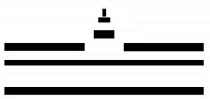
Westfälische Wilhelms-Universität



Institut für Theoretische Physik

CPU and GPU aided methods of solving problems in QCD

23. September 2015

Oliver Benjamin Smith

o_smit01@uni-muenster.de

Contents

1 Introduction

The aim of this thesis will be to show the benefits of computational solutions to problems in theoretical particle physics using examples in quantum chromodynamics (QCD) and quantum electrodynamics (QED). Two major problems will be tackled by these means. The first being the calculation of the trace of the product of colour matrices in QCD. The second being the numerical problem of solving the high-dimensional integrals that stem from calculating the cross section of particle interactions.

2 A short history of particle physics

Since the dawn of science, scientists have tried to find structure in nature using Occam's razor to eliminate the complex and derive a most simple explanation for natural phenomena which still enables us to make predictions. The first attempts to find out what everything is made of was probably the model of the four elements earth, fire, water and air. Though simple, this model does not bring forth any predictions or clearly defined structure of what it actually means to be made of those four elements.

Not much later Mendeleev asserted that there is structure in matter which could actually be quantitatively verified when deriving the periodic table of elements. Due to the sheer amount of different elements and the inherent substructures one can expect there to be a simpler list of subatomic particles which in turn make up theses elements. Following this principle the proton and neutron as components of the nuclei were assumed to exist and in sequence verified by experiments. The force "gluing" these together is called the strong force, hence the name "gluon" for the respective force carrier.

When it turned out that both the proton and the neutron were just two of a vast number of baryons one was at a point once again where the structure in the known particles suggests a simple substructure. The building blocks of baryons were called quarks. Quarks manifest themselves as being the only particles which interact in form of the strong force while all particles made up of quarks are called hadrons with the subgroups of baryons made up of three and mesons made up of two quarks. All

Table 1: Building blocks of elementary particles and gauge bosons for interactions

Name	Spin	Baryon Number	Lepton Number	Charge
Quarks				
u	$\frac{1}{2}$	$\frac{1}{3}$	0	$\frac{2}{3}$
d	$\frac{1}{2}$	$\frac{1}{3}$	0	$-\frac{1}{3}$
Leptons				
e	$\frac{1}{2}$	0	1	-1
ν	$\frac{1}{2}$	0	1	0
Gauge bosons				
γ	1	0	0	0
$W^{\pm}, Z$	1	0	0	$\pm$1,0
g_i	1	0	0	0

elementary particles that do not interact via strong interaction (such as electrons) are called leptons.

Since we know that a proton is a baryon (B=1, L=0) and has a charge of +1 we can deduce that a proton is a uud bound state (see tab.1). Analogously we know that a neutron is a udd bound state of quarks. For these particles the model of the quarks does not run into any problems so far since both states can be accomplished by having three quarks in different states. The Δ^{++}–baryon, being a uuu bound state, yields the problem of having three identical particles (all three spins must be $\frac{1}{2}$ since the spin is $\frac{3}{2}$ in total) in a completely symmetric state. This contradicts the spin–statistic theorem which states that any fermion state must be totally antisymmetric.

Since the Δ^{++} state can be observed this must mean that quarks have another quantum number which can differentiate between the three u–quarks. This problem as well as others such as the apparent non existence of qq (quark-quark) or $\bar{q}\bar{q}$ (antiquark-antiquark) bound states can be solved by the introduction of the "colour charge". Over time more quarks and leptons have been confirmed as seen in table 2.

Table 2: Proliferation of quarks and leptons in the standard model

	Quarks	
up	charm	top
down	strange	beauty
	Leptons	
electron	muon	tau
e–neutrino	μ–neutrino	τ–neutrino

3 Calculations in quantum electrodynamics

3.1 A brief introduction

We know from experiments that particle production is a measurable phenomenon and we know that any physical theory must be in accordance with special relativity. Schrödingers equation satisfies neither of those requirements. A first attempt to solve these problems was brought forth by Oskar Klein and Walter Gordon in 1926 and has since been referred to as the Klein-Gordon equation:

$$\left(\Box + \frac{m^2c^2}{\hbar^2}\right)\psi = 0 \tag{1}$$

It solves the problem of Schrödingers equation not being Lorentz invariant but does not yet allow for particles with spin. This problem was solved in 1928 by Paul Dirac:

$$(\imath\gamma^\mu\partial_\mu - m)\,\psi(x) = 0 \tag{2}$$

with

$$\partial_\mu = \left(\frac{\partial}{\partial t}, \frac{\partial}{\partial x^1}, \frac{\partial}{\partial x^2}, \frac{\partial}{\partial x^3}\right)^T \tag{3}$$

$$\gamma^0 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \gamma^i = \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix}, \tag{4}$$

σ^i being the Pauli matrices. Since we will need this later on we shall discuss a few properties of the γ–matrices. We shall have a look at a few properties of the gamma

matrices. For $i \in \{1, 2, 3\}$ we have:

$$\left[\gamma^0, \gamma^0\right]_+ = 2 \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = 2 \cdot \mathbf{1} \tag{5}$$

$$\left[\gamma^0, \gamma^i\right]_+ = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix} + \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{6}$$

$$= \begin{pmatrix} 0 & \sigma^i \\ \sigma^i & 0 \end{pmatrix} + \begin{pmatrix} 0 & -\sigma^i \\ -\sigma^i & 0 \end{pmatrix} = 0 \tag{7}$$

$$\left[\gamma^i, \gamma^j\right]_+ = \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & \sigma^j \\ -\sigma^j & 0 \end{pmatrix} + \begin{pmatrix} 0 & \sigma^j \\ -\sigma^j & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & \sigma^j \\ -\sigma^j & 0 \end{pmatrix} \tag{8}$$

$$= - \begin{pmatrix} [\sigma^i, \sigma^j]_+ & 0 \\ 0 & [\sigma^i, \sigma^j]_+ \end{pmatrix} = -2\delta_{ij}\mathbf{1} \tag{9}$$

This can be written as

$$[\gamma^\mu, \gamma^\nu]_+ = 2 \cdot g_{\mu\nu} \cdot \mathbf{1} \tag{10}$$

in short form with the Lorentz metric

$$g_{\mu\nu} = g^{\mu\nu} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{11}$$

as known from special relativity. Furthermore we shall use the shorthand Feynman slash notation

$$\not{a} := \gamma^\mu a_\mu = \gamma_\mu a^\mu \tag{12}$$

We shall now show a few equations that we will need later on.

$$\gamma^\mu\gamma_\mu = \gamma^0\gamma^0 - \gamma^1\gamma^1 - \gamma^2\gamma^2 - \gamma^3\gamma^3 \tag{13}$$
$$= 4 \cdot \mathbf{1} \tag{14}$$
$$\gamma^\mu \not{p} \gamma_\mu = \gamma^\mu\gamma^\rho g_{\nu\rho}p^\nu\gamma_\mu \overset{(10)}{=} \left(2g_{\mu\rho} \cdot \mathbf{1} - \gamma^\rho\gamma^\mu\right)\gamma_\mu g_{\nu\rho}p^\nu = 2\not{p} - 4\not{p} \tag{15}$$
$$= -2\not{p} \tag{16}$$
$$\mathrm{Tr}\left[\not{p}\not{q}\right] = \mathrm{Tr}\left[\gamma^\mu\gamma^\nu p_\mu q_\nu\right] = \mathrm{Tr}\left[\left(2g_{\mu\nu} \cdot \mathbf{1} - \gamma^\nu\gamma^\mu\right)p_\mu q_\nu\right] = 8 \cdot pq - \mathrm{Tr}\left[\not{p}\not{q}\right] \tag{17}$$
$$\Rightarrow \mathrm{Tr}\left[\not{p}\not{q}\right] = 4pq \tag{18}$$
$$\mathrm{Tr}\left[\not{a}\not{b}\not{c}\not{d}\right] = \mathrm{Tr}\left[\gamma^\mu\gamma^\nu a_\mu b_\nu \not{c}\not{d}\right] \overset{(10)}{=} -\mathrm{Tr}\left[\not{b}\not{a}\not{c}\not{d}\right] + 2 \cdot ab \cdot \underbrace{\mathrm{Tr}\left[\not{c}\not{d}\right]}_{4 \cdot cd} \tag{19}$$
$$\overset{(10)(18)}{=} \mathrm{Tr}\left[\not{b}\not{c}\not{a}\not{d}\right] - 8 \cdot ac \cdot bd + 8 \cdot ab \cdot cd \tag{20}$$
$$\overset{(10)(18)}{=} -\mathrm{Tr}\left[\not{b}\not{c}\not{d}\not{a}\right] + 8 \cdot ad \cdot bc - 8 \cdot ac \cdot bd + 8 \cdot ab \cdot cd \tag{21}$$
$$\overset{\text{trace is cyclic}}{\Rightarrow} \mathrm{Tr}\left[\not{a}\not{b}\not{c}\not{d}\right] = 4\left(ab \cdot cd - ac \cdot bd + ad \cdot bc\right) \tag{22}$$

3.2 Feynman diagrams

Feynman Diagrams were originally sketches by Richard Feynman noting how he pictured particle interactions. Each line and vertex in a diagram corresponds to a term in the interaction $\mathcal{M}$. By knowing the interaction we can then calculate the transition amplitude

$$A_{\varphi\to\psi} = \langle\psi|\mathcal{M}|\varphi\rangle \tag{23}$$

between two states $|\varphi\rangle$ and $|\psi\rangle$ and the corresponding transition probability

$$P_{\varphi\to\psi} = \left|A_{\varphi\to\psi}\right|^2. \tag{24}$$

To get the probability of any end state one must therefore integrate $P_{\varphi\to\psi}$ over all possible input states. To correctly introduce Feynman diagrams would go greatly beyond the scope of this thesis which is why we shall propose the rules we need as seen in table 3.

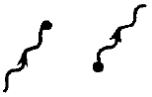
		Multiplicative Factor
• **External Lines**		
Spin 0 boson (or antiboson)		1
Spin $\frac{1}{2}$ fermion (in, out)		$u, \bar{u}$
antifermion (in, out)		$\bar{v}, v$
Spin 1 photon (in, out)		$\epsilon_\mu, \epsilon_\mu^*$
• **Internal Lines—Propagators (need $+i\varepsilon$ prescription)**		
Spin 0 boson		$\frac{i}{p^2 - m^2}$
Spin $\frac{1}{2}$ fermion		$\frac{i(\not{p} + m)}{p^2 - m^2}$
Massive spin 1 boson		$\frac{-i\left(g_{\mu\nu} - p_\mu p_\nu / M^2\right)}{p^2 - M^2}$
Massless spin 1 photon (Feynman gauge)		$\frac{-ig_{\mu\nu}}{p^2}$
• **Vertex Factors**		
Photon—spin 0 (charge $-e$)		$ie(p + p')^\mu$
Photon—spin $\frac{1}{2}$ (charge $-e$)		$ie\gamma^\mu$

Table 3: Feynman rules for calculating $i\mathcal{M}$ [3 p.149]

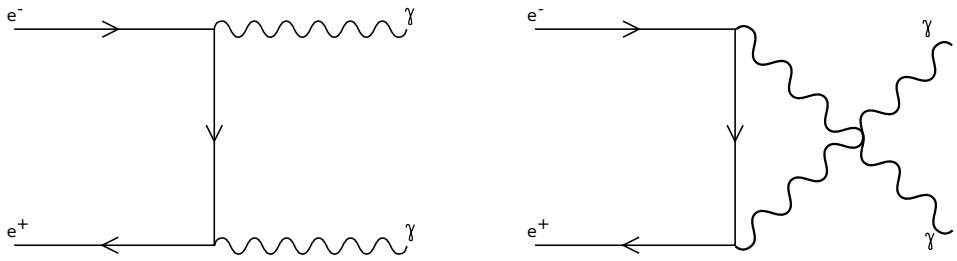


Figure 1: Feynman diagrams of electron-positron annihilation

3.3 $e^-e^+ \to 2\gamma$

We shall calculate the differential cross section of a electron-positron annihilation (see fig.1) assuming all masses are negligible since we want to consider high energy collisions. We will use the indices a, b for the electron momenta and the indices $1, 2$ for the photon momenta. With the Feynman rules (see tab.3) we get the following equation for $i\mathcal{M}_1$ and $i\mathcal{M}_2$, the parts of $\mathcal{M}$ corresponding to the two possible $e^-e^+ \to 2\gamma$ annihilations (see fig.1):

$$i\mathcal{M}_1 = \bar{v}(p_b) \cdot ie\gamma^\nu \frac{i\left(\not{p}_a - \not{p}_1\right)}{\left(p_a - p_1\right)^2} ie\gamma^\mu u(p_a)\epsilon^*_{\mu,1}\epsilon^*_{\nu,2} \tag{25}$$

$$i\mathcal{M}_2 = \bar{v}(p_b) \cdot ie\gamma^\nu \frac{i\left(\not{p}_a - \not{p}_2\right)}{\left(p_a - p_2\right)^2} ie\gamma^\mu u(p_a)\epsilon^*_{\mu,1}\epsilon^*_{\nu,2} \tag{26}$$

Since both amplitudes have the same form we shall focus on calculating the absolute square of $\mathcal{M}_1$:

$$\left|\mathcal{M}_1\right|^2 = -\bar{v}_b \cdot ie\gamma^\nu \frac{i\left(\not{p}_a - \not{p}_1\right)}{\left(p_a - p_1\right)^2} ie\gamma^\mu u_a \cdot \bar{u}_a ie\gamma^\rho \frac{i\left(\not{p}_a - \not{p}_1\right)}{\left(p_a - p_1\right)^2} ie\gamma^\sigma v_b \cdot \epsilon^*_{\mu,1}\epsilon^*_{\nu,2}\epsilon_{\rho,1}\epsilon_{\sigma,2} \tag{27}$$

Since we must consider all possible spins of the particles we will try to calculate the average over all spins using following identities (see [1]):

$$\sum_s u_s \bar{u}_s = \not{p} \tag{28}$$

$$\sum_s \epsilon^\mu_s \epsilon^{\nu *}_s = -g^{\mu\nu} \tag{29}$$

$$\tag{30}$$

and the Mandelstam variables:

$$s = (p_a + p_b)^2 = (p_1 + p_2)^2 = 2p_a p_b = 2p_1 p_2 \tag{31}$$
$$t = (p_a - p_1)^2 = (p_b - p_2)^2 = -2p_a p_1 = -2p_b p_2 \tag{32}$$
$$u = (p_a - p_2)^2 = (p_b - p_1)^2 = -2p_a p_2 = -2p_b p_1 \tag{33}$$
$$\Rightarrow s + t + u = 0 \tag{34}$$

This gives us:

$$\overline{|\mathcal{M}_1|}^2 = \frac{e^4}{t^2}\frac{1}{n_{spins}}\sum_{spins} v_{b,i}\gamma^\nu_{i,j}(\not{p}_{a,j,k} - \not{p}_{1,j,k})\gamma^\mu_{k,l}u_{a,l}u_{a,m}\gamma^\rho_{m,n}(\not{p}_{a,n,o} - \not{p}_{1,n,o})\gamma^\sigma_{o,p}v_{b,p}\cdot\epsilon^*_{\mu,1}\epsilon^*_{\nu,2}\epsilon_{\rho,1}\epsilon_{\sigma,2} \tag{35}$$

$$= \frac{e^4}{4t^2}\mathrm{Tr}\left[\not{p}_b\gamma^\nu\left(\not{p}_a - \not{p}_1\right)\gamma^\mu\not{p}_a\gamma^\rho\left(\not{p}_a - \not{p}_1\right)\gamma^\sigma g_{\mu\rho}g_{\nu\sigma}\right] \tag{36}$$
$$= \frac{e^4}{4t^2}\mathrm{Tr}\left[\gamma^\nu\not{p}_b\gamma_\nu(\not{p}_a - \not{p}_1)\gamma^\mu\not{p}_a\gamma_\mu(\not{p}_a - \not{p}_1)\right] \tag{37}$$
$$= \frac{e^4}{4t^2}\cdot 4\cdot\mathrm{Tr}\left[\not{p}_b(\not{p}_a - \not{p}_1)\not{p}_a(\not{p}_a - \not{p}_1)\right] \tag{38}$$
$$\overset{(22)}{=} \frac{e^4}{t^2}\cdot\left[8p_b(p_a - p_1)\cdot p_a(p_a - p_1) - 4p_a p_b\cdot(p_a - p_1)^2\right] \tag{39}$$
$$= \frac{4e^4}{t^2}\left[2(p_a p_b - p_b p_1)\cdot(\underbrace{p_a p_a}_{=0} - p_a p_1) - p_a p_b(p_a - p_1)^2\right] \tag{40}$$

Writing this with the Mandelstam variables we get:

$$\overline{|\mathcal{M}_1|}^2 = \frac{4e^4}{t^2}\left[2\cdot\left(\frac{1}{2}s + \frac{1}{2}u\right)\cdot\frac{1}{2}t - \frac{1}{2}st\right] \tag{41}$$
$$= \frac{4e^4}{t^2}\cdot\left[2\cdot\left(-\frac{1}{2}t\cdot\right)\frac{1}{2}t - \frac{1}{2}st\right] \tag{42}$$
$$= \frac{4e^4}{t^2}\cdot\left[-\frac{1}{2}t^2 + \frac{1}{2}(u+t)t\right] \tag{43}$$
$$= 2e^4\frac{ut}{t^2} = 2e^4\frac{u}{t} = 2e^4\frac{(p_a - p_2)^2}{(p_a - p_1)^2} \tag{44}$$
$$\overset{1\to 2}{\underset{2\to 1}{\Rightarrow}}\overline{|\mathcal{M}_2|}^2 = 8e^4\frac{(p_a - p_1)^2}{(p_a - p_2)^2} = 2e^4\frac{t}{u} \tag{45}$$

Therefore we can now calculate the squared amplitude:

$$\overline{|\mathcal{M}|}^2 = 2e^4\left(\frac{u}{t} + \frac{t}{u}\right) \tag{46}$$

and with that we get the cross section [3 p.144]:

$$\frac{d\sigma}{d\Omega} = \frac{1}{64\pi^2 s}\overline{|\mathcal{M}|}^2 \tag{47}$$

$$= \frac{1}{64\pi^2 s} 2e^4 \left(\frac{u}{t} + \frac{t}{u}\right) \tag{48}$$

4 Calculations in quantum chromodynamics

4.1 Symmetries and group theory

Groups are a mathematical structure abiding by the following axioms:

Definition 1 *A group is a set G together with an operation $+ : G \times G \to G$ which is associative and there exists a neutral element $e \in G$ with $\forall g \in G : e + g = g + e = g$.*

More precisely we will be interested in a special case of groups called **Lie groups**. We shall first introduce these by definition and then go on to discussing their application. For the introduction of Lie groups one must first define the concept of a **differentiable manifold** which shall just be mentioned briefly here.

Definition 2 *A differentiable manifold consists of a subset $M \subset \mathbb{R}^n$, subsets $U_i \overset{open}{\subset} M$ and homeomorphisms $\varphi_i : U_i \to V_i \subset \mathbb{R}^k$ for $i \in \mathcal{I}$ such that $\varphi_i \circ \varphi_j^{-1} : \varphi_j(U_i \cap U_j) \to \varphi_i(U_i \cap U_j)$ is a diffeomorphism for any pair $i, j \in \mathcal{I}$.*

This rather abstract definition of a manifold gives us the tools to talk about differentiation on other spaces than the euclidean space since a manifold "looks like" euclidean space on a small neighbourhood of any point. Once again the Lie group is a special case of this.

Definition 3 *A **Lie group** is a group $(G, \cdot)$ which is also a differentiable manifold. In addition to this the function: $G \times G \to G : (x, y) \mapsto x^{-1}y$ shall be a smooth mapping.*

We will discuss two examples of Lie groups to get a feeling for the definition.

1. **Special orthogonal group**

   $SO(2) = \{M \in \mathbb{R}^{2\times 2} | \det M = 1\}$ multiplication being the operation of choice (49)

   This group corresponds to all rotations of the euclidean space $\mathbb{R}^2$. This means that $SO(2)$ principally has the form of the unit circle S^1 and can for example be parametrized as one would a sphere in $\mathbb{R}^3$ by stereographic projection. Furthermore it is easy to see that the mapping $SO(2)^2 \to SO(2) : (A, B) \mapsto A^{-1}B$ is a smooth mapping since the inversion of 2×2 can be explicitly calculated for matrices with a non vanishing determinant. Therefore the mapping consists of compositions of smooth functions in each component. We can easily see that this Lie group is in fact an ablian group since it is isomorphic to $\mathbb{R}/_{r\sim r+2\pi n}$. This is certainly not always the case as the next example will show.

2. **SU(n)**

   $$SU(n) = \{U \in \mathbb{C}^{n\times n} | \det U = 1, U^\dagger U = \mathbf{1}\} \tag{50}$$

   The special unitary group is an important group in physics. We have already seen SU(2) as the group representing the spin of particles and we will see that analogously the Lie group SU(3) will serve to model the colour charge of quarks.

Since continuous symmetries (as in the examples) can be viewed as Lie groups we shall take a look at Symmetries in physics. For a symmetry of a physical system we demand that any measurable outcome does not depend on this symmetry. This means that if we have a symmetry operation $U : \mathcal{H} \to \mathcal{H}$ acting on the Hilbert space of our physical states $\mathcal{H}$ the following equation must hold for any two states $\varphi, \psi \in \mathcal{H}$:

$$|\langle\varphi|\psi\rangle|^2 = |\langle U\varphi|U\psi\rangle|^2 = \left|\left\langle\varphi|U^\dagger U|\psi\right\rangle\right|^2 \tag{51}$$

By which follows, that U must be a unitary operator. Since also the energy of the system and therefore the Hamiltonian must be independent of any symmetry operations the following must also hold.

$$\langle U\varphi|H|U\psi\rangle = \left\langle\varphi|U^\dagger HU|\psi\right\rangle = \langle\varphi|H|\psi\rangle \tag{52}$$

Which translates to

$$[U, H] \doteq UH - HU = 0. \tag{53}$$

Since we will discuss symmetry groups which are Lie groups and as such have a continuous structure (unlike discrete symmetries like time inversion) we shall take a look at such symmetry operations U which can be written as an exponential function of one of the generators g_k of the Lie group, where the set of generators is called the corresponding Lie algebra:

$$U = e^{i\epsilon g_k} \tag{54}$$

Since U must be unitary we get:

$$\mathbf{1} = U^\dagger U = e^{-i\epsilon g_k^\dagger} \cdot e^{i\epsilon g_k} = e^{i\epsilon(g_k - g_k^\dagger)} \tag{55}$$

$$\overset{\forall \epsilon > 0}{\Rightarrow} g_k = g_k^\dagger \tag{56}$$

This means that all the generators of the group must be hermitian. Furthermore we define the structure factors f_{abc} of the Lie algebra through

$$[g_a, g_b] = f_{abc} g_c \tag{57}$$

Although one often does not explicitly mention this we have come across this concept before when discussing the spin. The symmetry of the spin corresponds to rotations in $\mathbb{C}^2$ which means that the Lie group is the before introduced SU(2) group. The generators of the spin group are the Pauli matrices:

$$\sigma^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{58}$$

Here we used the convention of defining the spin states up and down as the eigenvectors of σ^3.

$$\psi_+ = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \psi_- = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{59}$$

We shall now try to transfer this idea to the colour charge of quarks. The basis of the colour space shall be the three colours:

$$R = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad G = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{60}$$

Analogously to the spin the symmetry group of colour must therefore be $SU(3)$. One choice for the generators of the $SU(3)$ group are the Gell-Mann matrices:

$$\lambda_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \lambda_2 = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \lambda_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \lambda_4 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{61}$$

$$\lambda_5 = \begin{pmatrix} 0 & 0 & -i \\ 0 & 0 & 0 \\ i & 0 & 0 \end{pmatrix} \quad \lambda_6 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \lambda_7 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -i \\ 0 & i & 0 \end{pmatrix} \quad \lambda_8 = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{pmatrix} \tag{62}$$

For scaling purposes we shall not use the Gell-Mann matrices but instead the colour matrices $T^a = \frac{1}{2}\lambda_a$. The structure factors are f^{abc} of the Lie algebra are defined by:

$$\left[T^a, T^b\right] = i f^{abc} T^c \tag{63}$$

One can show the following properties of the colour matrices and the structure factors in SU(N):

$$f^{acd} f^{bcd} = N \cdot \delta_{ab} \tag{64}$$

$$i \cdot f^{abc} = 2 \cdot \mathrm{Tr}\left[T^c \left[T^a, T^b\right]\right] \tag{65}$$

$$\mathrm{Tr}\left[T^a T^b\right] = \frac{1}{2}\delta_{ab} \tag{66}$$

$$T^a T^a = \frac{N^2 - 1}{2N}\mathbf{1} \tag{67}$$

$$T^a_{ik} T^a_{lj} = \frac{1}{2}\delta_{ij}\delta_{kl} - \frac{1}{2N}\delta_{ik}\delta lj \tag{68}$$

4.2 $q\bar{q} \to \gamma g$

Similar to the previously discussed interaction we shall now calculate the differential cross section for a quark-antiquark annihilation to a photon and a gluon (see fig.2). Since the only difference to the $e^- e^+ \to \gamma\gamma$ is that the electron charge is replaced by the quark charge and one electron-photon vertex is replaced by a quark-gluon

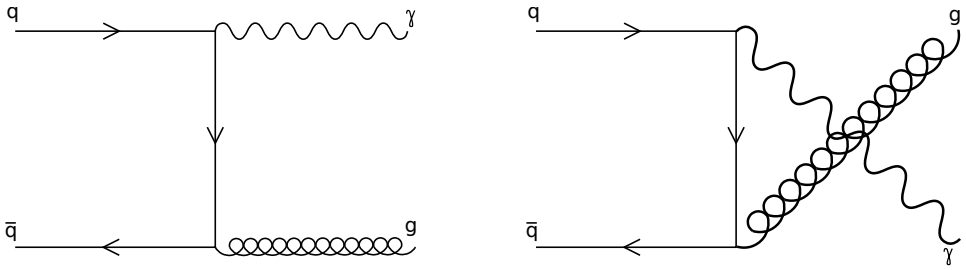


Figure 2: Feynman diagrams of electron-positron annihilation

vertex:

$$-ie\gamma^\mu \to g\gamma^\mu T^a \tag{69}$$

This means that we can calculate the transition elements $\overline{|\mathcal{M}_1|}^2$ and $\overline{|\mathcal{M}_2|}^2$ easily from previous results:

$$\overline{|\mathcal{M}_1|}^2 = \frac{1}{n_{colours}} \sum_{colours} 2e_q^2 g^2 \frac{t}{u} \cdot \mathrm{Tr}\left[T^a T^b\right] \tag{70}$$

$$= \frac{1}{n_{colours}} \sum_{colours} 2e_q^2 g^2 \frac{t}{u} \cdot \frac{1}{2}\delta_{ab} \tag{71}$$

$$= e_q^2 g^2 \frac{t}{u} \tag{72}$$

$$\Rightarrow \overline{|\mathcal{M}|}^2 = e_q^2 g^2 \left(\frac{t}{u} + \frac{u}{t}\right) \tag{73}$$

$$\Rightarrow \frac{d\sigma}{d\Omega}(q\bar{q} \to \gamma g) = \frac{1}{2}\left(\frac{e_q g}{e^2}\right)^2 \frac{d\sigma}{d\Omega}(e^+e^- \to \gamma\gamma) \tag{74}$$

4.3 The colourtrace program

When calculating cross sections of reactions in QCD we often have to calculate traces of the colour matrices T_a for $a \in \{1, 2, ..., 8\}$. Doing this by hand can be a tedious task which is why a package was written for Mathematica which can simplify the trace of a product of colour matrices, structure factors f_{abc} of the $SU(N)$ Lie algebra and Kronecker deltas δ_{ab} and δ_{ij} with colour indices a, b and matrix indices i, j. The package uses the following identities (Dr. Karol Kovarik, personal communication,

2015):

$$T_F := \frac{1}{2} \tag{75}$$

$$C_A := N \tag{76}$$

$$C_F := \frac{N^2 - 1}{2N} \tag{77}$$

$$T^a T^a = C_F \cdot \mathbf{1} \tag{78}$$

$$\mathrm{Tr}\left[T^a T^b\right] = \frac{1}{2}\delta^{ab} \tag{79}$$

$$f^{abc} f^{acd} = N\delta^{bd} \tag{80}$$

$$if^{abc} = 2\mathrm{Tr}\left[T^c \left[T^a, T^b\right]\right] \tag{81}$$

$$\delta_{ij}\delta_{kl} = \frac{1}{N}\delta_{ik}\delta_{lj} + 2T^a_{ik}T^a_{lj} \tag{82}$$

```
(*=============FILE:colourtrace.m==============*)
(*                                              *)
(*              ---colourtrace---               *)
(*            a mathematica package             *)
(*     for computation of colour matrices       *)
(*                  Author:                     *)
(*          Oliver Benjamin Smith               *)
(*                                              *)
(*==============================================*)
BeginPackage["colourtrace`"];
(* Manual entries for used functions *)
TF::usage=
"TF[a,i,j] refers to the (i,j) entry of the colourmatrix T^a with a
    in {1,2,...,N^2-1}";
f::usage=
"f[a,b,c] refers to the structure factor of the SU(N) Lie-algebra.
   I.e. it satisfies [T^a,T^b]=\[ImaginaryI]f[a,b,c]T^c";
Dd::usage=
"Dd[i,j] is the kronecker delta symbol for two matrix indices i and
    j.";
Dg::usage=
"Dg[a,b] is the kronecker delta symbol for the colour indices a and
    b.";
Calc::usage=
"Calc[expr] yields the simplified result of the expr in terms of Dd
   , Dg, f and N";
CalcNice::usage=
```

```
"CalcNice[expr] yields the simplified result of the expr in terms
   of Dd, Dg, f, CA, CF, TF";

(* Rules for simplification of expressions *)
Rules = {
(* Contraction of two structure factors to a kronecker delta symbol
    *)
f[a_, c_, d_] f[b_, c_, d_] ->  N*Dg[a, b],
f[a_, c_, d_] f[b_, d_, c_] -> -N*Dg[a, b],
f[a_, c_, d_] f[d_, b_, c_] ->  N*Dg[a, b],
f[a_, c_, d_] f[c_, b_, d_] -> -N*Dg[a, b],
f[a_, c_, d_] f[c_, d_, b_] ->  N*Dg[a, b],
f[a_, c_, d_] f[d_, c_, b_] -> -N*Dg[a, b],
f[d_, a_, c_] f[d_, b_, c_] ->  N*Dg[a, b],
f[d_, a_, c_] f[c_, b_, d_] -> -N*Dg[a, b],
f[d_, a_, c_] f[c_, d_, b_] ->  N*Dg[a, b],
f[d_, a_, c_] f[d_, c_, b_] -> -N*Dg[a, b],
f[c_, d_, a_] f[c_, d_, b_] ->  N*Dg[a, b],
f[c_, d_, a_] f[d_, c_, b_] -> -N*Dg[a, b],

(* Commutation relation 'backwards' *)
f[a_, b_, c_] TF[c_, i_, j_] ->  I* TF[a, k, j] TF[b, i, k] - I *TF
   [a, i, k] TF[b, k, j],
(* i*f[a,b,c]=2Trace[T^c Commutator[T^a,T^b]] *)
f[a_, b_, c_] ->
Module[{i, j, k}, -I*2*TF[a, i, j] TF[b, j, k] TF[c, k, i] + I*2*TF
   [b, i, j] TF[a, j, k] TF[c, k, i]],

(* SU(N)-generators are traceless *)
TF[a_, i_, i_] ->  0,
(* Trace of two colour matrices *)
TF[a_, i_, j_] TF[b_, j_, i_] ->  1/2 Dg[a, b],
(* Square of equal colour matrices (for faster computation)*)
TF[a_, i_, j_] TF[a_, j_, k_] ->  CF*Dd[i, k]),
(* If no more squares can be found *)
TF[a_, i_, k_] TF[a_, l_, j_] -> 1/2 Dd[i, j] Dd[k, l] - 1/(2 N) Dd
   [i, k] Dd[l, j],
(* Computation of kronecker symbols *)
TF[b_, i_, j_] Dg[a_, b_] ->  TF[a, i, j],
TF[a_, k_, j_] Dd[i_, k_] ->  TF[a, i, j],
TF[a_, i_, k_] Dd[j_, k_] ->  TF[a, i, j],
Dd[i_, i_] ->  N,
Dg[a_, a_] ->  CF*N*2,
Dd[i_, j_] Dd[j_, k_] ->  Dd[i, k],
```

```
Dg[a_, b_] Dg[b_, c_] ->  Dg[a, c],
Dd[i_, j_]^2 ->  N,
Dg[a_, b_]^2 ->  CF*N*2
};

NiceForm = {
(N^2 - 1) ->  2*N*CF,
N ->  CA,
1/2 ->  TF,
2*TF ->  1
};

SetAttributes[Dd, Orderless];
SetAttributes[Dg, Orderless];

Calc[expr___]:=Expand[Expand[expr//.Rules]//.Rules];
CalcNice[expr___]:=Calc[expr]//.NiceForm;
EndPackage[]
```

4.3.1 Results of the colourtrace program

We shall compare the outputs of the program to calculations by hand.

4.3.2 $(T^a)^2$, $f^{abc}f^{dbc}$, $\mathbf{Tr}\left[T^aT^b\right]$

Our program verifies the equations (78), (80) and (79):

```
In[2]:= CalcNice[TF[a,i,j]*TF[a,j,k]]

Out[2]= CF Dd[i, k]
```

```
In[3]:= CalcNice[f[a,b,c]f[d,b,c]]

Out[3]= CA Dg[a, d]
```

```
In[4]:= CalcNice[TF[a,j,i]TF[b,i,j]]

Out[4]= TF Dg[a, b]
```

4.3.3 $T^bT^aT^b$

Calculation by hand (example taken from [6]):

$$(T^bT^aT^b)_{ij} = \left(T^b_{ik}T^b_{lj}\right)T^a_{kl} \overset{(82)}{=} \frac{1}{2}\left(\delta_{ij}\delta_{kl} - \frac{1}{N}\delta_{ik}\delta_{lj}\right)T^a_{kl} \tag{83}$$

$$= \frac{1}{2}\delta_{ij}T^a_{kk} - \frac{1}{2N}T^a_{ij} \overset{\mathrm{Tr}[T^a]=0}{=} -\frac{T^a_{ij}}{2C_A} \tag{84}$$

Program output:

```
In[5]:= CalcNice[TF[b,k,i]TF[a,l,k]TF[b,j,l]]

          -TF[a, j, i]
Out[5]= ------------
             2 CA
```

4.3.4 $\mathbf{Tr}\left[T^aT^bT^bT^a\right]$

Calculation by hand (example taken from [6]):

$$\mathrm{Tr}\left[T^aT^bT^bT^a\right] = T^a_{ij}T^b_{jk}T^b_{kl}T^a_{li} \overset{(82)}{=} \frac{1}{4}\left(\delta_{ii}\delta_{jl} - \frac{1}{N}\delta_{ij}\delta_{li}\right)\left(\delta_{jl}\delta_{kk} - \frac{1}{N}\delta_{jk}\delta_{kl}\right) \tag{85}$$

$$= \frac{1}{4}\left(\delta_{ii}\delta_{jl}\delta_{kk} - \frac{1}{N}\delta_{ii}\delta_{jl}\delta_{jk}\delta_{kl} - \frac{1}{N}\delta_{ij}\delta_{li}\delta_{jl}\delta_{kk} + \frac{1}{N^2}\delta_{ij}\delta_{li}\delta_{jk}\delta_{kl}\right) \tag{86}$$

$$= \frac{1}{4}\left(N^3 - \frac{N^2}{N} - \frac{N^2}{N} + \frac{N}{N^2}\right) = N\frac{N^4 - 2N^2 + 1}{4N^2} = C_AC_F^2 \tag{87}$$

Program output:

```
In[6]:= CalcNice[TF[a,j,i]TF[b,i,l]TF[b,l,k]TF[a,k,j]]

                  2
Out[6]= CA CF
```

4.3.5 $\mathbf{Tr}\left[T^aT^b\right]\mathbf{Tr}\left[T^aT^b\right]$

Calculation by hand (see [6]):

$$\mathrm{Tr}\left[T^aT^b\right]\mathrm{Tr}\left[T^aT^b\right] \overset{(79)}{=} \frac{1}{2}\delta^{ab}\frac{1}{2}\delta^{ab} = \frac{1}{4}\left(N^2-1\right) = \frac{1}{2}N\frac{N^2-1}{2N} = \frac{1}{2}C_AC_F \tag{88}$$

Program output:

```
In[7]:= CalcNice[TF[a,k,i]TF[b,i,k]TF[a,l,j]TF[b,j,l]]

Out[7]= CA  CF  TF
```

4.3.6 Tr $\left[f^{abc}f^{abd}T^cT^d\right]$

Calculation by hand (see [6]):

$$\mathrm{Tr}\left[f^{abc}f^{abd}T^cT^d\right] \overset{(80)}{=} N\delta^{cd}\cdot\frac{1}{2}\delta^{cd} = N\cdot\frac{N^2-1}{2} = N^2\cdot C_F = C_A^2 C_F \tag{89}$$

Program output:

```
In[8]:= CalcNice[f[a,b,c]f[a,b,d]TF[c,j,i]TF[d,i,j]]

              2
Out[8]= CA   CF
```

4.3.7 Tr $[T^aT^a]$

Calculation by hand (see [6]):

$$\mathrm{Tr}\left[T^aT^a\right] \overset{(79)}{=} \frac{1}{2}\delta^{aa} = N\frac{N^2-1}{2N} = C_A C_F \tag{90}$$

Program output:

```
In[9]:= CalcNice[TF[a,j,i]TF[a,i,j]]

Out[9]= CA  CF
```

5 Monte Carlo method

Monte Carlo methods are probabilistic approaches to numerical problems in which the aim is to find a probabilistic interpretation of the problem at hand. A classic example of a Monte Carlo method is the calculation of the area of the unit disk in the first quadrant which should of course be $\frac{\pi}{4}$. To do this we construct a random variable

for which the expected value is the integral of $f(x) = \sqrt{1-x^2}$ in the unit interval. We can achieve this by simply picking the random variable $X : [0,1] \to \mathbb{R}, x \mapsto f(x)$ with the probability density $p : [0,1] \to \mathbb{R} \equiv 1$. That way we get the expectation of X:

$$\mathbb{E}X = \int_{[0,1]} X(x) \cdot p(x)dx = \int_{[0,1]} f(x)dx = \frac{\pi}{4} \tag{91}$$

Since the maximum likelihood estimator for the expectation is the arithmetic mean and is unbiased, we can estimate $\frac{\pi}{4}$ by picking random samples according to the probability distribution and averaging over the outcome. This method can be used for almost arbitrary integrals (the function must of course obey certain restrictions to guarantee convergence). To integrate a function $f : \Omega \to \mathbb{R}$ we can simply choose a positive probability density $p : \Omega \to \mathbb{R}^+$ and then reformulate the problem as follows:

$$\int_\Omega f(x)dx = \int_\Omega \frac{f(x)}{p(x)} \cdot p(x)dx =: \int_\Omega X(x) \cdot p(x)dx \tag{92}$$

When computing we are not only interested in knowing that we will eventually get to the desired result since the computing time is also relevant. This means, that we must estimate how quickly we will in fact get close to our result. To quantify this we shall take a look at the variance of our estimation $\bar{X} = \frac{1}{N}\sum X_i$ after N samples.

$$\mathbb{V}\bar{X} = \frac{1}{N^2}\sum \mathbb{V}X_i = \frac{1}{N}\mathbb{V}X \tag{93}$$

With this we gain the relation of the standard deviation $\sigma = \frac{1}{\sqrt{N}}\sigma(X)$. Here we see that we can minimize the variance of our estimator by minimizing the variance of X. In other words, we want to ensure that $X$is an almost constant function. This can be done by choosing a probability density $p \approx \frac{f}{c}$ for a constant c (see [2 app. A]). To choose p to be precisely proportional to f we would have to calculate the normalization constant c though, which would force us to calculate the integral over f and therefore brings us to our initial problem. If we have prior knowledge of f we can try to select a probability density function predominantly weighted at great values of f. If this is not the case we must resort to algorithms which try to dynamically optimize p in multiple iteration steps. This process is called *importance sampling* and one method will be discussed when introducing the VEGAS algorithm developed by G.P.Lepage (see[5]).

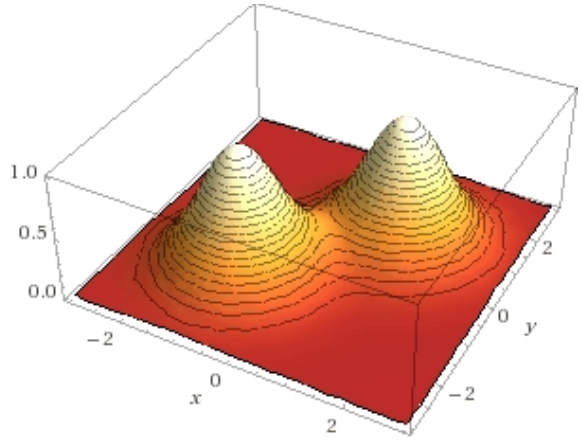


Figure 3: Plot[1] of the function $f(x) = e^{-(x-1)^2-(y-1)^2} + e^{-(x+1)^2-(y+1)^2}$

5.1 VEGAS

The VEGAS algorithm is a Monte Carlo method for calculating high-dimensional integrals implementing importance sampling for variance reduction. The algorithm can integrate a function over a hypercube. The initialization is done by dividing each edge of the N-dimensional hypercube in k equidistant segments thus dividing the hypercube into k^N smaller cubes. Each of these cube is assigned the probability $\frac{1}{k^N}$ for normalization. The first part of each iteration consists of selecting cubes at random and subsequently choosing a point in each selected cube by a uniform probability distribution to evaluate the function. The evaluated points are used to estimate the integral as well as creating histograms along each axis of the integration domain. The second step in each iteration is the improvement of the current grid of cubes. This is done by decreasing (increasing) the size of segments along each axis in which the average evaluated function is high (low). By following these two steps repeatedly the effective probability distribution will get close to being proportional to the integrand, hence the variance of our estimates will diminish. This method assumes the function to be separable with respect to the coordinates though. This for example is not the case for functions such as the sum of two Gaussians with different peaks as in figure 3.

[1]Plotted with wolframalpha.com

5.2 CUDA

nVidia has created a library of functions for C++ enabling it to use the resources of certain dedicated nVidia graphic processing units (GPUs). This enables a program to compute independent tasks on single cores of the GPU simultaneously. Converting a CPU driven program to a GPU driven program requires several steps. First we must introduce a few concepts of the computer architecture (see [4]). The **host** is the compound consisting of central processing unit (CPU) and it's random access memory (RAM) whereas the **device** is the GPU and it's seperate RAM. To declare a routine executed on the GPU one must simply declare it with keyword **__global__**:

```
__global__ void myTestFunction (void) {}
```

The keyword tells the compiler that the function shall be run on the device and can be called from the host. When a function is executed on the device it can be called with two additional arguments, the number of **blocks** as well as the number of **threads** in each block the process shall be called with. We shall discuss these in more detail later. This is the form of a call of a device function from a host function:

```
void main (void){
...
myTestFunction <<blk,thrd>>();
...
}
```

Here *blk* and *thrd* are the number of blocks and threads respectively. Since the CPU and GPU each can only access their own memory one must allocate memory on the device for each variable needed in GPU calculations first and then copy values from the host memory to the device memory. CUDA supplies functions to do exactly that which are similar to the C-native functions malloc(), free() and memcpy():

```
int main(void) {
        int hostVar;
        int *devPointer;
        int size=sizeof(int);
        cudaMalloc((void **)&devPointer,size);
        cudaMemcpy(devPointer,&hostVar,size,cudaMemcpyHostToDevice)
           ;
        someDeviceFunction <<numBlk,numThrd>>(devPointer);
        cudaMemcpy(&hostVar,devPointer,size,cudaMemcpyDeviceToHost)
           ;
```

```
        cudaFree(devPointer);
}
```

cudaMalloc will allocate memory on the device of length *size* and will assign the location to the pointer *devPointer*. The function *cudaMemcpy* will then copy the data of length *size* at the address of hostVar to *devPointer*. Now the device pointer *devPointer* can be given to a device function. After this the new value at *devPointer* is copied back to the address of *hostVar* with the function *cudaMemcpy* an the argument *cudaMemcpyDeviceToHost*. Now that we can run a function on the GPU we shall discuss the use of blocks and threads which are the main concepts that make GPU programming so efficient.

The number of blocks defines the number of cores that shall be used to when invoking a function. This means that when a function is called with N blocks it is actually invoked N-times on N seperate blocks. To be able to make the function compute different things on each block we can access a variable called *blockIdx.x* to access the index of the invocation.

Threads are parallel invocations of a function inside of one block. That means if a function is called with N blocks and K threads in each block there will be K invocations of the function resulting in $N \times K$ invocations in total. Just like in the case of blocks we can access a variable called *threadIdx.x* giving us the thread index of the invocation. The advantage of threads is that, unlike different blocks, different threads of the same block can access their shared variables (which can be declared with the *___shared___* keyword). In addition all the threads in one block can be synchronized with the command *___syncthreads()* giving them the ability to wait for other threads completing the computation of interim results. When handling output variables of a device function it is also important to use the built in atomic functions (e.g. atomicAdd(), atomicSub(), atomicInc(), atomicDec()) which ensure that the memory is not accessed by different threads simultaneously which can lead to a multitude of errors.
The following section will show an example of a CUDA program using the GPU processing capabilities to compute the scalar product of two vectors.

5.3 Example of a CUDA program

```
#define DIM 3000
#define THREADS 500
```

```
#include <iostream>
using namespace std;

void random_ints (int *a, int n);
__global__ void scalarprod(int *a, int *b, int *c);

int main(void) {
        int *a, *b, *c, *d_a, *d_b, *d_c;
        int size=DIM*sizeof(int);

        //allocate memory on device for variables d_a, d_b and d_c
        cudaMalloc( (void**)&d_a,size);
        cudaMalloc( (void**)&d_b,size);
        cudaMalloc( (void**)&d_c,sizeof( int ));

        //allocate memory on host for variables d, b and c
        a=(int *)malloc( size );
        b=(int *)malloc( size );
        c=(int *)malloc( sizeof( int ) );

        //initialize a and b with random data
        random_ints ( a,DIM );
        random_ints ( b,DIM );
        *c=0;

        //copy variables to device memory
        cudaMemcpy(d_a,a,size,cudaMemcpyHostToDevice);
        cudaMemcpy(d_b,b,size,cudaMemcpyHostToDevice);
        cudaMemcpy(d_c,c,sizeof( int ), cudaMemcpyHostToDevice);

        //execute scalarprod on device with DIM/THREADS blocks
        //and THREADS threads
        scalarprod <<<DIM/THREADS,THREADS>>> (d_a,d_b,d_c);


        //copies calculated value of d_c to host memory
        cudaMemcpy(c,d_c,sizeof( int ), cudaMemcpyDeviceToHost );

        //output
        cout << endl << "c="  << c[0] << endl;
        cout << "Time in ms:" << clock()-start<<endl;

        //freeing of device and host memory
        free (a);
```

```
        free (b);
        free (c);
        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_c);
        return 0;
}

void random_ints (int *a, int n)
{
        for (int i=0; i<n; ++i)
        a[i]=rand();
}

__global__ void scalarprod(int *a, int *b, int *c)
{
        //temps is created as an array shared between threads of
            same block
        __shared__ int temp[THREADS];

        //by block and thread indexing no component of *a or *b is
            accessed twice
        int index=blockIdx.x*THREADS+threadIdx.x;

        //calculation of scalar product
        temp[threadIdx.x]=a[index]*b[index];

        //wait till entire block is at this step
        __syncthreads();

        //only the first thread calculates the sum of the block
        if (threadIdx.x==0)
        {
                int sum=0;
                for (int i=0;i<THREADS;i++)
                {
                        sum+=temp[i];
                };
                //atomicAdd makes sure that c is not accessed
                    simultaneously by different blocks
                atomicAdd(c,sum);
        };
};
```

5.4 gVEGAS

gVEGAS is program written by Junichi Kanzaki which uses the VEGAS algorithm and utilizes the CUDA libraries to improve performance by parallel computing on GPUs (see [7]).

5.4.1 How to use gVEGAS package

Using the gVEGAS package consists of the following steps.

1. In "gVegasMain.cu" (line 103) we can define the number of dimensions to integrate over in the variable *ndim* as well as the upper and lower bounds of the specific dimensions (line 109) in the arrays *xl* and *xu*.

2. We can specify the integrand in the source file "gVegasFunc.cu" in the function *func*:

```
REAL func(REAL* rx, REAL wgt)
{
        REAL value = 0;
        value= f(rx[0],rx[1],...,rx[DIM]);
        return value;
};
```

   by replacing *f(rx[0],rx[1],...,rx[DIM])* with the desired integrand.

3. By defining the constant *REAL* in "real.h" as either *float* or *double* we can decide whether the calculations should be computed with double or single precision. This can influence the computation time by up to a factor of 3.

4. We can then compile the code to the executable "gVegasDouble" by executing the "build.sh" script. The compiled code can be called either by running the "run_vegas.sh" script or by calling the "gVegasDouble" program with the following arguments:

```
./gVegasDouble --ncall=NCALL --itmx=ITMX --acc=ACC --blk=BLK --
   dev=DEV
```

   Here *NCALL* is the number of samples created and evaluated per iteration in scientific format (e.g.: *NCALL="e05n2"*$\equiv NCALL = 2 \cdot 10^5$), *ITMX* specifies the maximum number of iterations to compute, *ACC* sets the accuracy threshold at which the computation will stop, *BLK* defines the number of

blocks that the computations will use on the *GPU* and *DEV* is the device number of the *GPU*.

5.4.2 Comparison of computation times

For testing both the VEGAS and the gVEGAS algorithm we used the following three dimensional integral:

$$\int_{x,y,z\in[0,1]} e^{-x^2-y^2-z^2}dxdydz = \left(\int_0^1 e^{-x^2}dx\right)^3 = \left(\sqrt{\pi}\text{erf}(1)\right)^3 \approx 0.416538385... \quad (94)$$

The CPU algorithm was set to calculate a total of 10^8 evaluations of the integrand on a computer with the specifications as seen in table 4 while the GPU algorithm was set to calculate either 10^{10} or 10^9 samples with different numbers of blocks on a GeForce GTX Titan Black (see tab.5). The times were taken from the "user" output of the Linux "time" command and averaged over ten runs:

Table 4: Specifications of processor for CPU execution

Brand	Intel(R) Core(TM)
Model	i5-4210U
Clock	1.7 GHz
L1 cache	32 KiB
L2 cache	256 KiB
L3 cache	3 MiB
RAM	8 GiB
RAM type	SODIMM DDR3@1600 MHz

Table 5: Specifications of graphics card for GPU execution

Brand	nVidia
Model	Geforce GTX Titan Black
Base clock	889 MHz
CUDA cores	2880
Memory	6144 MiB
Memory type	GDDR5@3500MHz

Table 6: Computation times on GPU

Number of samples	Num. of blocks	Time
10^{10}	64	$(112.3 \pm 1.1)s$
10^{10}	128	$(110.8 \pm 1.3)s$
10^{10}	512	$(110.2 \pm 1.1)s$
10^{9}	64	$(12.22 \pm 0.10)s$
10^{9}	128	$(12.06 \pm 0.11)s$
10^{9}	512	$(11.98 \pm 0.09)s$

This shows a trend of the algorithm being faster for a higher number of blocks though the results are somewhat inconclusive due to the standard deviation.

$$t_{CPU} = (8.12 \pm 0.14)s \tag{95}$$

$$t_{GPU} = (1.16 \pm 0.07) \tag{96}$$

This shows that the gVEGAS program is faster by a factor of (7 ± 0.4) when computing three dimensional algorithms.

6 Conclusion

When calculating cross sections for particle interactions we often come to a point where tedious calculations of traces or high dimensional integrals are unavoidable. It is therefore sensible to use computational techniques to solve these problems. The colourtrace program has shown that the problem of traces in QCD can be solved elegantly by implementing replacement rules in Mathematica. For the case of high dimensional integration we have shown that utilization of GPUs in the VEGAS algorithm can significantly improve performance compared to the conventional computation on the CPU even if the predicted factor of more than 10 between CPU and GPU times [7] could not be replicated the improvement is still significant and further implementation is worth investigating.

7 Acknowledgements

I would like to thank Professor Klasen and his work group for their kind support with my bachelor thesis. I would especially like to thank Florian König and Thomas Biekötter for always being willing to explain and clarify when I had any difficulties, Karol Kovarik for taking the time to tutor me on the subject matter of colour matrices and Thomas Zub for his great efforts in rewriting the gVEGAS code to run on our system.
Lastly I would like to thank my parents and brothers whose ideas, corrections and inquiries kept me on track.

8 References

1. Ohl, T. (2012). Feynman Diagrams For Pedestrians. Retrieved from physik.uni-wuerzburg.de

2. Jarosz, W. (2008). Efficient Monte Carlo Methods for Light Transport in Scattering Media. Retrieved from cs.dartmouth.edu

3. Halzen, F.; Martin, A. D. Quarks and leptons: an introductory course in modern particle physics. Wiley, 1984.

4. Zeller, C. (2011). CUDA C/C++ Basics. Retrieved from nvidia.com

5. Lepage, G. P. A New Algorithm for Adaptive Multidimensional Integration, Journal of Computational Physics 27, 192-203. 1978.

6. Derya, V. (2008). Color factors in QCD. Retrieved from lpsc.in2p3.fr

7. Kanzaki, J.; Hagiwara, K.; Li, Q.; Okamura, N.; Stelzer, T. Fast computation of MadGraph amplitudes on graphics processing unit. The European Physical Journal C. (05.11.2013)