEN with `pandora`, we have found it necessary to isolate the 4–8 variables with the strongest peaking and do feature analysis only in these variables.

Table 2.3 shows representative results for four processes whose integral poses increasing difficulty. First, we consider e^+e^- annihilation to top quark pairs, adding in all other important beam effects and also integrating by Monte Carlo over the final–state angular distribution. The next process is the production of W boson pairs from e^+e^-. This adds the complication of a strongly forward–peaked angular distribution: the peaking gets more obvious in terms of Compton scattering shown as the last interaction. The third example, Higgs boson production in W^+W^- fusion, is a process with e^+e^- annihilation to a three–body final state, with strong forward peaking in the distributions of the final neutrino and antineutrino. In these examples, we see the progressive advantage of MAVEN as the physics becomes more complex. We also see the relative selection times for MAVEN and VEGAS come into line with the relative efficiencies, as we expect for more complex target integrands.

2.7 Conclusions

In this chapter, we have introduced a new algorithm for Monte Carlo selection of points under a probability distribution. The algorithm, called MAVEN (Multichannel Adaptive Integration with Vegas ENhancement), uses feature analysis to locate the major peaks of the target probability distribution, then applies the VEGAS algorithm to smooth the boundary regions around these peaks. We have shown that the algorithm is effective both for model problems and for problems of practical interest in up to 10 dimensions.

The MAVEN algorithm might be the basis of an effective multi–dimensional integrator. For this application, one should add stratified sampling [2] to the algorithm as we have proposed it. This might be an interesting extension.

There is another direction that we also wish to pursue. The addition of VEGAS to the feature analysis gives a noticeable gain over pure feature analysis in terms of

	integral	W/I	σ/I	efficiency	t(adapt)	t(select)
$e^+e^- \to t\bar{t}$: $N_M = 2$; $N = 19$						
VEGAS	721.1	197.8	1.9	1:395	6.3	1124.6
MAVEN	722.0	75.3	3.0	1:147	12.9	947.6
$e^+e^- \to W^+W^-$: $N_M = 2$; $N = 13$						
VEGAS	14514.3	334.8	4.1	1 : 670	5.3	723.2
MAVEN	14522.2	113.5	9.3	1 : 230	12.2	504.3
$e^+e^- \to \nu\bar{\nu}h^0$: $N_M = 6$; $N = 26$						
VEGAS	107.5	4480.1	21.8	1 : 9290	64.5	667.7
MAVEN	106.9	878.0	8.0	1 : 1663	418.0	525.0

	W_1	W_2	W_3	t(adapt)	t(select)
$e^-\gamma \to e^-\gamma, e^-Z, \nu W^-$: $N_M = 2,2,2$; $N = 5,8,8$					
VEGAS	1351.5	743.6	2270.5	6.2	2383.1
MAVEN	98.2	320.7	63.0	27.1	731.9

Table 2.3: Comparison of the VEGAS and MAVEN event selection algorithms for various reactions in e^+e^- annihilation at high energy, as simulated by the `pandora` event generator. For each process, we give the total number N of dimensions of the phase space and the number N_M of fast dimensions to which feature analysis is applied. For both algorithms, the adaptation used 10^6 function calls. The columns are as in Table 2.1. The integral is a physical cross section, in femtobarns.

decreasing the variance, but it is less impressive in decreasing the maximum weight. In the example we have discussed, the points of maximum weight are pushed into corners and crevices but not completely eliminated. Perhaps there is a better adaptation criterion for the VEGAS grids that would help in this task.

For problems of very high dimensionality, it is possible to treat the fastest dimensions with feature analysis while treating more mild variables with VEGAS. This gives an event selection algorithm that is more powerful than VEGAS but scales well with dimensionality. We feel that this approach can be easily incorporated into practical event generation programs for high–energy physics and other domains.

Chapter 3

Parton Showering Simulation with Perturbative QCD

3.1 Introduction

In the previous chapter, we have presented a method for choosing weight-1 events from a probability distribution based on techniques of Monte Carlo integral in many variables. Correlations between the variables and even strong peaking can be handled by an appropriate Monte Carlo integrator. It is very useful to represent physics processes as multi-dimensional integrals to which these techniques can be applied. For the problem of QCD showers, however, this global approach has another advantage. Since we work with the entire integral at once, we can look at the QCD shower as a whole and know exactly what cross section we are assigning to each exclusive multi–parton configuration. This gives us the ability to adjust each multi–parton cross section in a systematic way to agree with exact finite–order results from QCD.

In this chapter, we apply this philosophy to the simplest problem of QCD event generation, the generation of parton showers for e^+e^- annihilation to hadrons. Our general approach is to write the cross sections for fully exclusive production of quark–antiquark–gluon final states as multiple integrals. For a maximum number N of gluons produced, we work with a 5N dimensional integral. From the integration variables, we produce final states that naturally and exactly respect 4–momentum conservation and cover all of the available phase space. Starting from this definite

initial situation, we can then set out prescriptions for reweighting the cross sections to agree with finite–order QCD calculations.

The algorithm first seeks to describe a parton shower using the Altarelli–Parisi Equations up to the leading–logarithm accuracy: the infra–red divergence cancellation is handled by Sudakov suppression factors. A strategy of "angular–ordering" is implemented at this stage also, so that gluons radiated into regions suppressed by QCD destructive interference are strictly prohibited. Each individual event subsequently gets reweighted according to its full cross section calculated by the exact Feynman diagrams. The purpose of reweighting is that we want to match our exclusive cross sections with QCD perturbative results of inclusive cross sections. Combining the above mentioned two steps together we have an algorithm which agrees fairly well with finite–order QCD calculations and achieves a competitive speed in real time.

The detailed analysis of this chapter is organized as follows: In Section 2, we outline the theory of QCD showers and the summation of collinear radiation by the Altarelli–Parisi equation. In Section 3, we briefly review the two state-of-art QCD Monte Carlo algorithms — PYTHIA and HERWIG. In Section 4, we explain our algorithm for building up the shower configurations, including generation of the shower tree as a set of multiply–running Poisson processes and our prescription for the treatment of 4–momentum conservation. In Section 5, we discuss methods for angular ordering and reweighting the events produced by this algorithm to agree with the exact order α_s 3–jet cross sections from QCD. In Section 6, we present some comparisions of the algorithm to QCD to experimental data at the Z mass. Section 7 presents some conclusions.

3.2 Theory of QCD Parton Showering

Parton showering exists both in QCD and QED due to the interaction between fermion and gauge boson. An incoming or outgoing quark or electron can radiate a gluon or photon, as shown in Fig. 3.1. The amplitude for the process diverges in the soft and collinear limits, leading to the proliferation of processes with multiple boson emission.

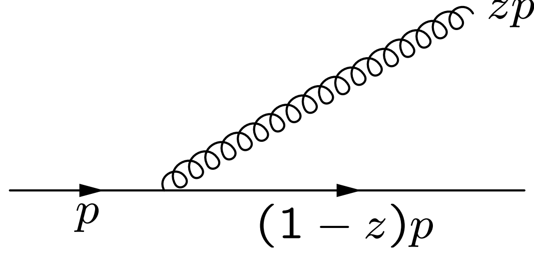


Figure 3.1: The vertex for emission of a collinear photon or gluon. A gluon comes off of a parton carrying fraction z of its longitudinal momentum.

In the case of QED, including the effects of pair creation, Gribov and Lipatov [34] showed that the leading logarithms from collinear singularities can be summed to all orders in QED coupling constant α. They are called evolution equations, they take the form

$$\begin{aligned}
\frac{d}{d\log Q} f_\gamma(x,Q) &= \frac{\alpha}{\pi}\int_x^1 \frac{dz}{z}\{P_{\gamma\leftarrow e}(z)[f_e(\frac{x}{z},Q)+f_{\bar{e}}(\frac{x}{z},Q)] \\
&\quad +P_{\gamma\leftarrow\gamma}(z)f_\gamma(\frac{x}{z},Q)\}, \\
\frac{d}{d\log Q} f_e(x,Q) &= \frac{\alpha}{\pi}\int_x^1 \frac{dz}{z}\{P_{e\leftarrow e}(z)f_e(\frac{x}{z},Q)+P_{e\leftarrow\gamma}(z)f_\gamma(\frac{x}{z},Q)\}, \\
\frac{d}{d\log Q} f_{\bar{e}}(x,Q) &= \frac{\alpha}{\pi}\int_x^1 \frac{dz}{z}\{P_{e\leftarrow e}(z)f_{\bar{e}}(\frac{x}{z},Q)+P_{e\leftarrow\gamma}(z)f_\gamma(\frac{x}{z},Q)\}.
\end{aligned} \tag{3.1}$$

$f_e(x,Q)$ is defined as the distribution function for an electron relevant to a given momentum tranfer Q. The splitting functions are given by

$$\begin{aligned}
P_{e\leftarrow e}(z) &= \frac{1+z^2}{(1-z)_+}+\frac{3}{2}\delta(1-z), \\
P_{e\leftarrow\gamma}(z) &= \frac{1+(1-z)^2}{z}, \\
P_{\gamma\leftarrow e}(z) &= z^2+(1-z)^2, \\
P_{\gamma\leftarrow\gamma}(z) &= -\frac{2}{3}\delta(1-z).
\end{aligned} \tag{3.2}$$

To obtain the distribution function for a physical positron or photon, we should

integrate these equations with initial conditions

$$f_e(x,Q) = \delta(1-x), \qquad f_{\overline{e}}(x,Q) = 0, \qquad f_\gamma(x,Q) = 0, \tag{3.3}$$

at $Q = m_e$. The solutions to these equations are used as in eq.(3.4) to compute cross sections involving processes induced by electrons, positrons, photons that involve large momentum transfer.

$$\begin{aligned} \sigma(e^- X \to e^- + n\gamma + Y) &= \int_0^1 dx f_\gamma(x,Q)\sigma(\gamma X \to Y), \\ \sigma(e^- X \to n\gamma + Y) &= \int_0^1 dx f_e(x,Q)\sigma(e^- X \to Y). \end{aligned} \tag{3.4}$$

We carry the same analysis into high energy perturbative QCD, taking care of the non–Abelian gauge group effects and the running coupling constant. The parton evolution equations are presented as Altarelli–Parisi equations:

$$\begin{aligned} \frac{d\, f_g(x,Q)}{d\log Q} &= \frac{\alpha_s(Q^2)}{\pi}\int_x^1 \frac{dz}{z} \\ & \left\{ P_{qg}(z)\sum_f \left[f_f\left(\frac{x}{z},Q\right) + f_{\overline{f}}\left(\frac{x}{z},Q\right)\right] + P_{gg}(z) f_g\left(\frac{x}{z},Q\right)\right\} \\ \frac{d\, f_f(x,Q)}{d\log Q} &= \frac{\alpha_s(Q^2)}{\pi}\int_x^1 \frac{dz}{z}\left\{ P_{qq}(z) f_f\left(\frac{x}{z},Q\right) + P_{gq}(z) f_g\left(\frac{x}{z},Q\right)\right\} \\ \frac{d\, f_{\overline{f}}(x,Q)}{d\log Q} &= \frac{\alpha_s(Q^2)}{\pi}\int_x^1 \frac{dz}{z}\left\{ P_{qq}(z) f_{\overline{f}}\left(\frac{x}{z},Q\right) + P_{gq}(z) f_g\left(\frac{x}{z},Q\right)\right\} \end{aligned} \tag{3.5}$$

with their corresponding splitting funtion:

$$P_{qg}(z) = \frac{4}{3}\frac{1+(1-z)^2}{z} \tag{3.6}$$

$$\begin{aligned} P_{gg}(z) &= 6\left[\frac{1-z}{z} + z(1-z) + \frac{z}{(1-z)_+} + \right. \\ & \left. \frac{11}{12}\delta(1-z) - \frac{n_f}{18}\delta(1-z)\right] \end{aligned} \tag{3.7}$$

$$\begin{aligned} P_{gq}(z) &= \frac{1}{2}\left[z^2 + (1-z)^2\right] \\ P_{qq}(z) &= \frac{4}{3}\left[\frac{1+z^2}{(1-z)_+} + \frac{3}{2}\delta(1-z)\right] \end{aligned}$$

The algorithm to be described here will generate final states, containing one pair of $q, \overline{q}$ with multiple gluons. We argue that it suffices to simulate the gluon structure function $f_g(x, Q)$, because it corresponds to the leading order in $1/N$, where N is the number of colors. Therefore, we will be mainly concerned with the generation of the gluon distribution. We will generate this distribution as a Poisson process. The algorithm will naturally respect the conservation of momentum and quark number. In (3.5), these conservation laws are guaranteed by the + distribution and the δ function in (3.7). Our approach will not use these constructions but, rather, will work with a probability-conserving algorithm.

We present the algorithm first for multiple gluon radiating from a quark line. The quark to gluon splitting cross section is:

$$\int \frac{dQ}{Q}\frac{\alpha_s(Q)}{\pi}\int \mathrm{d}z\frac{4}{3}\frac{1+(1-z)^2}{z} \tag{3.8}$$

As we can tell, this integral exhibits an infrared divergence, which physically indicates an arbitrarily large number of soft and collinear gluons radiated off the quark. However, if one adds up all the Feynman diagrams corresponding to real and virtual emissions, this infra–red divergence will cancel order by order, this is illustrated by the KLN Theorem [35], on the condition that there is no interactions between final and initial states. This condition is usually satisfied by infrared–safe observables. In the leading–logarithmic (LL) approximation, this resummation can be exponentiated [25]. The exponent is called the Sudakov factor. For a multiple gluon emission off of a single quark as being drawn in Fig. 3.2, its LL cross section is

$$\begin{aligned}\int d\sigma \;=\; & \left[\int_\Lambda^{\sqrt{s}}\frac{dQ_1}{Q_1}\frac{\alpha_s(Q_1)}{\pi}\int_\epsilon^1 dz_1\frac{4}{3}\frac{1+(1-z_1)^2}{z_1}\right] \\ & \left[\int_\Lambda^{Q_1}\frac{dQ_2}{Q_2}\frac{\alpha_s(Q_2)}{\pi}\int_\epsilon^1 dz_2\frac{4}{3}\frac{1+(1-z_2)^2}{z_2}\right]\cdots\times e^{-S}\end{aligned} \tag{3.9}$$

where

$$S \;=\; \left[\int_\Lambda^{\sqrt{s}}\frac{dQ}{Q}\frac{\alpha_s(Q)}{\pi}\int_\epsilon^1 dz\frac{4}{3}\frac{1+(1-z)^2}{z}\right] \tag{3.10}$$

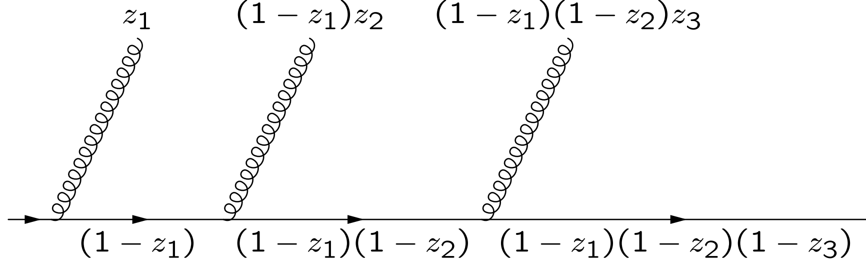


Figure 3.2: Multiple emissions of gluons off the quark line.

Summing over the number of gluons , the total integral

$$\int \mathrm{d}\sigma = 1.$$

Due to the QCD running coupling constant effect, using the 1–loop β function,

$$\alpha_s(Q) = \frac{\alpha_s(M_Z)}{1 + \frac{b_0 \alpha_s(M_Z)}{2\pi} \log \frac{Q}{M_Z}}$$

If we use a transformation

$$t = \frac{2}{b_0} \log \left(1 + \frac{b_0 \alpha_s(M_Z)}{2\pi} \log \frac{Q}{M_Z} \right), \tag{3.11}$$

the Q generation — $\frac{dQ}{Q} \frac{\alpha_s(Q)}{\pi}$ is tranformed into a random generation of t — dt in its perturbative QCD range. Similarly, using the transformation

$$z = e^{-\xi}, \tag{3.12}$$

on z, the very much peaked distribution $\frac{dz}{z} f(z)$ becomes a relatively flat distribution $d\xi f(z(\xi)) |\frac{dz}{d\xi}|$. The Sudakov factor now becomes

$$S = \int \mathrm{d}t \int \mathrm{d}\xi \frac{4}{3}\left(1 + (1 - e^{-\xi})^2\right) \tag{3.13}$$

We can now treat this multiple–gluon emission as a Poisson process. The Sudakov form factor is its exponent. The normalization of the multiple Poisson process and the corresponding event generation mechanism are presented in the appendix.

3.3 PYTHIA and HERWIG

The ideas of the previous section are implemented in the event generators HERWIG and PYTHIA, which represent the current state of the art in QCD shower simulation. In this section we give a brief description of the shower simulation algorithms used by these programs.

A documentation of HERWIG can be found in [20]. The starting point for the algorithm is that parton cascade has the structure of a Poisson process, as we have explained in the previous section. This process is implemented in HERWIG parton-by-parton. HERWIG adds to this a presumption for treating soft gluons ($z \ll 1$), for which the coherence between gluons that radiate from different parents can be taken into account. Marchesini and Webber [20] argued that the principal interference effect is fully destructive (to the leading order). The proper inclusion of leading-order infrared contribution is very simple: the available phase space of the successive branching is reduced to an angular-ordered region. The branching angle decreases as one moves from hard vertex to the final emitted partons. Outside this angular-ordered region the coherence of different emission graphs leads to destructive interference. The analysis of leading infrared singularities is performed through a technique called "eikonal current approximation". For a parton with momentum p, radiating multiple gluons with momentum $q_1, \cdots, q_n$, the result of this technique is to restrict outgoing partons created by a QCD color singlet to the region:

$$\theta_{pq_n} < \cdots < \theta_{pq_1}. \tag{3.14}$$

The angular ordering with respect to partons from initial-state QCD radiation depends on the energy of each parton. Since we are simulating e^+e^- processes with no hadrons in the initial state, we do not need to worry about the QCD space-like branchings.

PYTHIA [17,36] is also based on the parton-by-parton realization of the Poisson process that generates the solutions to the Altarelli–Parisi equations. In PYTHIA's approach, a parameter m_{min} is introduced to regularize soft-parton divergences. PYTHIA imposes the restriction on the range of longitudinal fraction z ($1 > z_+(m) > z > z_-(m) > 0$), depending on m_{min} and the parton virtuality. This introduces correlation between the various emissions in a way that is similar to angular ordering [36]. The condition similar to angular ordering ($\theta_3 \leq \theta_1$) takes the form

$$\frac{z_3(1-z_3)}{m_3^2} \geq \frac{1-z_1}{z_1 m_1^2}.$$

The paper [17] demonstrates that for a single observable at any fixed energy, it is possible to find a conventional algorithm which can give the same result as the coherent algorithm once the hadronization effects are included. However, such an algorithm does not give consistent results at other energy scales. Thus in our work we will use a coherent approach by including the phase-space reduction through angular-ordering.

PYTHIA and HERWIG also include simulation of the hadronization stage of a QCD reaction, in both cases, by assembling partons into corlor-singlet clusters which are then decayed to ground state hadrons. This process is controlled by many parameters which are fit to experimental data. However, the hadronization stage involves only small momentum transfers, and does not affect the global momentum flow in the event.

The treatment of QCD showers given by HERWIG and PYTHIA is very effective at modelling experimental data. However, there is a longstanding problem of how to make these algorithms represent perturbative QCD results more accurately. Angular ordering is a leading-order approximation, but it is not simple to improve upon it. Towards this goal, we need to represent the whole shower at once, so that we can include more subtle correlation between partons. This is the problem we are trying

to solve in our work.

3.4 Algorithmic construction of Parton Shower

Our goal is to construct the entire parton shower as a single multi-dimensional integral.

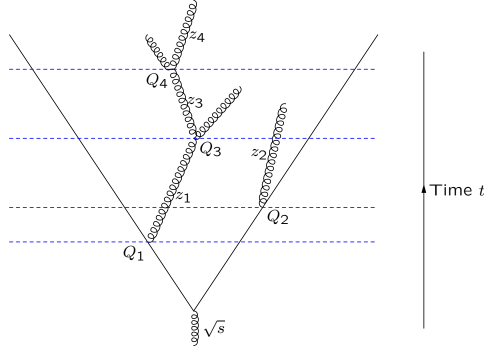


Figure 3.3: Gluon showering, horizontal lines are represented by branching parameter b[i]

We need to construct a showering event in two steps. The first step is to generate the tree structure of a shower as in Fig. 3.3 using the Sudakov suppressed Poisson generation. This step, as we can tell, from the arrow of Fig. 3.3, is going upward, in the direction of decreasing virtual parton mass $Q[i]$. Following Appendix B.1, we have the inputs as: $\xi[i], q[i], B[i]$ (random numbers $\sim [0,1]$).

Starting with $T = T(\sqrt{s})$,
if $(q[i] < e^{-WT})$, exit;
else $\{$
$t[i] = T + \frac{1}{W}\log(q[i]), t[i] \to T, Q[i] = Q(t[i])$;
if $(B[i] < W_{quark}/W)$, quark radiates gluon $i, z[i] = z_q(\xi[i])$;
else if $(B[i] < 2W_{quark}/W)$, anti–quark radiates gluon $i, z[i] = z_q(\xi[i])$;
else determine jth gluon left (right) side radiates gluon $i, z[i] = z_g(\xi[i])$;
record corresponding weight $W_j(z[i])/W_j$; $\}$
where,

$$W = 2W_{quark} + N_g W_{gluon},$$

$$W_{quark} = \int_{z_{min}}^{1} dz P_{qg}(z),$$
$$W_{gluon} = \int_{z_{min}}^{1} dz P_{gg}(z).$$

Here N_g is the number of gluons the algorithm generates at each step. The program keeps track of the number of gluons that have been created and uses the corresponding Poisson formulae to generate the next gluon. $Q(t)$ is the inverse transformation to (3.11), z_q is the transformation (3.12). z_g is defined as:

$$z_g(\xi) = \frac{e^{-\xi}}{1 + e^{-\xi}}, \tag{3.15}$$

which always yields a number $\leq .5$. This transformation is intended to take care of the fact that gluon splitting function has singularities at both 0 and 1. And it possesses the symmetry $z \leftrightarrow 1 - z$. So we'll only work on the generation of $z \leq \frac{1}{2}$, and then use branching parameter $B[i]$ to determine if it comes out from the left side or the right side, as in Fig. 3.4.

Figure 3.4: gluon branching

After the structure of the showering tree is established, we need to focus on how to get the output as a set of massless partons, expressed in terms of their Lorentz 4–momentum vectors, linked by a color chain. This is the second step of the construction, which should be carried out now in a downward direction, each time starting from the lowest–invariant–mass ($Q[i]$) virtual parton. In order for these states to cover the whole phase space, we should allow each local frame to longitudinally rotate to any direction in the space. We outline the boosting steps in detail:

One more addition to the Input set: $\phi[i]$ (random number $\sim$ $[0, 2\pi]$) as the transverse rotation angle.

Start with the lowest invariant mass among unboosted Q's — labelled Q,

if (Q has no connection to previously boosted systems) Instantiate the 2 partons produced by Q as :

$P_1 = \frac{Q}{2}(1,0,0,1), \qquad P_2 = \frac{Q}{2}(1,0,0,-1);$

else if (Q connects to 1 previously boosted system with invariant mass Q_0)

Boost the whole system Q_0 from its rest frame $(Q_0, 0, 0, 0)$ to frame $(E, 0, 0, \pm p)$, instantiate a new massless parton $(p, 0, 0, \mp p)$. The sign depends on whether the new gluon is on the left– or right–hand side, E and p satisfy $E+p = Q, \qquad E^2-p^2 = Q_0^2$;

else (which means Q connects to 2 previously boosted systems Q_0 and Q_1)

Boost the whole system Q_0 from its rest frame $(Q_0, 0, 0, 0)$ to frame $(E_1, 0, 0, \pm p)$, boost the whole system Q_1 from its rest frame $(Q_1, 0, 0, 0)$ to frame $(E_2, 0, 0, \mp p)$, the parameters sastisfy:

$E_1 + E_2 = Q, \qquad E_1^2 - p^2 = Q_0^2, \qquad E_2^2 - p^2 = Q_1^2.$

Finally, apply longitudinal plane rotation with angle θ ($\cos\theta$ is defined to be $1 - 2z$), transverse plane rotation ϕ for all the partons in the newly established system Q.

Recursively repeat above steps until we reach the center–of–mass system $\sqrt{s}$

Note that we can establish the color ordering of the final partons through above operation as well: it is realized when we decide if the new parton is added to the left or the right side of the boosted system.

The last question we would like to address is: are all the partons instantiated in this way physically real? There are several constraints to this problem:

1. Each time a new gluon is radiated, we should check that energy–momentum conservation is not violated. i.e., in Fig. 3.5, if all three of the partons are off–shell, then we should make sure that $Q_2 + Q_3 \leq Q_1$;

2. Because each z_i is a relative fraction, i.e. the longitudinal fraction of the last parton that is emitted, we would like to make sure that the absolute value of its energy should be within the range of perturbative QCD, namely, $\geq \Lambda_{QCD}$. Symbolically, this constraint is written as :

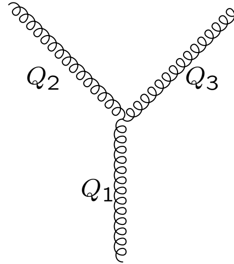


Figure 3.5: Dynamic check: When virtual parton Q_1 branches off to give two partons, and their mass (virtualities) are Q_2 and Q_3, $Q_2 + Q_3 \leq Q_1$ should be satisfied.

$$\Pi_i^P \{z_i, 1 - z_i\} \times \sqrt{s} \quad \geq \quad \Lambda_{QCD}.$$

P indicates the path from $\sqrt{s}$ vertex to the interaction point.

Any of the partons which fail to concord with these constraints will not get instantiated in the second step. However, their weights are still being accumulated in the first step.

In summary, this prescription exactly observes the probability in the Poisson process. It achieves the following at the parton shower level:

1. It automatically conserves energy–momentum.

2. It fills all of the physically possible multi–parton phase space.

3. It precisely implements the parton shower by realizing the multi–dimensional integral.

4. It achieves competitive speed in real time. The event selection happens at a rate of 1 per 10 millisecond.

3.5 Leading-Order perturbative QCD treatments

Now we have finished the basic construction of the parton shower. For our final states to match perturbative QCD results, we have to apply modifications to our

simulation in order to make it more accurate. Currently, this includes schemes for angular ordering and reweighting of 3-jets distributions.

3.5.1 Angular Ordering

We impose coherence on the branching process following the strategy laid out in [23]. For the branching $a \to bc$, $\zeta = p_b \cdot p_c / E_b E_c \simeq 1 - \cos\theta$, where θ is the emssion angle. We would like to impose $\zeta' < \zeta$ for the successive branchings ζ'.

Appendix B.2 gives the detailed kinematics of the constraints for a 3–parton–final state. However, because ζ is not Lorenz invariant, in our current treatment, we perform the angular ordering after the boosting stage is finished. To ensure $\zeta = 1 - \cos\theta$ strictly, both p_b and p_c have to be massless, so we apply this condition to the momenta of the final partons. Note that this is a criterion that requires construction of the whole event. We feel this is a safe measure because in this way the color–chain ordering will also be automatically preserved. Nevertheless this global approach introduces some disadvantages: Because we assign all the events which violate angular ordering the weight of 0, the normalization of the shower integral is no longer exactly 1. One possible way to remedy this will be to recalculate the Sudakov factor, in this case, in eq. (3.10) the upper integration limit for z will be $f(Q)$, instead of being 1. The event generation scheme must be modified to accomodate this. For the present, we have not included this effect.

3.5.2 Next–to–Leading–Order Reweighting

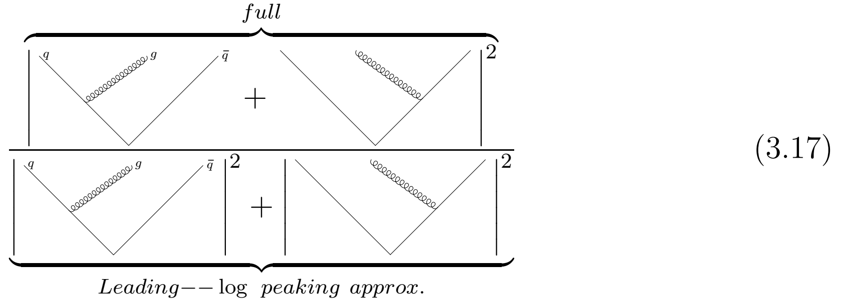
In addition to the imposition of angle-order, we can attempt to match our parton showering Monte Carlo to a finite–order QCD calculation. The motivation for the matching is due to the fact that the parton shower is only accurate to the leading–logarithm accuracy $(\alpha_s \log^2(\frac{Q^2}{s}))^n$. We want our simulation to yield finite results which can be compared with experiments. This means not only that all orders of infra–red divergence in our simulation rate should cancel, but also that the finite number left over should correspond to the more exact quantum field theory calculation.

We propose to implement this matching order by order. For now, we work at the lowest order — $O(\alpha_s)$. We choose our starting point to be the simplest 3-parton final

state, whose exact Feynman calculation to O(α_s) is well–understood. It is

$$\frac{1}{\sigma_0}\frac{d^2\sigma}{dx_1 dx_2}(e^+e^- \to \overline{q}qg) = \frac{2}{3}\frac{\alpha_s(Q)}{\pi}\frac{x_1^2 + x_2^2}{(1-x_1)(1-x_2)} \tag{3.16}$$

Our strategy is to "tag" each parton coming out of showering with this cross section, i.e., they have to be reweighted by a factor of



(3.17)

This can be pursued in two ways:

1. Perform such a reweighting at each branching;

2. Perform such a reweighting on the final states clustered into 3 jets.

It is not clear apriori which approach is better. For simplicity, we have applied the second approach. We apply a cluster (jet) algorithm in order to cluster a multiple–parton final state into 3 jets. The first step is, among pairs of nearest neighbors in the color chain, to find the pair which has lowest invariant mass and combine the two partons. We then carry out this process recursively until there are only 3 jets left. There are different schemes for the combination of two partons into a jet [24]. Each scheme conserves different physical quantities. We choose to use the E0 scheme which combines the two partons in the following way:

$$E = E_i + E_j, \qquad \vec{p} = \frac{E}{|\vec{p_i} + \vec{p_j}|}(\vec{p_i} + \vec{p_j}). \tag{3.18}$$

As one can see, E0 scheme conserves energy, but violates momentum conservation. The reason we choose this scheme is because it is the scheme for which, in full QCD simulation at the Z mass QCD simulation, the parton level distributions are closest to the hadron level distributions. A diagrammatic way to understand the clustering process is shown in Fig. 3.6.

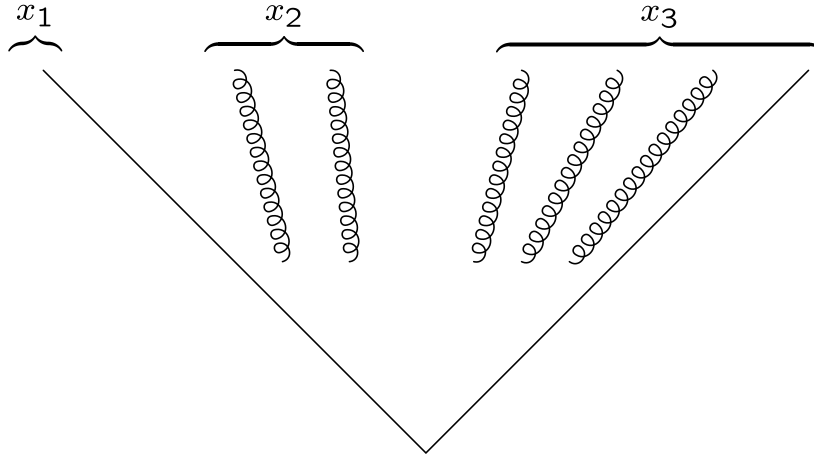


Figure 3.6: Clustering of final states into 3 jets.

After 3 jets are formed, we calculate the jet variables according to:

$$x_1 = \frac{2E_{jetq}}{\sqrt{s}}, \qquad x_2 = \frac{2E_{jet\bar{q}}}{\sqrt{s}} \tag{3.19}$$

We use these parameters to calculate the O(α_s) Feynman cross section in eq. (3.16). This gives the implementation of (3.17). We still need the denominator to O(α_s). Appendix B.3 demonstrates the calculation.

When we compute the reweighting factor of each event, we need to apply it to various physical observables, like global shape variables and jet rates. Instead of counting each event with weight 1 as after showering, now we count it with the reweighting factor. In this way, the preservation of total rate up to O(α_s) is achieved.

3.6 Physical Simulation of QCD at Q $= m_Z$

3.6.1 Jet Rate at O(α_s)

Extensive experimental efforts have been directed at understanding how the jet rates vary according to the cut–off criterion. Using cluster algorithms such as those described in the previous section, we can define the cross section for e^+e^- annihilation

into n jets σ_n. These quantities depend on a parameter

$$y_{cut} = Q_{cut}^2/s, \tag{3.20}$$

where Q_{cut} is the maximum mass of a pair of clusters that are conglomerated by the jet algorithm. Our simulation results are plotted in Fig. 3.7 and Fig. 3.8.

Fig. 3.7 compares our simulation result, using clustering with the E0 algorithm, to the experimental results and to PYTHIA simulations using the E0 algorithm. In [24], it is shown that the experimental and PYTHIA parton level jet rates closely agree with each other. The experimental results are interpolated by a smooth curve. Our simulation results have the same shapes as the data curve for $y_{cut} > 0.02$, but are offset by a constant factor. This suggests the necessity of including higher–order corrections into our algorithm. Both in our simulation and in PYTHIA, the jet rates at parton level do not vary significantly with the different jet clustering schemes.

In Fig. 3.8, we have plotted our jet rate using k_T (or Durham) jet algorithm [23]. The result of our simulation is compared to HERWIG results at parton level [37]. One can observe that our 2, 3 and 4-jet rates agree quite well with HERWIG for $y_{cut} \geq 0.01$. Fig. 3.9 is a zoomed-in version of Fig. 3.8 for $y_{cut} < 0.1$. We hope to better improve our algorithm by inclusion of higher order perturbation results. Currently we do not understand why our simulation agrees with HERWIG in the k_T scheme, nevertheless does not agree with PYTHIA in the E0 scheme.

3.6.2 Thrust Simulation at O(α_s)

A list of collinear–safe measures of the topology of hadronic final states is given in [38]. Out of these, we have simulated the thrust: $T = \max_{\vec{n}} \sum_i |\vec{p_i} \cdot \vec{n}| / \sum_i |\vec{p_i}|$ [39,40].

An exactly back–to–back 2 jet event has $T = 1$. Very evenly distributed 3 identical jets have $T = \frac{2}{3} = 0.67$, all the other 3–jet–like or 2–jet–like events should have their thrust value between these two values. Thrust distributions have been measured experimentally, for example, the results from OPAL [41] are shown in Fig. 3.10.

We can tell in this plot that reweighting has mildly improved the performance of the parton Shower Monte Carlo. Our thrust plot agrees with the data in terms of its shape. However, there is an offset between our simulation curve and the data curve. We hope to remove this by inclusion of higher-order perturbative results.

3.7 Conclusion and Outlook

We have presented a new algorithm of Parton Shower Monte Carlo matching to a finite–order QCD calculation. The algorithm is set up so that we can introduce arbitrary correlations between radiated partons, to mimic the results of finite-order QCD computation. As examples , we have incorporated the "angular ordering" mechanism to suppress QCD destructive interference in the parton shower simulation stage. It uses O(α_s) exact QCD calculation to reweight the generated events. The advantage of this method lies in the fact that it successfully avoids the appearance of negative weights. It generates events at a very fast speed (1 per 40 milliseconds after "angular ordering").

However, compared to data and second-order simulation curves, our simulation results have significant offset. This shows that this work at $O(\alpha_s)$ is definitely not state-of-art. Further work including higher order correction needs to be implemented. This will involve the following:

1. Use 2-loop β function in the running $\alpha_s(Q)$;
2. Take into account the 1-loop threshold effects in $\alpha_s(Q)$ corresponding to the self-energy of c and b quarks;
3. The exact result of σ_{tot} to $O(\alpha_s^2)$;
4. The exact formula of $e^+e^- \to q\overline{q}gg$ to O(α_s^2);

These items are not diffcult to incorporate them into our current algorithm. However, there is one detail which needs special attention.

In fig. 3.11, the final state two gluons are in a region of phase space where one results from hard emission, while the other is collinear. This configuration corresponds to the subleading logarithmic term $\alpha_s^2 log^2(q)$. To include such term properly, a seperate reweighting is needed in this region.

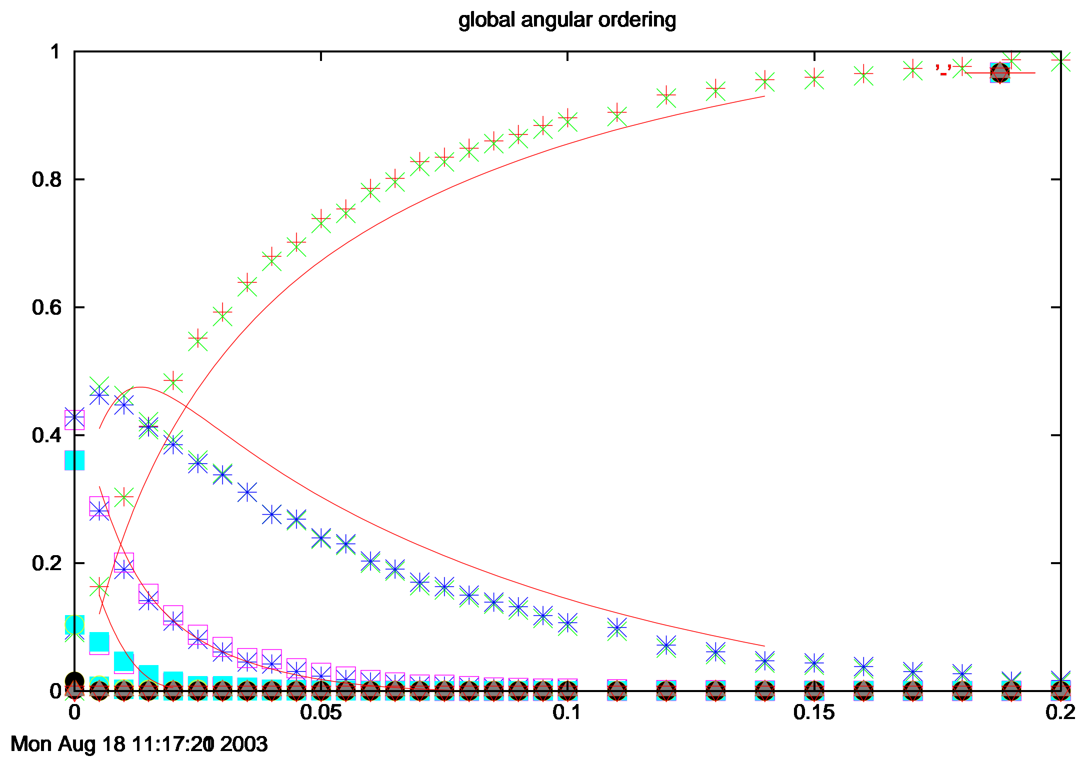


Figure 3.7: Jet rates as a function of the paramter y_{cut} defined by eq.(3.20). The solid lines are a smooth fit of the OPAL data we collected in [23]. Since in [24], PYTHIA's parton level simulation yields a jet rate plots sitting right on top of the data, this can also be viewed as the comparison of our $O(\alpha_s)$ simulation against PYTHIA's simulation. In all cases, clustering is done with the E0 algorithm. The discrete points are our simulation results. For 2-jet, 3-jet, 4-jet curves, +, ×, star lines are produced by purely showering, while ×, star, box lines are produced by the combination of showering and reweighting. One can see that in this case the reweighting scheme seems to have little effect.

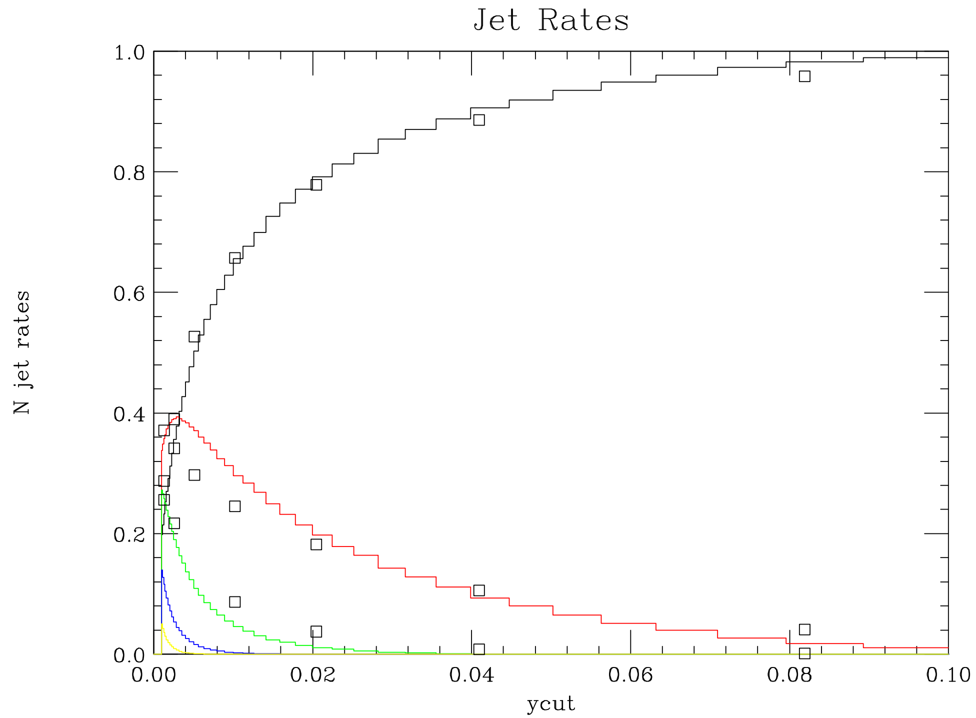


Figure 3.8: Jet rates as a function of the paramter y_{cut} defined by eq.(3.20). Solid lines are parton level results from HERWIG [37], discrete boxes are our simulation. Both cases are using k_T (or Durham) scheme for the jet algorithm. Our data deviates the HERWIG curve for $y_{cut} \leq 0.01$, which is roughly the scale that perturbative QCD breaks down.

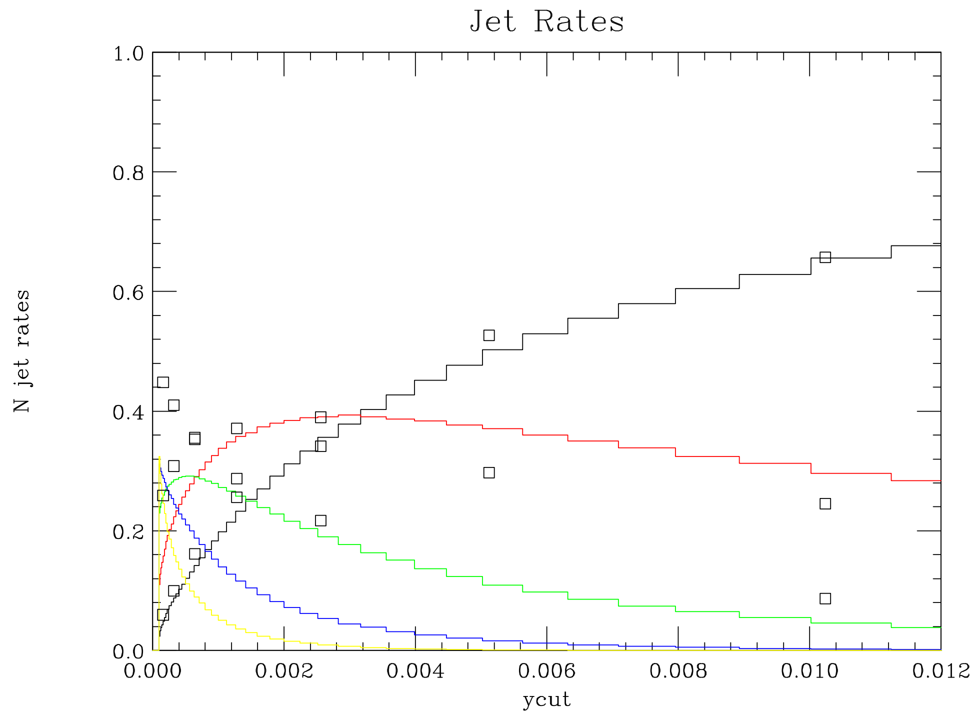


Figure 3.9: This is the zoomed-in version of Fig. 3.8.

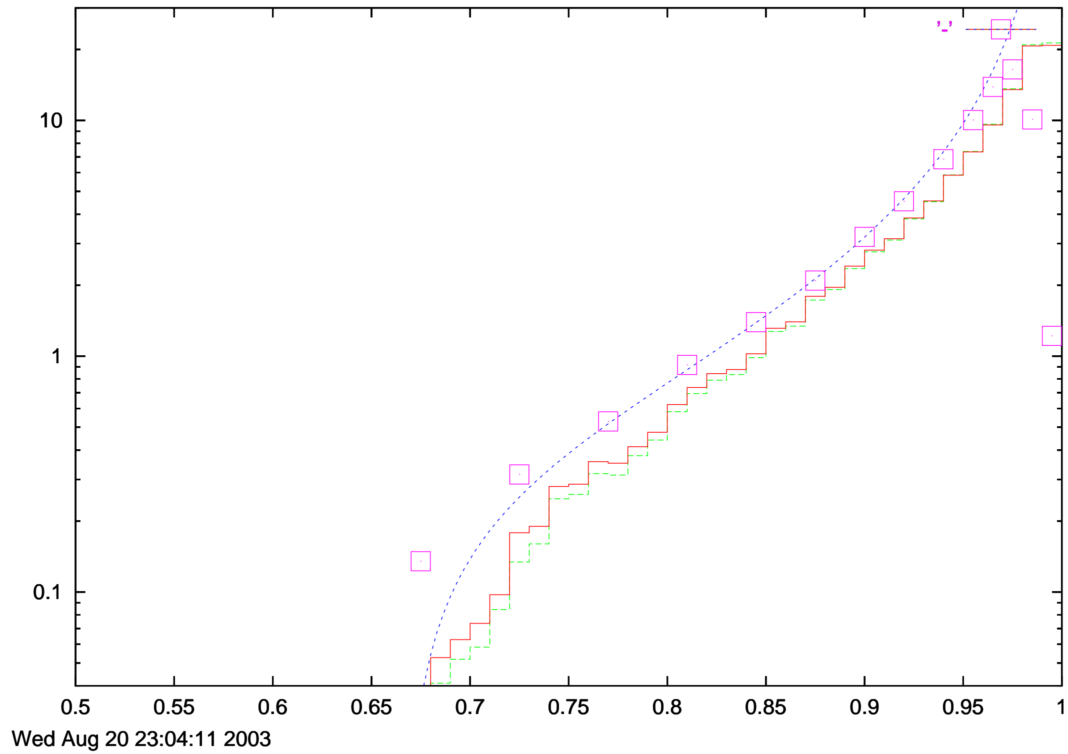


Figure 3.10: Thrust plot at $\sqrt{s} = m_Z$. The discrete squares are taken from OPAL data, the dashed curve simulates showering, the solid curve is the reweighted version of the dashed ones, the dotted curve is an $O(\alpha_s)$ perturbative theory calculation fit to the data, with α_s chosen at 0.265 according to [23].

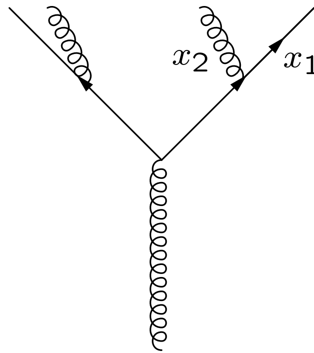


Figure 3.11: The final 4-parton state consists of 3 jets, quark x_1, gluon x_2, and anti-quark combined with a collinear gluon as the third jet.

Chapter 4

Conclusions

We have discussed several issues in physics simulation of high energy e^+e^- annihilation. Our research on MAVEN algorithm suggests a bright future of borrowing ideas from other fields and applying them successfully in open questions in physics. However, MAVEN, as well as VEGAS, FOAM, etc., is only the starting point during the journey of achieving a universally effective multi–dimensional integrator. VEGAS is efficient enough to handle distributions which have relatively flat landscapes, this includes necessary coordinate transformation like what we have implemented in the Parton Shower simulation. For an unknown, highly complicated distribution with several dimensions strongly correlated with each other, MAVEN is a good method for trial. In fact, our research shows that the more complicated a distribution is, the more powerful MAVEN shows itself to be. However, it does suffer from the *curse of dimension*. And in simple cases, it is more cumbersome than VEGAS. We hope that MAVEN's birth due to a simple application of neural network theory can help contribute to further, more effective simulation algorithms. Nevertheless, it is useful to enrich the pool of event generation algorithms so that people will have more tools in solving challenging problems. Although we didn't use MAVEN to generate parton shower, the philosophy of viewing the whole process of particle creation and decaying as a multi–dimensional integrator guides the writing of simulation for parton showers. Because the structure of each subdistribution function is well–known before the simulation starts, and because of its suitability to modularization, VEGAS turns out to be the most efficient algorithm. Its main appealing properties are the speed, and

the avoidance of negative weights. However, it remains as an open question as to if this algorithm can achieve as good simulation behavior as HERWIG and PYTHIA. To test that, the inclusion of higher–loop radiative corrections is necessary.

In summary, I hope that my experience through these explorations in simulation of high energy e^+e^- interaction process can help contribute to the improvement of simulation methods or schools–of–thinking. It is essential to make the MC simulations meet the ever–demanding needs of experimentalists, and the improvement on both efficiency and accuracy of algorithms is vital for the fast–advancing scientific knowledge acquiring. We will only achieve improvement through persistent trial–and–error.

Appendix A

Notes for the use of the MAVEN Software

To accompany Chapter 2, we have posted in eprint archive a `tar` file that contains the code for a `C++` class that implements the MAVEN algorithm. The distribution also contains a program that exercises the MAVEN event selection and a useful parent class for Monte Carlo event selection. In this Appendix, we give some documentation for these programs. But first let's review the mathematical foundation of the core equations in neuron construction and evolution.

A.1 Bayesian statistics applied to neuron evolution

If the random variable $\vec{x}$ has distribution $f(\vec{x})$, according to information theory this distribution's entropy is

$$S = \int_{\Omega} d^n x f(\vec{x}) \ln(f(\vec{x})) \tag{A.1}$$

Furthermore, if one uses an approximation $p(\vec{x})$ to sample $\vec{x}$, a standard measure of the distance between $f(\vec{x})$ and $p(\vec{x})$ is in the form

$$L = -\int_{\Omega} d^n x f(\vec{x}) \ln(\frac{p(\vec{x})}{f(\vec{x})}) \tag{A.2}$$

known as Kullback–Leiber distance [42]. Using function variation method, one can show that $L \geq 0$, the equality holds true if and only if $p(\vec{x})$ is exactly $f(\vec{x})$.

Suppose $p(\vec{x})$ is a linear superpostion of basis functions. According to Chapter 2, each basis function is called one feature. Typically a feature is represented by its first and second moment. i.e., its center of location and its one–sigma radius. It is computationally convenient to use the same basis function with different parameters to build up $p(\vec{x})$. For example, it can be a Gaussian, or more sophiscatedly, a nonlinear funtion with the mechanism of emphasizing important parameter range while suppressing the unimportant ones into background(such as the one being used in Chapter 2). We write down $p(\vec{x})$ in the form

$$p(\vec{x}) = \sum_{i=1}^{M} p(\vec{x}|j)P(j) \tag{A.3}$$

M is the number of features in the density $p(\vec{x})$. $p(\vec{x}|j)$ is the probablity of $\vec{x}$ conditioned on the jth feature. It is a function of the parameters — center μ_j and one sigma matrix Σ_j, possibly with more parameters $\{A_j, ...\}$. P(j) is the probability that jth feature gets selected, we treat P as priors in our statistical modeling. All these parameters will be determined through iterative adaption.

Substituting (A.3) into (A.2), and using the Lagrange muliplier, we get,

$$\sum_{\vec{x}} \frac{f(\vec{x})}{p(\vec{x})} \frac{\delta p(\vec{x}|i)}{\delta\{\mu_i, \Sigma_i, ...\}} = 0 \tag{A.4}$$

$$\sum_{\vec{x}} \frac{f(\vec{x})}{p(\vec{x})} p(\vec{x}|i) = \lambda = \sum_{\vec{x}} f(\vec{x}) \tag{A.5}$$

If $p(\vec{x}|i)$ has the Gaussian distribution with μ_i and Σ_i, and taking into acount of the Bayesian formula

$$p(i|\vec{x}) = \frac{P(i)p(\vec{x}|i)}{p(\vec{x})} \tag{A.6}$$

then (A.4) and (A.5) will turn out to be

$$\vec{\mu} = \frac{\sum_{i=1}^{M} f(\vec{x})p(i|\vec{x})\vec{x}}{\sum_{i=1}^{M} f(\vec{x})p(i|\vec{x})} \tag{A.7}$$

$$\Sigma = \frac{1}{d}\frac{\sum_{i=1}^{M} f(\vec{x})p(i|\vec{x})(\vec{x}-\vec{\mu})(\vec{x}-\vec{\mu})^T}{\sum_{i=1}^{M} f(\vec{x})p(i|\vec{x})} \tag{A.8}$$

$$P(i) = \frac{1}{M}\frac{\sum_{\vec{x}} p(i|\vec{x})f(\vec{x})}{f(\vec{x})} \tag{A.9}$$

which lays out the rule for iteration. The right hand side is calculated using the parameters from the current iteration, the left hand side will be fed into the next iteration.

Some statistics insight from the above equation. $p(j|\vec{x})$ is the probability that given $\vec{x}$(data) and the basis funtion(model), the jth basis function(model) is the correct one reproducing $\vec{x}$(data). Thus $p(j|\vec{x})$ is the posterior knowledge, indicating the credibility of the jth basis function(model). (A.9) shows that the prior $P(j)$(which can be regarded as the overall credibility of jth basis function) is the average of $\vec{x}$'s posterior knowledge with jth basis($p(j|\vec{x})$) weighted by its true probability $f(\vec{x})$. The same line of thought extends to (A.7) and (A.8). When one changes the basis function from Gaussian to another continuous positive function, holding on to the parametrization in terms of feature's geographic location, the above equations will still stand intact as the core of the algorithm. With this analysis, we have finished introduction of EM(Expectation Maximization) algorithm.

A.2 Structure of the software distribution

The software is presented as a `tar` distribution which unpacks to a directory `maven`. This directory contains the `MavenMC` class and various associated programs, all written in `C++`. Theses include definitions of vector and matrix classes and class for files to be read by the graphing program `gnuplot` [43]. The program `testMaven.cpp` carries out adaptation and event selection with VEGAS and MAVEN for one of a set of two–dimensional functions listed in `myfunctions.h`. To compile this program, put the correct name of the `C++` compiler in the makefile, then type `make testMaven`. The program will write VEGAS and MAVEN statistics to the terminal and will produce three `gnuplot` files, `MC.gp`, `MCFeatures.gp`, and `MCWeights.gp`. These show, respectively, the final event selection with VEGAS and MAVEN, the adapted features as in Fig. 2.3(a), and the generated points with weights as in Fig. 2.3(b).

A.3 Monte Carlo interface

The MAVEN and VEGAS event generators `MavenMC` and `VegasMC` are constructed as subclasses of a general interface for Monte Carlo event selection, the class `MonteCarlo`. This class defines the basic operations of event selection and gathers the statistics of a selection operation while leaving the actual method of event selection abstract. The public methods of `MonteCarlo` are shown in Fig. A.1. The function $f(x)$ is called `surface(X)` in the code. The class is operated by first calling `prepare`, giving the number of function calls to be used in the initialization. Then one calls `getPoint` to return a weighted or unweighted point x. Note that the important functions `surface`, `prepare`, and `getPoint` are virtual functions to be filled in by the subclass. We have incorporated this general interface into `pandora` as a way to easily swap in and out different algorithms for event selection.

The random number generator is chosen at the top of `MonteCarlo.h` to be `ran2()` from [2], which is defined in `random.h`. Any other random number generator could be substituted at this point.

A.4 Initialization and control of `MavenMC`

The public interface of the class `MavenMC` is shown in Fig. A.2. The constructor for the class `MavenMC` is

```
MavenMC M(N);
```
(A.10)

givien the total number of dimensions `N`. To mark fast dimensions, one calls them out by number. The following sequence of commands marks dimensions 1, 2, 7, 8, turns on feature analysis for these dimensions with 3 initial features per dimension per initialization step, and then calls `prepare` requesting a total of 10^6 function calls:

```
M.mark(1);  M.mark(2);  M.mark(7); M.mark(8);
M.usemultigrid(3);
M.prepare(1000000);
```
(A.11)

```
class MonteCarlo{

 public:

    int N;   /*  number of variables integrated over  [0,1]   */

    MonteCarlo(int N);   /* initializes MonteCarlo variables  */

    virtual double surface(DVector & X)=0;  /* function integrated */

    virtual void prepare(int nevents, int nseed = 1)=0;

    virtual DVector  getPoint()=0;
    virtual DVector  getPoint(double & weight)= 0;

    void reset();
    void resetMC();  /* reset Monte Carlo counters only */
    void resetseed(int nseed = 1);

    double integral(double & sd);

    void setThreshold(double x);

    void printTitle();
    void printIntegral(int mycalls = 0, int info = 0);
      /* call printTitle at the beginning of the run
         then call printIntegral after each step,
  passing the calls/step */
    void printStatistics();
      /* call printStatistics at the end of the run for statistics on
   the Monte Carlo selection */
    void quiet();
    void verbose();
};
```

Figure A.1: Public methods of the class `MonteCarlo`.

```
class MavenMC : public MonteCarlo {

 public:

  MavenMC(int N);
  virtual ~MavenMC();

  void mark(int i);
  void usemultigrid(double nfpd);
  void prepare(int nevents, int nseed = 1);

  DVector getPoint();
  DVector getPoint(double & weight);

  void visualizeFeatures(gnuplot & G, int k, int l);
  void visualizeWeights(gnuplot & G, int k, int l);
};
```

Figure A.2: Public methods of the class `MavenMC`.

We can then simply call

```
nextx = M.getPoint();
```
(A.12)

to fill a vector `nextx` with a point chosen according to the target PDF.

The class `MavenMC` contains as a protected member a list of pointers to `feature` classes. This structure allows features to be created and destroyed as they are needed. We also specify a class `VegasGrid` that takes care of VEGAS grid accounting and updating.

The command

```
M.visualizeFeatures(G,k,l)
```
(A.13)

draws the adapted features of the MAVEN event selector — projected into the two–dimensional (`k`, `l`) plane — into a file `G` that can be read by `gnuplot`. The command

```
M.visualizeWeights(G,k,l)
```
(A.14)

selected 1,000 weighted points and plots their projection into the (`k`, `l`) plane in a file `G` that can be read by `gnuplot`. The points are drawn with a color–coding command. It draws the adapted features of the MAVEN event selector into a file `G` that can be read by `gnuplot`. The command gives each point a color–coding: blue if $w(x)/I < 3$, green if $w(x)/I < 10$, red if $w(x)/I < 100$, and purple otherwise.

A.5 Use of vector and matrix classes

The programs in the `maven` directory make use of double and integer vector and matrix classes defined in the files `DClasses` and `IClasses`. These vectors and matrices are indexed between arbitrary limits; for example, a vector indexed from `1` to `N` is constructed by

```
DVector V(1,N); .
```
(A.15)

In `MavenMC`, we avoid costs in speed from vector initialization by initializing in advance all vectors and matrices needed in event generation. Overloaded addition and multiplication are defined for the vector and matrix classes, but they are not used into the code, except in one specific place. Because the feature updating described in (2.15)–(2.18) is called only at the end of each iteration, it is reasonable to allow operations in that step that involve memory allocation. Thus, the function `feature::update()` freely uses vector algebra methods, including matrix diagonalization, defined in `DClasses`. Users who would like to substitute their own vector and matrix classes should take care to rewrite this function.

A.6 Implementation of Feature Adaptation

This section will outline the essential parts of the code `MavenMC.cpp`, which implements the adapted iteration. We'll cover this in two sections. This section will be focused on feature adaption part of `prepare()` function, as shown in Fig. A.3. Feature adaption is realized through `nasteps` iteration. In each adaption, subroutine `adaptstep()` in Fig. A.4 is performed, where knowledge of the object function is obtained, with each data point selected incorporating (2.15). Afterwards, subroutine `update()` is in Fig. A.5 carried out, where Bayesian analysis, (A.7) to (A.9) are

```
void MavenMC::prepare(int nevents, int nseed){
  int k;
  int nasteps = 10;  /* steps for feature adaptation */
  int ngasteps = 8;  /* steps for grid adaptation  */
  int ngssteps = 2;
  int mustrepeat;
  int rgasteps, rgssteps;   /* steps remaining  */
  double amaxweight;
  resetseed(13);
  reset();     /* reset MonteCarlo integration buffers  */
  if (suppressPrinting == 0) printTitle();
  int ncalls = nevents/(nasteps + ngasteps + ngssteps);
  if (nfeatures == 0){
    ngasteps = 5;
    ncalls = nevents/(ngasteps + ngssteps);
  } else {
   if (suppressPrinting == 0) cout <<
    "                 features adaptation:  "
    << "                   no. of features "<< endl ;
      /* feature adaptation */
   int print = 0;
   for (k = 1; k <= nasteps; k++){
     adaptstep(ncalls);
     if (k < nasteps-2){
       int nadd = (int) (NM * nfpd);
       addfeatureblock(nadd);
     }
   }
  }
  finishfeatures();
}
```

Figure A.3: prepare function of the class `MavenMC`.

```
void MavenMC::adaptstep(int ncalls){
  flist[0]-> p = background;
  maxweight = 0.0;
  double weight;
  for (int i =1; i <= ncalls; i++){
    findPoint(weight);
    adddatum(weight);
  }
  update();
  if (suppressPrinting == 0) printIntegral(ncalls, nfeatures);
}
```

Figure A.4: `adaptstep` function of the class `MavenMC`.

realized in each `flist[k]->update()`, correspondingly, its details are in (2.16) to (2.19).

To ensure the numerical stability, features with too small priors get deleted. After each `adaptstep()`, `nadd` random features are added into the feature pool, this intends to introduce a repulsive force among features which might later get conglomerated into one piece, thus leading the program into malfunctioning. `finishfeatures()` makes sure that the priors are normalized so that their sum is 1.

A.7 Implementation of Grid Adpatation in Each Feature

We'll continue to show the remaining part of `prepare()` function. We propose to let each feature carry with it a Vegas grid, and for the "unmarked" dimensions, we use one Vegas grid to cover it all, assuming(or after some dimensional analysis) that these dimensions of the phase space don't have strong correlations with each other or the "marked" dimensions. The adaption of the grids are in Fig. A.6

`makegrids()` subroutine switches on the mentioned VEGAS grids. These grids are going through `ngasteps` rounds of adaption and `ngssteps` rounds of searching for the worst `maxweight` for the finally established features. `gridadaptstep()` is similar to `adaptstep()` programatically, except that `adddatum()` part is replaced

```
void MavenMC::update(){
  int k, l;
  for (k = 0; k <= nfeatures; k++)  {
      flist[k]->update();
  }
  k = 1;
  while (k <= nfeatures){
   if (flist[k]->p < smallestp){
     deletefeature(k);
   } else {
     k++;
   }
  }
  for (k=0; k <= nfeatures; k++) flist[k]->resetdata();
}
```

Figure A.5: `update` function of the class `MavenMC`.

by:

```
addgriddatum(weight);
if (Slow) Slow->recorddatum(weight,1);
```
(A.16)

`Slow` is a pointer which points to the "unmarked" space Vegas grid. `Slow` being a `NULL` pointer means that all the dimensions are strongly correlated with each other and thus each feature spans the whole phase space. Now `weight` is the updated weight calculated by (2.22) to (2.23). Special attention needs to be paid to the method `addgriddatum()` in Fig. A.7, only the feature which has the largest contribution from the data point selected gets updated.

Finally, combining the VEGAS grids and features, let's see how points get selected in Fig. A.8. One feature is being selected by `choose()` method in class `fnoodle`, which is representing the feature priors. When `Slow` is a `NULL` pointer, `choosePoint()` and calculation of `modelvalue` are repeated after discussions in section 2.3. Now when `Slow` is pointing to a physical subspace("unmarked" region) and the grids have been switch on, in this region, `choosePoint()` is adopting VEGAS

```
{
  makegrids();
  rgasteps = ngasteps;
  rgssteps = ngssteps;
  omaxweight = 1.0e30;
  //  omaxweight = maxweight;
  do {
    while (rgasteps > 0){
      rgasteps--;
      gridadaptstep(ncalls,rgasteps);
    }
    omaxweight = maxweight;
    maxweight = 0.0;
    mustrepeat = 0;
    rgssteps = ngssteps;
    while (rgssteps > 0){
      rgssteps--;
      gridsearchstep(ncalls);
      if (maxweight > 3.0 * omaxweight){
        if (suppressPrinting == 0)
            cout << "    A surprisingly large maximum weight"
                 <<  " requires more grid adaptation." << endl;
rgasteps = 3;
        mustrepeat = 1;
        break;
}
    }
  } while (mustrepeat > 0);
  resetMC();
```

Figure A.6: grid adaption of the function `prepare`.

```
void MavenMC::addgriddatum(double weight){
  if (nfeatures) {
    flist[0]->addgriddatum(weight,totalxvalue);
    int imax = 0;
    double pmax = 0.0;
    for (int i=1; i <= nfeatures; i++) {
      double p = flist[i]->featurexvalue;
      if (p > pmax) {
pmax = p;
imax = i;
      }
    }
    flist[imax]->addgriddatum(weight,totalxvalue);
  }
}
```

Figure A.7: selectively update one feature using one data point

point selection method, and the associated VEGAS probability of this subspace contributes a factor , which is multiplied with its corresponding brother's probability from the feature space to give the total probability of this one point. `shufflex()` is an operation to reindex the dimensions, so that marked and unmarked space can go into two seperate continuous blocks in the phase space. This concludes our presentation of the algorithm and its implementation. A few of the parameters in this algorithm are still experimental, like the prior of `flist[0]`, and the `addgriddatum()` method of features(when the best feature gets selected to update its grid data, is it going to update it with weight 1 or with weight proportional to its fraction of contribution in the total Bayes probability?). But all in all, they will not significantly change the performance of the algorithm.

A.8 A More Systematic Estimation of `Maxweight`

The work presented in this section owes its originality to Jadach. After my presentation of all the previous materials, I had a very instructive discussion with Jadach, and later with my advisor Michael Peskin on the extraction of `Maxweight`. The

```
void MavenMC::findPoint(double & weight, int isgrid){
  double fvalue = 0.0;
  double modelvalue, p=0.0;
  owner = fnoodle->choose();
  if (Slow) {
    DVector sx(1,N-NM);
    for (int i=1;i<=N-NM;i++)
      sx[i] = ran();
    if (isgrid) {
      Slow->choosePoint(sx,p);
      sx = Slow->xg;
    }
    flist[owner]->choosePoint(sx);
  }
  else {
    flist[owner]->choosePoint();
  }
  x = flist[owner]->myxstar;
  zstar = flist[owner]->myzstar;
  if (isgrid && Slow) {
    modelvalue = value()*p;
  }
  else
    modelvalue = value();
  shufflex();
  fvalue = surface(xshuffled);
  weight = fvalue/modelvalue;
  recordI(weight);
}
```

Figure A.8: point selection mechanism

	Maxweight	t(select)
$N = 3$:		
VEGAS	1.5	0.19
MAVEN	2.5	0.53
$N = 6$		
VEGAS	97	1.72
MAVEN	50	12.79
$N = 8$		
VEGAS	122	2.48
MAVEN	36	10.92

Table A.1: `Maxweight` obtained more systematically and its corresponding event selection time

question is due to the oscilation of `Maxweight` in a numerical range. The adaptions show the decrease of `Maxweight` in scale, but within each scale, it oscilates in a quite uncontrollable way, making the comparison of two algorithms rather artificial if they happen to have `Maxweight` in the same scale. In order to have stability over the oscilating `Maxweight`, we propose to do following. After the finishing of adaption, we create a histogram of the `Maxweight`, each time when an event is being selected, we record its `weight` into the histogram, i.e., let the bin corresponding to `weight` increases by 1. When a statistics of 10,000 is reached, we have a distribution of weights displayed by this histogram. Let's take the center value of `i`th bin as $w_c[i]$, the number of points in this bin as $N[i]$, we choose our `Maxweight` as $w_c[j]$, such that `j` is the first `j` satisfying

$$\frac{\sum_j^{N_{bin}} w_c[i]N[i]}{\sum_1^{N_{bin}} w_c[i]N[i]} \geq \epsilon \tag{A.17}$$

where ϵ is a fixed small number, like 0.001.

When we apply this idea into the testing of Maven and VEGAS codes, some results are collected. e.g., a revisit to the double Gaussian profiles gives Table A.1. We can tell that the definitive advantage of Maven over VEGAS seems to shrink to some extent. Further understanding in this methodology is still needed, thus it is not employed either in `Maven` or in `pandora`. But this serves as an illustrative example

in terms of how to control statistically oscillating parameters, and the principle can be applied to other simulation work.

Appendix B

Derivations of Shower Simulation Equations

B.1 Monte Carlo based on Poisson Distribution

The most general form of Poisson distribution is

$$P(n) = \frac{\lambda^n e^{-\lambda}}{n!} \tag{B.1}$$

It normalization is

$$\sum P(n) = 1$$

Now if $\lambda = \int_0 T\omega$, temporarily we set ω to be a constant, then

$$\begin{aligned} P(n) &= \int \mathrm{d}t_1 \cdots \mathrm{d}t_n P(n, t_1, \cdots, t_n) \\ &= [\int_0^T \mathrm{d}t_1 \int_0^{t_1} \mathrm{d}t_2 \cdots \int_0^{t_{n-1}} \mathrm{d}t_n \omega^n] e^{-T\omega} \end{aligned}$$

The question we want to address is: What is the probablity distribution of the first (or the highest) t, i.e. t_1? Integrating out the other t's gives

$$\tilde{P}(n, t_1) = \int_0^T dt_1 \frac{\omega^n t^{n-1}}{(n-1)!} e^{-T\omega} \tag{B.2}$$

summing over n, integrating over t, gives

$$\int \mathrm{d}t\tilde{P}(t) = \int_0^T \mathrm{d}t\omega e^{-(T-t)\omega} = 1 - e^{-T\omega} \tag{B.3}$$

Usually T or ω is large enough to render the above result to be 1. This leads us to an algorithm for generating Poisson random variables.

1. Generate a random variable $y \in [0, 1]$;
2. if $y < e^{-T\omega}$, stop and go back to 1;
3. otherwise, we get $t = T + \frac{1}{\omega}log(y)$, substitute T with t and go back to 1;

These steps will be repeated until the stopping criterion is met.

Now when ω is a distribution function $\omega(z)$, we'll generate t as above, and generate z according to distribution $\omega(z)$. Since the Poisson distribution on t is well-normalized, we need only to update the weight $\omega(z)$, so that the overall normalization of this process becomes $\int dz\omega(z)$.

To accomodate the parton showering in Sec. 2, we analyze the case in which ω is replaced by $\sum \omega_i$. The event generation process will be slightly modified. Process j gets instantiated according to probability $\frac{W_j}{\sum W_i}$, where $W_i = \int dz\omega_i(z)$. Correspondingly, the weight $\frac{\omega_j(z)}{W_j}$ gets updated to make sure the overall normalization is 1. In our problem,

$$\sum W_i = \int \mathrm{d}t[2 \int \mathrm{d}zP_{qg}(z) + N_g \int \mathrm{d}zP_{gg}(z)]$$

Its proof of normalization is presented as:

$$\begin{aligned} &\frac{\int dt[2\int \mathrm{d}zP_{qg}(z) + N_g \int \mathrm{d}zP_{gg}(z)]}{\sum W_i} \\ &= \int \mathrm{d}t[2\frac{W_q}{\sum W_i}\int \mathrm{d}z\frac{P_{qg}(z)}{W_q} + N_g\frac{W_g}{\sum W_i}\int \mathrm{d}z\frac{P_{gg}(z)}{W_g}] \end{aligned}$$

where,

$$W_q = \int \mathrm{d}zP_{qg}(z),$$

$$\begin{aligned} W_g &= \int \mathrm{d}z P_{gg}(z), \\ \sum W_i &= 2W_q + N_g W_g. \end{aligned}$$

Note that because of the Sudakov suppression, the + and δ functions in splitting functions go away, thus,

$$P_{gg}(z) = 6[\frac{1-z}{z} + z(1-z) + \frac{z}{(1-z)}]$$

should be used in the code.

B.2 Suppressing back–to–back Events

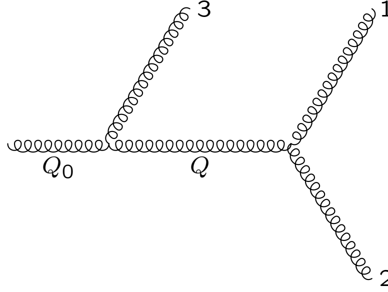


Figure B.1: suppression mechanism

All our calculation is done according to Fig. B.1. The question we ask is: When is parton 2 being emitted out of the region spanned by parton 1 and 3? To answer this question we work in the rest frame of their parent–mass Q. If we align them along their longitudinal axis, their 4-momentum are: $P_1 = \frac{Q}{2}(1,0,0,1)$, $P_2 = \frac{Q}{2}(1,0,0,-1)$. Following the strategy mentioned in Section 3, assign $\cos\theta = 1 - 2z$ to rotate the partons in the longitudinal plane, and ϕ in the transverse plane. With this the 4–momentums become:

$$P_1 = \frac{Q}{2}(1, \sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \ , \tag{B.4}$$

$$P_2 = \frac{Q}{2}(1, -\sin\theta\cos\phi, -\sin\theta\sin\phi, -\cos\theta) \, . \tag{B.5}$$

Now we transform to the rest frame of parton Q_0. We boost the Q parton system from frame $(Q, 0, 0, 0)$ to frame $(E, 0, 0, \pm p)$, the third parton is asigned momentum $(p, 0, 0, \mp p)$. One can solve E and p as mentioned, furthermore, one can determine the Lorenz boost factors for the Q system

$$\beta = (Q_0^2 - Q^2)/(Q_0^2 + Q^2), \qquad \gamma = (Q_0^2 + Q^2)/(2Q_0 Q).$$

We apply the same Lorenz boost to parton 1 and 2, which are in Q's rest frame, and achieve:

$$\begin{aligned}
P_1 &= (\frac{Q_0}{4}(1+\cos\theta) + \frac{Q^2}{4Q_0}(1-\cos\theta), \\
&\quad \frac{Q}{2}\sin\theta\cos\phi, \frac{Q}{2}\sin\theta\sin\phi, \\
&\quad \frac{Q_0}{4}(1+\cos\theta) - \frac{Q^2}{4Q_0}(1-\cos\theta)), \\
P_2 &= (\frac{Q_0}{4}(1-\cos\theta) + \frac{Q^2}{4Q_0}(1+\cos\theta), \\
&\quad -\frac{Q}{2}\sin\theta\cos\phi, -\frac{Q}{2}\sin\theta\sin\phi, \\
&\quad \frac{Q_0}{4}(1-\cos\theta) - \frac{Q^2}{4Q_0}(1+\cos\theta)) \\
P_3 &= (\frac{Q_0}{2} - \frac{Q^2}{2Q_0})(1, 0, 0, -1)
\end{aligned} \tag{B.6}$$

Applying the angular ordering,

$$\frac{p_1 \cdot p_3}{E_1 E_3} > \frac{p_1 \cdot p_2}{E_1 E_2}, \tag{B.7}$$

we get

$$z < \frac{1 - 2Q^2/Q_0^2}{1 - Q^2/Q_0^2}. \tag{B.8}$$

B.3 Calculation of the Reweighting Factor

In perturbative QCD, the cross section for ($e^+e^- \rightarrow$ 3–jets) can be written as

$$\frac{\frac{d^2\sigma}{dx_1 dx_2}(e^+e^- \rightarrow q\bar{q}g)}{\sigma(e^+e^- \rightarrow q\bar{q})} = \frac{4}{3}\frac{\alpha_s}{2\pi}\frac{x_1^2 + x_2^2}{(1-x_1)(1-x_2)} \tag{B.9}$$

With $x_1 = \frac{2E_jetq}{\sqrt{(s)}}, x_2 = \frac{2E_jet\bar{q}}{\sqrt{(s)}}$. In order to express the right-hand side using jet–variables, we need to relate variables z, Q to x_1, x_2. We go through these steps in the case of a gluon jet coming out of the quark line. Following appendix B.2 we make the correspondences $Q_1 \rightarrow Q$, $Q \rightarrow \sqrt{s}$, in Fig. B.2, and temporarily ignore the rotation in the transverse plane. We have:

$$\begin{aligned} P_1 &= \frac{Q}{2}(\frac{s+Q^2}{2\sqrt{s}Q} + \frac{s-Q^2}{2\sqrt{s}Q}\cos\theta, \sin\theta, 0, \frac{s+Q^2}{2\sqrt{s}Q}\cos\theta + \frac{s-Q^2}{2\sqrt{s}Q}) \\ P_2 &= \frac{Q}{2}(\frac{s+Q^2}{2\sqrt{s}Q} - \frac{s-Q^2}{2\sqrt{s}Q}\cos\theta, -\sin\theta, 0, \frac{s+Q^2}{2\sqrt{s}Q}\cos\theta - \frac{s-Q^2}{2\sqrt{s}Q})\,. \end{aligned}$$

Since $1+\cos\theta = 2(1-z), 1-\cos\theta = 2z$, we have

$$\begin{aligned} P_1 &= (\frac{\sqrt{s}}{2}[(1-z) + \frac{Q^2}{s}z], \frac{Q}{2}\sin\theta, 0, \frac{\sqrt{s}}{2}[(1-z) - \frac{Q^2}{s}z]), \\ P_2 &= (\frac{\sqrt{s}}{2}[z + \frac{Q^2}{s}(1-z)], -\frac{Q}{2}\sin\theta, 0, \frac{\sqrt{s}}{2}[z - \frac{Q^2}{s}(1-z)]), \\ P_3 &= \frac{\sqrt{s}}{2}(1 - \frac{Q^2}{s})(1, 0, 0, -1)\,. \end{aligned}$$

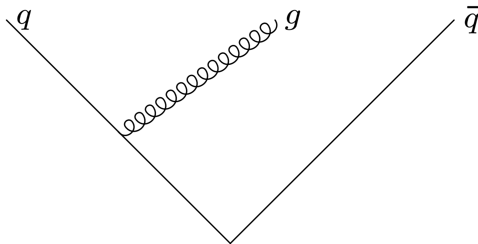


Figure B.2: Gluon radiation

Applying the definition of x_i's, we get

$$x_1 = (1-z) + \frac{Q^2}{s}z,\ x_2 = 1 - \frac{Q^2}{s},\ x_3 = z + \frac{Q^2}{s}(1-z)$$

To check these equations we add them together and confirm that the sum is 2. Therefore we can write

$$Q^2 = s(1-x_2),\ z = \frac{1-x_1}{1-Q^2/s}\ \text{ or }\ z = \frac{1-x_1}{x_2}$$

This leads us to calculate the Jacobian of the transformation from Q, z coordinates to x_1, x_2 coordinates

$$\frac{z, Q^2}{x_1, x_2} = [\frac{x_1, x_2}{z, Q^2}]^{-1} = \begin{vmatrix} -(1-\frac{Q^2}{s}) & \frac{z}{s} \\ 0 & -\frac{1}{s} \end{vmatrix}^{-1} = \frac{s}{x_2}$$

then

$$\begin{aligned} \frac{\alpha_s}{\pi}\frac{dQ}{Q}dz\frac{4}{3}\frac{1+(1-z)^2}{z} &= \frac{4}{3}\frac{\alpha_s}{\pi}dx_1dx_2\Sigma(x_1,x_2) \quad \text{(B.10)} \\ &= \frac{4}{3}\frac{\alpha_s}{2\pi}dx_1dx_2\frac{1}{(1-x_1)(1-x_2)}[x_2^2 + (x_1+x_2-1)^2]\frac{1}{x_2^2}\,. \end{aligned}$$

The same procedure can be applied to the process where the gluon jet originates from an anti–quark line, in this case we will achive the above result with x_1 and x_2 reverted. Now we have to involve the supression mechanism into the calculation of above AP amplitude.

In order to achieve that, we need to do a little exercise. In eq. (B.8), z will not be valid if $\frac{Q^2}{Q_0^2} > \frac{1}{2}$. In our case, $Q_0 = \sqrt{s}$. This means for quark emitting gluon obeying angular ordering, x_2 has to be greater than $\frac{1}{2}$. However, since the phase space requires $x_1 + x_2 > 1$, it is possible that the gluon can be radiated from $\overline{q}$. The exact possibility of such an emission depends on z. In the case of quark emission, eq. (B.8) translates to $z < 2 - \frac{1}{x_2}$. We therefore multiply $\Sigma(x_1, x_2)$ by this factor. Similar treatments are applied to the $\overline{q}$ emission. The reweighting factor for the

parton shower cross section eq. (3.17) can then be written

$$\frac{x_1^2 + x_2^2}{D}, \tag{B.11}$$

where

$$D = \frac{1}{2}(\Sigma(x_1, x_2)(2 - \frac{1}{x_2}) + \Sigma(x_2, x_1)(2 - \frac{1}{x_1}))$$

for events with $x_1 > 0.5$ and $x_2 > 0.5$,

$$D = \Sigma(x_1, x_2)(2 - \frac{1}{x_2})$$

for events with $x_2 > 0.5$ and $x_1 < 0.5$,

$$D = \Sigma(x_2, x_1)(2 - \frac{1}{x_1})$$

for events with $x_1 > 0.5$ and $x_2 < 0.5$. Note that this is not the best reweighting scheme. But it is the scheme which can assign a positive, finite value to the part of the phase space which is consistent with global angular ordering.

Therefore we can tell the following traits about this reweighting factor:

1. When $x_1 = 1$ and $x_2 = 1$, this factor is 1;
2. When $x_1 = 1$ or $x_2 = 1$, this factor is a finite number < 1;
3. In the other parts of the phase space consistent with global angular ordering, this factor is a positive, finite number of O(1).

B.4 Accompanying software

In this section, we give some documentation of the programs which are responsible for the showering and reweighting.

```
class process{

 public:

  int n;   /* number of integration variables X[i] in [0,1]
  needed to specify the final state of the process   */
  char * name;
  /*  identifying name, e.g. " e+e- -> q qbar "*/
  int ninitial, nfinal;
  /* number of initial and final helicity states to be summed over */
  CMatrix Camps;
  /*  ninitial X nfinal CMatrix containing production amplitudes  */
  DMatrix cs;
  /*  3 X 3 DMatrix containing helicity-dependent
                               differential cross section */
  process(int n, int ninitial, int nfinal);
  virtual int computeKinematics(double & J, DVector & X,
   double s, double beta) = 0;
  virtual void crosssection() = 0;
  virtual double prefactor() = 0;
  virtual void amplitudes() = 0;
  virtual LEvent buildEvent() = 0;
  /* return the parton-level LEvent determined by the kinematic
      variables fixed by computeKinematics   */
  virtual LVlist buildVectors() = 0;
  /*  use the results of computeKinematics to compute the list
   of 4-vectors determined by the kinematic variables */
  double simplecrosssection();

};
```

Figure B.3: Public methods of the class `process`.

B.4.1 Variable documentation of a shower event

The parton shower simulator `shower` is constructed as the subclass of a general interface for high energy physics interaction processes `process`. This class defines the basic operations like computing the kinematics for an interaction and documenting all the final states. The public methods of `process` are shown in Fig. B.3. In Fig. B.4, the virtual functions left blank in the class `process` get implemented specifically for the high energy gluon showering of the process $e^+e^- \to$ hadrons.

Member function `computeKinematics` determine whether the event, boosted longitudinally by β, satisfies the kinematic cuts defining the cross section. It also computes the kinematic variables which determine the final-state momenta and writes the answers into appropriate class variables of the process. Note that the computation of kinematic variables can be aborted whenever the variables are determined to lead to invalid kinematics. This function should return 1 for valid kinematics, 0 for invalid kinematics. The variable J is the Jacobian of the transformation to the variables X_i from the usual variables for expressing differential cross sections.

Member function `crosssection()` computes the cross section (in fb) from the kinematic variables fixed by `computeKinematics`. The cross section should be returned by filling the class variable cs(-1,1,-1,1), a 3×3 matrix which can hold the cross sections for various initial helicities from [-1,-1] to [1,1].

Member function `simplecrosssection()` acts as a check on the cross section computation, this function produces $\frac{1}{4} \times prefactor \times \sum_{i,j} |Camps_{ij}|^2$.

In Fig. B.4, `shower` class creates QCD showers at the leading order in $1/N$. Its integration variables for each gluon are: X[5n-4], which gives the Q^2, X[5n-3] which gives the z, X[5n-2] which gives the azimuthal angle ϕ, X[5n-1] which gives the branching, and finally, an optional X[5n], which is used to modify the angular ordering, currently it is not being used in this version. It is worth mentioning that the vertices are labelled i = 0, 1, .. , Ngmax. 0 indicates $q\overline{q}$ vertex. The partons are labelled j = 1(q), 2($\overline{q}$) , 3, 4, .., Ngmax +2. At vertex i, parton to its left is called "left", parton to its right is called "right". Their values will be 0 if the parton is not realized. Its previous vertex is called "origin", its next vertex to the left is called "nextl", its next vertex to the right is called "nextr". Again they will be 0 if unassigned. "showerlist" is the list of final particle 4-vectors in color order.

```
class shower: public process{
 public:
  shower();
  int computeKinematics(double J, DVector X, double s, double beta);
  void crosssection();
  LVlist buildVectors();
  LEvent buildEvent();
  int npartons();
 protected:
  int Ng;  /* number of realized gluons */
  int Nv;  /* number of vertices  */
  DVector Q, t, z, phi;
  double tmin, wquark, wgluon;
  IVector origin, left, right, nextl, nextr, type;
  IVector birth, recent;
  DVector currentz;
  IVector showerlist;
  LVlist showervectors;
  double Qtot(double Q);
  double ttoQ(double t);
  double p(double Q1, double Q2, double Q3);
  double E2(double Q1, double Q2, double Q3);
  double E3(double Q1, double Q2, double Q3);
  double qnumerator(double z);
  double gnumerator(double z);
  double qSudakov(double zmin);
  double gSudakov(double zmin);
  int locate(int j);
  void makeShowerVectors();
  /* extra methods and data for interference of amplitudes  */
  IVector fhelicities, ihelicities;
  bool angulorder;
  void hLabel(IVector & helicities, int Label);
  complex qsplit(int hg, double z, double phi);
  complex qbarsplit(int hg, double z, double phi);
  complex gsplit(int hg1, int hg2, int hg3, double z, double phi);
  void AngOrder();
};
```

Figure B.4: Public methods of the class `shower`.

B.4.2 Construction of a shower event

Fig. B.5 starts establishing a shower event by having all the vectors properly initialized, and using input s(center of mass energy) to fix the scale of the coupling constant. Notice that the boolean variable "angulorder" is initialized to be "true", later to be checked by the angular ordering process. Fig. B.6 shows the choice of a position to insert gluon using the scheme discussed in Sec. 3.4. First Poisson selection of next t for gluon emission is performed, then the position of its insertion is decided. Fig. B.7 instantiates the gluon and determines its relation to all the other existing gluons.

B.4.3 Boosting of a shower event

Fig. B.8 implements the operation that the partons are listed in the order of the color chain, as later to be fed into a hadronization process. Fig. B.9 then uses the information of partons' relevant positions towards each other to boost them backward until all the partons are in the center-of-mass frame.

```
computeKinematics(double & J, DVector & X, double s, double beta){
  Q.zero(); t.zero(); z.zero(); phi.zero();
  origin.zero(); left.zero(); right.zero();
nextl.zero(); nextr.zero();
  type.zero();
  currentz.zero(); birth.zero(); recent.zero();
  angulorder = true;
  Npre++;
  int i, j, k;
  Q[0] = sqrt(s);
  if (Q[0] < Qmin) return 0;
  Q[1] = sqrt(s);
  double tcurrent = Qtot(Q[0]);
  t[0] = tcurrent;
  currentz[1] = 1.0;
  currentz[2] = 1.0;
  birth[1] = 0;
  birth[2] = 0;
  recent[1] = 0;
  recent[2] = 0;
  left[0] = 1;
  right[0] = 2;
  z[0] = 0.0;
  phi[0] = 0.0;
  J = 1.0;
      /* build a directory of Q, z, phi, links */
  Ng = 0;
  Nv = 0;
```

Figure B.5: Initialization

```
for (i = 1; i <= Ngmax; i++){
  double tremaining = tcurrent - tmin;
  double wtotal = 2.0 * wquark + Ng * wgluon;
  double y = X[5*i-4];
  if (y <= exp(- tremaining * wtotal))  break;
  Nv++;
  tcurrent += log(y)/wtotal;
  t[i] = tcurrent;
  Q[i] = ttoQ(tcurrent);
  phi[i] = 2.0 * PI * X[5*i-2];
  int leftright = 0;
  double insert =  wtotal * X[5*i-1];
  if (insert <= wquark){
          j = 1;
          type[i] = qtoqgtype;
  J /=  wquark;
          z[i] = exp(- X[5*i-3] * lzmin);
          J *= lzmin;
  } else if (insert <= 2.0*wquark){
          j = 2;
          type[i] = qbtoqbgtype;
  J /=  wquark;
          leftright = 1;
          z[i] = exp(- X[5*i-3] * lzmin);
          J *= lzmin;
  } else {
  int iinsert = (int)(2.0 *(insert - 2.0 * wquark)/wgluon);
          j = 3 + iinsert/2;
  J /=  0.5 * wgluon;
          leftright = iinsert%2;
          type[i] = gtoggtype;
          double zz =  exp(- lzmin2 * (1.0-X[5*i-3]));
  z[i] = zz/(1.0 + zz);
          J *= lzmin2;
  }
```

Figure B.6: gluon insertion

```
    int ii  = recent[j];
    double znext = z[i] * currentz[j];
    if (znext > zmin){
      double Qo = Q[ii] - Q[i];
      if (nextl[ii] > 0) Qo -= Q[nextl[ii]];
      if (nextr[ii] > 0) Qo -= Q[nextr[ii]];
      if (Qo > 0.0){
/* realize the parton */
          Ng++;
          int Np = Ng+2;
          birth[Np] = i;
          recent[Np] = i;
  recent[j] = i;
          currentz[Np] = znext;
          currentz[j] *= (1-z[i]);
          if (leftright == 0){
    left[i] = j;
            right[i] = Np;
          }  else {
    left[i] = Np;
            right[i] = j;
          }
          origin[i] = ii;
          if (j == left[ii]){
               nextl[ii] = i;
          } else if (j == right[ii]){
       nextr[ii] = i;
          } else {
               therror("  inconsistent shower tree ");
          }
      }
    }
  }
  return 1;
}
```

Figure B.7: gluon instantiation

```
int shower:: locate(int j){
  int k = 1;
  while (j != showerlist[k]) k++;
  return k;
}
void shower::makeShowerVectors(){
  showervectors.zero();
   clusterparent.zero();
  int i, j, jl, jr, jp, k, kp, kl, kr, km;
  int leftright; double mycos;
  showerlist[1] = 1;
  showerlist[2] = 2;
  for (i = 1; i <= Ngmax; i++){
    if (left[i] == 0) continue;
    /* if parton created at i is not realized */
    jl = left[i];
    jr = right[i];
    if (jl < jr){
      kp = locate(jl);
      for (k = i+1; k > kp; k--)  showerlist[k+1] = showerlist[k];
      showerlist[kp+1] = jr;
    } else {
      kp = locate(jr);
      for (k = i+1; k >= kp; k--)  showerlist[k+1] = showerlist[k];
      showerlist[kp] = jl;
    }
  }
```

Figure B.8: color ordering

```
  for (i = Ngmax; i >= 0 ; i--){
    if (left[i] == 0) continue;
    jl = left[i];  jr = right[i];
    jp = MIN(jl,jr);  double Q1 = Q[i];
    double Qleft, Qright, Eleft, Eright, plr;
    kl = locate(jl);  kr = locate(jr);
    if (clusterparent[kl] == 0){
      if (clusterparent[kr] == 0) {
showervectors[kl][0] = Q1/2.0;
showervectors[kl][3] = Q1/2.0;
showervectors[kr][0] = Q1/2.0;
showervectors[kr][3] = -Q1/2.0;
      }  else {
        while(clusterparent[kr+1] ==jr){ kr++;}
        Qright = Q[nextr[i]];
        plr = p(Q1,0.0,Qright);
        Eright = E3(Q1,0.0,Qright);
showervectors[kl][0] = plr;
        showervectors[kl][3] = plr;
        showervectors.boostset(kl+1,kr,-plr/Eright);
      }
    } else { if (clusterparent[kr] == 0) {
        while(clusterparent[kl-1] == jl){ kl--;}
        Qleft = Q[nextl[i]];
        plr = p(Q1,Qleft,0.0);
        Eleft = E2(Q1,Qleft,0.0);
showervectors[kr][0] = plr;
showervectors[kr][3] = -plr;
        showervectors.boostset(kl,kr-1,plr/Eleft);
      }
```

```
else { km = kl;
    while(clusterparent[kl-1] == jl ){ kl--;}
    while(clusterparent[km+1] == jl ){ km++;}
    while(clusterparent[kr+1] == jr ){ kr++;}
    Qleft = Q[nextl[i]];
    Qright = Q[nextr[i]];
    plr = p(Q1,Qleft,Qright);
    Eleft = E2(Q1,Qleft,Qright);
    Eright = E3(Q1,Qleft,Qright);
    showervectors.boostset(kl,km,plr/Eleft);
    showervectors.boostset(km+1,kr,-plr/Eright);}}
for (k = kl; k <= kr; k++) clusterparent[k] = jp;
showervectors.rotateinplaneset(kl,kr,1.0 - 2.0*z[i]);
showervectors.rotateset(kl,kr,phi[i]);}}
```

Figure B.9: boost

B.4.4 Angular Ordering

Fig. B.10 implements the angular ordering discussed in Sec. 3.5.1. If the checking finds out that this shower event violates the global angular ordering, then the boolean variable “angulorder” is set to be false. Consequently, the value of the cross section for this event in Fig. B.11 is set to be 0. Otherwise, the calculation of the cross section will proceed according to Sec. 3.4

B.4.5 Jet Algorithms

Fig. B.12, Fig. B.13 and Fig. B.14 are three jet algorithms which we use to cluster our final partons into 3 jets. The jet counting algorithm with fixed y_{cut} is presented in Fig. B.15. The clustering algorithm is presented in Fig. B.16. It returns the jet variables x_1 , x_2 in the DVector VX.

B.4.6 reweighting

Fig. B.17 presents the reweighting scheme we have discussed in 3.5.2.

```
void shower::AngOrder(){
  int jl, jr, kl, kr;
  DVector angle(0, Ngmax);
  angle.zero();
  for (int i = 0; i <= Ngmax; i++){
    if (left[i] == 0) continue;
    if (!(angulorder)) break;
    jl = left[i];
    jr = right[i];
    kl = locate(jl);
    kr = locate(jr);
    LVector L1 = showervectors.readout(kl);
    LVector L2 = showervectors.readout(kr);
    angle[i] = dot(L1, L2)/(L1[0]*L2[0]);
    if (i>0) {
      int ii = origin[i];
      if (ABS(angle[ii])>1e-10)
if (angle[i]>angle[ii]) {
  angulorder = false;
  double dd = Q[i]/Q[ii];
}
    }
  }
}
```

Figure B.10: angular ordering

```
void shower::crosssection(){

  double wgt = 1.0;
  for (int i = 1; i <= Nv; i++){
  if (type[i] == qtoqgtype || type[i] == qbtoqbgtype){
      wgt *= qnumerator(z[i]);
    }  else {
      wgt *= gnumerator(z[i]);
    }
  }
  makeShowerVectors();
  AngOrder();
  if (!(angulorder)) {cs.zero(); return;}
  cs[1][1] = wgt;
  cs[1][-1] = wgt;
  cs[-1][1] = wgt;
  cs[-1][-1] = wgt;
  Nvalid++;
}
```

Figure B.11: cross section

```
double LVlist::FindMin(int & i, double Q) {

  int j = 1;
  double f = SQR(Q);
  while (j<index) {
    LVector L1 = readout(j);
    LVector L2 = readout(j+1);
    tVector t1 = L1.tvector();
    tVector t2 = L2.tvector();
    t1 /= t1.length();
    t2 /= t2.length();
 double d1 = 2*MIN(SQR(L1[0]), SQR(L2[0]))*(1-t1*t2);
    if (d1<f) {
      f = d1;
      i = j;
    }
    j++;
  }
  return f;
}
```

Figure B.12: k_T algorithm

```
double LVlist::FindMin(int & i, double Q) {

  int j = 1;
  double f = SQR(Q);
  while (j<index) {
    LVector L1 = readout(j);
    LVector L2 = readout(j+1);
    tVector t1 = L1.tvector();
    tVector t2 = L2.tvector();
    t1 /= t1.length();
    t2 /= t2.length();
    double d1 = 2*L1[0]*L2[0]*(1-t1*t2);
    if (d1<f) {
      f = d1;
      i = j;
    }
    j++;
  }
  return f;
  }
```

Figure B.13: JADE algorithm

```
double LVlist::FindMin(int & i, double Q) {
  int j = 1;
  double f = SQR(Q);
  while (j<index) {
    LVector L1 = readout(j);
    LVector L2 = readout(j+1);
    LVector L = L1+L2;
    double p = L.masssq();
    if (p<f) {
      f = p;
      i = j;
    }
    j++;
  }
  return f;
  }
```

Figure B.14: E0 algorithm

```
int LEvent::NumberJets(double ECM, double cut) {
  LVlist jet(LV);
  double fm = 2*SQR(ECM);
  while (jet.index>2) {
    int i1;
    fm = jet.FindMin(i1, ECM);
    if (fm/SQR(ECM)>cut)
      return jet.index;
    else {
      LVector L1 = jet.readout(i1);
      LVector L2 = jet.readout(i1+1);
      tVector t1 = L1.tvector();
      tVector t2 = L2.tvector();
      t1 += t2;
      double d1 = t1.length();
      double E1 = L1[0]+L2[0];
      double r1 = E1/d1;
      t1 *= r1;
      LVector  L(E1, t1);
      jet.read(i1, L);
      jet.remove(i1+1);
    }
  }
  return 2;
}
```

Figure B.15: jet counting algorithm

```
void LEvent::IdentifyJets(double Q, DVector & VX){
  LVlist jet(LV); double ycut;
  int i,j, r, col; LVector L1, L2;
  double fm = 2*SQR(Q);
  while (jet.index>2) {
    if (jet.index==3)  break;
    else { int i1;
      fm = jet.FindMin(i1, Q);
      L1 = jet.readout(i1);
      L2 = jet.readout(i1+1);
      tVector t1 = L1.tvector();
      tVector t2 = L2.tvector();
      t1 += t2;
      double d1 = t1.length();
      double E1 = L1[0]+L2[0];
      double r1 = E1/d1; t1 *= r1;
      LVector  L(E1, t1);
      jet.read(i1, L);
      jet.remove(i1+1);
    }
  }
  if (jet.index==3) {
    LVector L1 = jet.readout(1);
    double E1 = L1[0];
    tVector t1 = L1.tvector();
    double p1 = t1.length();
    LVector L3 = jet.readout(3);
    double E3 = L3[0];
    tVector t3 = L3.tvector();
    double p3 = t3.length();
    VX[1] = 2*E1/Q;
    VX[2] = 2*E3/Q;
  }
  else
    return;
}
```

Figure B.16: clustering algorithm

```
double LEvent::ExactSurf(DVector X) {

  return (SQR(X[1])+SQR(X[2]));
}

double LEvent::Approxsurf(DVector VX) {
  double p;
  double d1 =1.0 + SQR(VX[1]+VX[2]-1)/SQR(VX[2]);
  double d2 =1.0 + SQR(VX[1]+VX[2]-1)/SQR(VX[1]);
  if (VX[1]>.5 && VX[2]>.5) {
    return .5*(d1*(2-1.0/VX[2])+d2*(2-1.0/VX[1]));
  }
  else if (VX[1]>.5)
    return d2*(2-1/VX[1]);
  else if (VX[2]>.5)
    return d1*(2-1/VX[2]);
}

double LEvent::Reweight(double ECM, DVector & VX){
  double q;
  IdentifyJets(ECM, VX);
  double f = ExactSurf(VX);
  f /= Approxsurf(VX);
  return f;
}
```

Figure B.17: reweighting algorithm

Bibliography

[1] G. P. Lepage, J. Comput. Phys. **27**, 192 (1978).

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

[3] S. Kawabata, Comput. Phys. Commun. **41**, 127 (1986).

[4] S. Kawabata, Comp. Phys. Commun. **88**, 309 (1995).

[5] R. Kleiss and R. Pittau, Comput. Phys. Commun. **83**, 141 (1994), hep-ph/9405257.

[6] F. A. Berends, R. Pittau, and R. Kleiss, Comput. Phys. Commun. **85**, 437 (1995), hep-ph/9409326.

[7] E. E. Boos *et al.*, Int. J. Mod. Phys. **C5**, 615 (1994).

[8] F. Maltoni and T. Stelzer, JHEP **02**, 027 (2003), hep-ph/0208156.

[9] T. Ohl, Comput. Phys. Commun. **120**, 13 (1999), hep-ph/9806432.

[10] W. Kilian, WHIZARD 1.0, LC-TOOL-2001-039.

[11] T. Ohl, hep-ph/0011287.

[12] M. E. Peskin, hep-ph/9910519.

[13] C. M. Bishop, *Neural Networks for Pattern Recognition* (Oxford University Press, 1995).

[14] S. Jadach, Comput. Phys. Commun. **130**, 244 (2000), physics/9910004.

[15] S. Jadach, Comput. Phys. Commun. **152**, 55 (2003), physics/0203033.

[16] H.-U. Bengtsson and T. Sjöstrand, Comput. Phys. Commun. **46**, 43 (1987).

[17] M. Bengtsson and T. Sjöstrand, Nucl. Phys. **B289**, 810 (1987).

[18] T. Sjöstrand, Comput. Phys. Commun. **82**, 74 (1994).

[19] T. Sjöstrand *et al.*, Comput. Phys. Commun. **135**, 238 (2001), hep-ph/0010017.

[20] G. Marchesini and B. R. Webber, Nucl. Phys. **B310**, 461 (1988).

[21] G. Marchesini *et al.*, Comput. Phys. Commun. **67**, 465 (1992).

[22] G. Corcella *et al.*, (2002), hep-ph/0210213.

[23] R. K. Ellis, W. J. Stirling, and B. R. Webber, Cambridge Monogr. Part. Phys. Nucl. Phys. Cosmol. **8**, 1 (1996).

[24] S. Bethke, J. Phys. **G17**, 1455 (1991).

[25] M. E. Peskin and D. V. Schroeder, *An Introduction to quantum field theory* (Addison-Wesley, 1995).

[26] S. Frixione, (2002), hep-ph/0211435.

[27] A. P. Dempster, N. M. Laird, and D. B. Rubin, J. Royal Stat. Soc. **B39** (1977).

[28] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, J. Chem. Phys. **21**, 1087 (1953).

[29] M. Skrzypek and S. Jadach, Z. Phys. **C49**, 577 (1991).

[30] C. T. Potter, J. E. Brau, and M. Iwasaki, eConf **C010630**, P118 (2001).

[31] D. M. Asner, J. B. Gronberg, and J. F. Gunion, Phys. Rev. **D67**, 035009 (2003), hep-ph/0110320.

[32] M. Peskin, Pandora, http://www-sldnt.slac.stanford.edu/nld/new/Docs/Generators/PANDORA.htm.

[33] K. Yokoya and P. Chen, *Frontiers of Particle Beams: Intensity Limitations* (Springer Verlag, 1992).

[34] V. G. Gorshkov, V. N. Gribov, L. N. Lipatov, and G. V. Frolov, Phys. Lett. **22**, 671 (1966).

[35] T. Kinoshita, J. Math. Phys. **3**, 650 (1962).

[36] M. Bengtsson and T. Sjöstrand, Phys. Lett. **B185**, 435 (1987).

[37] Richardson, personal communication., We are grateful to P. Richardson and B. Webber for providing us with this unpublished data.

[38] P. N. Burrows, (1997), hep-ex/9705013.

[39] A. De Rujula, J. R. Ellis, E. G. Floratos, and M. K. Gaillard, Nucl. Phys. **B138**, 387 (1978).

[40] E. Farhi, Phys. Rev. Lett. **39**, 1587 (1977).

[41] DELPHI, P. Abreu *et al.*, Z. Phys. **C54**, 55 (1992).

[42] T. M. Cover and J. A. Thomas, *Elements of Information Theory* (John Wiley & Sons, Inc., 1991).

[43] Open Source, Gnuplot, http://www.gnuplot.info.