DEPARTMENT OF PHYSICS AND ASTRONOMY
UNIVERSITY OF HEIDELBERG

Bachelor Thesis in Physics
submitted by

Christoph Dressler
born in Heidelberg

2011

Implementation of a Phase Space Generator in C++

This Bachelor thesis has been carried out by Christoph Dressler at

Physikalisches Institut in Heidelberg

under supervision of

Prof. Dr. André Schöning

Abstracts

Inspired by the search for physics beyond the Standard Model an experiment has been proposed to search for the lepton flavour violating decay $\mu \rightarrow eee$. The experiment has a target sensitivity four orders of magnitude beyond previous experiments. This makes it crucial to understand the background of this process. In this thesis a phase space generator for the simulation of the experiment has been implemented in C++. The viability of the code has been shown by describing the kinematics of the simulated events in Dalitz Plots. The results have been compared to experimental data.

Inspiriert von der Suche nach Physik jenseits des Standardmodells wurde ein Experiment entworfen, dass den Zerfall $\mu \rightarrow eee$ untersuchen wird. Dieser Zerfall verletzt die vom Standardmodell vorhergesagte Erhaltung der Leptonfamilienzahl. Die angestrebte Sensitivitaet dieses Experiments wird um bis zu vier Groessenordnungen hoeher als bei bisher durchgefuehrten Experimenten sein. Daher ist es von grosser Bedeutung, den Untergrund des Prozesses zu verstehen. In dieser Arbeit wurde ein Phasenraumgenerator fuer das Simulationssystem in C++ programmiert. Die Funktionsfaehigkeit des Programmes konnte anhand der kinematischen Darstellung der generierten Ereignisse in Dalitz Plots nachgewiesen werden.

Contents

1 Particle Physics

This thesis has been done in the field of *particle physics* which deals with the constituents of matter at subatomic size as well as their interactions. Under ambient conditions on earth, most particles are not stable and decay rapidly. To study them scientifically, they have to be specifically created at very high energies that can only be achieved in dedicated particle accelerators. For this reason, particle physics is also called *high energy physics*. This work contributes to the design of an innovative particle detector that will be built to search for the decay $\mu \rightarrow eee$. Details about the particular decay will follow in a later chapter.

In the following section, a brief introduction to the physical background and the theory this thesis is based on is given.

1.1 The Standard Model

Both the known subatomic particles and the forces that mediate their dynamics are currently best described by the Standard Model of particle physics, a theory that combines the electromagnetic, weak and strong nuclear interactions. Its origins go back to 1960, when Sheldon Glashow found a way to combine electromagnetic and weak interactions [1] . In 1967, Steven Weinberg [2] and Abdus Salam [3] incorporated the Higgs mechanism and gave the theory its modern form.

The Standard Model explains a wide variety of physical phenomena and experimental results and has thus gained a lot of credence in the last decades. Until today, none of the main predictions of the the Standard Model have been proven wrong, but not all of them have been confirmed experimentally either.

It is built upon three main pillars: particles, forces and the Higgs mechanism. The Higgs mechanism is widely believed to give the particles their masses, but this has not been confirmed to date.

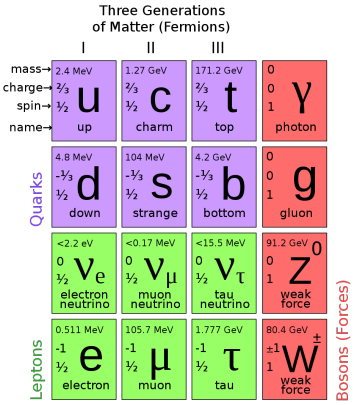

Figure 1: The Standard Model

Particles

The elementary particles of the Standard Model are categorized in two groups: fermions (spin 1/2) and bosons (spin 1), where *spin* is a quantum-mechanical property of the particles that can be regarded as their intrinsic angular momentum. Fermions are usually associated with ordinary matter, whereas bosons are the force carriers that mediate the strong, weak and electromagnetic interactions. The two basic categories can be subdivided as follows:

Fermions are particles that obey *Fermi-Dirac statistics*, stating that only one particle can occupy a particular quantum state at any given time. They are classified according to their *charge*, and other quantum numbers (isospin, colour,...). There are six *quarks* (up, down, charm, strange, top, bottom) and six *leptons* (electron, muon, tau, each with a corresponding neutrino). They can be classified in three families according to their electric charge. In addition, they are arranged in three generations, and there is a mass hierarchy between these generations.

Quarks carry fractional electric charge and leptons integer electric charge. Additionally, they are distinguished by the fact that that quarks carry *color charge*. Color charge is a property that enables particles to interact via the strong interaction (see below). A phenomenon called *color confinement* results in quarks not being able to exist as free particles, but being forced to form color-neutral composite particles (*hadrons*) that contain either a quark and an antiquark (*mesons*) or three quarks

(*baryons*).

Leptons do not carry color charge. The neutrinos do not carry electric charge, either. Only electron-like leptons do and thus interact electromagnetically.

Forces

As mentioned above, all these particles experience different forces. In the Standard Model, those forces are explained to be caused by the exchange of force mediating particles, so-called bosons. As opposed to fermions, all bosons have integer spin and obey *Bose-Einstein statistics*, which means that, unlike fermions, two or more bosons can occupy the same quantum state at the same time. Bosons can be both elementary particles and composite ones, since obviously any particle containing an even number of fermions will behave according to Bose-Einstein statistics, due to its integer spin. There are several types of elementary bosons:

- Photon - the force carrier for the electromagnetic force
- $W^{\pm}$ and Z bosons - the force carriers for the weak force
- Gluons - the force carriers for the strong force

Those three kinds of bosons are called *gauge bosons* since each one of the corresponds to one of the three Standard Model interactions. Two more gauge bosons have been predicted, but not observed experimentally yet:

- Higgs boson - a massive scalar particle explaining why the other elementary particles (except photon and gluon) are massive
- Graviton - the force carrier of gravitation

The reason why we have this very detailed knowledge about the constituents of matter in our universe is mainly due to the construction of better and better particle accelerators and particle detectors. Since many of the elementary particles are extremely massive, the required energy to create those particles is correspondingly

large according to Einsteins $E = mc^2$. After the electron has been discovered around 1900, most of the following discoveries were results of experiments with cosmic rays due to the fact that was were simply no other source of high energy particles available at the time. It was not until 1930 that physicists started to build dedicated particle accelerators that provided controlled beams of particles of known energy. This beam energy has been increased again and again over the last decades up to magnitudes of several TeV (*Large Hadron Collider*, CERN), allowing to create and observe evermore particles of higher and higher mass.

1.2 Beyond the Standard Model

All kinds of experiments manifest the power of the Standard Model. Nevertheless, there are several inherent deficiencies, such as an explanation for gravity, the observed matter-antimatter asymmetry, the origins of dark matter/ dark energy and several other important phenomena. Resulting from these shortcomings of the Standard Model, the search for a more complete 'theory of everything' has become one of the most active areas of research in physics.

Theoretical physicists have been developing models that provide solutions to those problems. The most promising models are the various forms of *Supersymmetry*, a theory that postulates a fermion-boson symmetry – all known fundamental bosons (fermions) would then have 'new' fermion (boson) partners. Since in case of perfect symmetry, all these partners would have equal masses, this symmetry could not be exact. A different ansatz are *technicolor* theories – they address the breaking of electroweak symmetry. Even though the ideas differ, they all imply the existence of new particles. If the theories are found to be true, they will induce a couple of experimental implications that can be verified. For example, the Standard Model introduces the conservation of the *lepton number* (the number of leptons minus the number of antileptons) in each leptonic family. These new particles might cause the violation of the lepton flavour conservation which could not be explained by the Standard Model.

Indeed, physics beyond the Standard Model can only be found once the rules of the Standard Model are broken, so the general way to find out more about these violations is to search for phenomena that can not be explained or are in fact not allowed by the Standard Model, e.g. certain particle decays that are kinematically possible but not in accordance to the known rules. The foundation of this thesis is the planned search for one of those decays, namely $\mu^- \rightarrow e^- e^+ e^-$. The observation of this particular decay would give powerful evidence for possible physics beyond the Standard Model. This is due to the violation of *lepton number conservation* that is supposed to stay the same through an interaction.

1.3 Particle Interactions

To describe the interactions that happen in a particle detector, one needs to understand some concepts of particle physics. Classically, physical interactions are described in terms of a potential or a field that was caused by one of the particles and had an impact on another particle. The quantum concept of interaction is rather based on *exchange* than on influence, stating that the exchanged boson carries momentum from one particle to the other with the rate of exchange providing the force. Since the whole process needs to satisfy energy conservation, it has to take place within a timescale Δt due to the Uncertainty Principle. Those transient particles that exist only for a limited space and time are said to be *virtual* and can not be directly observed.

The likelihood for a given particle interaction is specified by the so-called *cross-section*, which is a hypothetical target area in a beam of particles. There will be an interaction if a particle of the beam hits that surface. The approach originates from the classical picture, where the probability that a single point-like particle hits a solid surface was actually given by the ratio of the section of the solid to the total targeted area. If one imagines an experiment

$$a + b \rightarrow c + d \tag{1.1}$$

in which a well-defined parallel beam of particles of type a and density n_a hits a target that contains n_b particles of type b per unit volume and has the thickness dx. The flux through the target will be

$$\Phi = n_a v_i \tag{1.2}$$

with v_i being the velocity of the beam relative to the target. Since each target particle has a cross-section σ, the fraction of the target area covered by the target particles b equals the probability $\Phi\sigma n_b dx$ that any particle a will hit a target particle. Obviously, the reaction rate per target particle is therefore

$$W = \Phi\sigma. \tag{1.3}$$

In general, the reaction rate describes the probability of reaction or transition per unit time. It can be calculated using *Fermi's Golden rule*

$$W = \frac{2\pi}{\hbar}|M_{if}|^2\rho_f. \tag{1.4}$$

and depends upon the transition probability $|M_{if}|^2$ between the initial and final state of the system and upon the number of possibilities for the transition considered, which is expressed as the phase space density of the final states ρ_f.

The phase space of a specific system contains all states the system can represent, giving a rather abstract but very practical depiction of allowed points that obey energy and momentum conservation. A mathematical expression for the phase space can for instance be derived by imagining a single particle within a cube of sides L and (quantized) momentum.

$$p_x = \frac{2\pi n_x}{L} \qquad p_y = \frac{2\pi n_y}{L} \qquad p_z = \frac{2\pi n_z}{L} \tag{1.5}$$

Once transferred to *momentum space*, each state stays within the elemental volume $(2\pi)^3/V$. The number of states available to that particle becomes

$$dN_1 = \frac{\text{total phase space}}{\text{volume element}} \tag{1.6}$$

$$= \frac{1}{\frac{(2\pi)^3}{V}}\frac{1}{V}\int dp_x dp_y dp_z \tag{1.7}$$

$$= \frac{1}{(2\pi)^3}\int d^3p \tag{1.8}$$

once the momentum space is normalised to one particle per (spatial) element volume. For n particles, this leads to

$$dN_n = \frac{1}{(2\pi)^{3n}} \int \prod_{i=1}^{n} d^3p \tag{1.9}$$

and thus the number of states per unit energy (phase space) becomes

$$\rho(E_f) = \frac{dN_n}{dE} = \frac{1}{(2\pi)^{3n}} \frac{d}{dE} \int \prod_{i=1}^{n-1} d^3p_i. \tag{1.10}$$

The product goes no further than $n-1$ because the n^{th} particle is constrained by total momentum conservation.

2 Muon decay experiments

The minimal Standard Model postulates the separate conservation of electron, muon and tau number and does not allow lepton flavour changing processes. The following table shows the decays that satisfy this condition. The main muon decay is the decay to an electron, an anti-electron-neutrino and a muon-neutrino, antimuons would decay to the corresponding antiparticles. For every decay, the *branching ratio* (the fraction of particles which decay by an individual decay mode with respect to the total number of particles which decay) is given as well.

Decay	Branching ratio	Reference
$\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu$	$\approx 100\%$	[4]
$\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu \gamma$	$(1.4 \pm 0.4) \times 10^{-2}$	[5]
$\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu e^- e^+$	$(3.4 \pm 0.4) \times 10^{-5}$	[6]

Table 1: Muon decay modes permitted in the Standard Model

Certain other decay modes are kinematically allowed, but forbidden by the minimal Standard Model. These Lepton Family number violating modes are given in the following table:

Decay	Branching ratio	Confidence level	Reference
$\mu^- \rightarrow e^- \nu_e \bar{\nu}_\mu$	$< 1.2 \times 10^{-2}$	90%	[7]
$\mu^- \rightarrow e^- \gamma$	$< 1.2 \times 10^{-11}$	90%	[8]
$\mu^- \rightarrow e^- e^+ e^-$	$< 1.0 \times 10^{-12}$	90%	[9]
$\mu^- \rightarrow e^- \gamma\gamma$	$< 7.2 \times 10^{-11}$	90%	[10]

Table 2: Upper Limits for the branching ratio for forbidden muon decay modes

These processes have not been observed (yet), so one can only specify an upper limit for the corresponding branching ratios. Their observation would give clear evidence for physics beyond the Standard Model.

Several theoretical extensions break this conservation law. All models considered provide different branching ratios and predict (non-zero) observable branching ratios for lepton family number violating decays. Improving the upper limit of the $\mu^- \to e^-e^+e^-$ decay could lead to a better understanding of the properties of new particles, possibly even refute some potential beyond the Standard Model theories and can thus be seen as a major improvement in particle physics.

2.1 The SINDRUM experiment

The search for the decay $\mu^+ \to e^+e^+e^-$ with the SINDRUM magnetic spectrometer started in 1983. The result was published in [9] and gave a new upper limit for the branching ratio of the decay $\mu \to 3e$ of

$$B_{\mu\to 3e} < 1.0 \times 10^{-12} \,(90\% \,\text{Confidence Level}) \tag{2.1}$$

The branching ratio of this experiment is especially important since many Standard Model extensions predict an observable for the branching ratio. By pushing this limit, one might be able to learn more about the properties of a Beyond the Standard Model (BSM) theory or exclude certain BSM models.

2.2 Proposed experiment

A new experiment [11] is proposed. The aim is to reach a sensitivity of

$$B_{\mu\to 3e} < 1.0 \times 10^{-16} \tag{2.2}$$

corresponding to an improvement by a factor 10000 compared to the SINDRUM experiment. To achieve this sensitivity a particle rate of around 10^9 is required. The concept is similar to the SINDRUM experiment: a continuous beam of surface anti-muons μ^+ of momentum $p = 28MeV/c$ is stopped with a hollow double-cone target that consists of thin, low Z (low number of protons) material. A low Z in the cones reduces multiple scattering and thereby improves the resolution, since it is especially sensitive to multiple scattering for low momentum particles. A strong magnetic field bends the tracks and thus enables momentum measurement due to

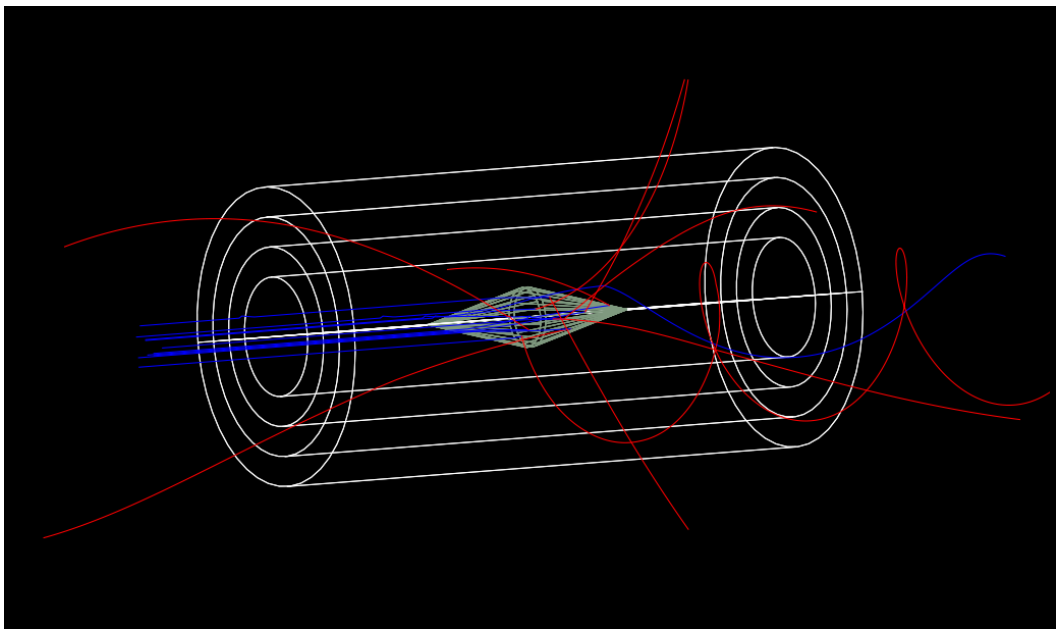

Figure 2: Simulation of muon decays using a simple detector geometry with four concentric layers of silicon sensors. The blue trajectories represent muons which are stopped in the hollow double cone target. The red trajectories are decay electrons which are bend in a solenoidal magnetic field.

the Lorentz force

$$\vec{F} = q(\vec{v} \times \vec{B}) \tag{2.3}$$

The detector is designed to reconstruct about 10^9 muon stops per second (see below). To resolve the vertices of all the different decays, the surface of the target should be large compared to the vertex resolution. The double cone structure both yields a large target surface and small opening angles, which minimizes the target material in the transverse projection of decaying electrons while at the same time providing optimum material in the beam direction so that the muon stopping efficiency is high.

The tracking system of the proposed experiment uses semiconducting material which could be as thin as $50\mu m$. This reduces multiple scattering of the electrons which is the dominant effect on the momentum measurement and improves the total energy resolution.

Detector Geometry

As mentioned above, the tracking detector is dominated by multiple scattering effects. Since the track resolution degrades with the square root of the thickness of the material [12], the number of layers should not be larger than three or four, and for redundancy reasons, four layers has been considered optimal for the proposed experiment. This choice is supported by the expected momentum range of $15 - 50\,MeV/c$. The four-layer design makes it possible to measure higher momentum tracks (with large bending radius) using four layers and lower momentum tracks using only three layers.

Monte Carlo Simulations

Around 10^9 muon events will be reconstructed in the detector per second, so there will definitely be some background. It is important to distinguish the actual events from the ones that were misidentified. To learn more about the background of the new detector, we want to simulate possible background processes with Monte Carlo simulation and run the reconstruction scheme on the generated events. With these studies we can benchmark efficiency and fake rate.
The background can have different reasons [11]. It can be real physics backgrounds (in this case only the radiative muon decay with internal conversion $\mu \rightarrow eee\nu\nu$) or accidental backgrounds due to combinational phenomena such as

- accidental coincidences of two muon decays (yielding two positrons) and the existence of a fake electron track due to wrong reconstruction, not identified back curling or Bhabha Scattering [17]
- accidental coincidences of an ordinary muon decay and an radioactive muon decay with internal conversion where one electron is not detected

Lepton flavour violating muon decays are not observed until now. If observed they are a clear signature of Physics beyond the Standard Model. The proposed experiment has an improved sensitivity by four orders of magnitude. This also means that the background of the process $\mu \rightarrow eee$ and detector effects should be well understood.

3 Introduction to Monte Carlo methods

Event generation involves solving multidimensional integrals. For the work of this thesis, the following integral had to be solved (see Sec.4).

$$V(\sqrt{P^2}) = \int \delta^4 \left(P - \sum_i^N p_i \right) \prod_i^N [d^4 p_i \, \delta(p_i^2) \theta(p_i^0)] \tag{3.1}$$

This integral is high-dimensional and can not be solved efficiently with ordinary methods, but there are other possibilities to address this problem. The following explanations of various integration techniques are based on [13] and illustrate how classic numerical integration methods approach a simple, one-dimensional integration problem. It will become obvious why their adaption for higher dimensional integrals causes problems that can be addressed using the *Monte Carlo* approach, a numerical integration method that is capable of (and thus helpful for) calculating higher dimensional integrals that can not be solved analytically.
Let us have a look at the one-dimensional integral

$$I = \int dx \, f(x) \tag{3.2}$$

for which an estimation is wanted.

3.1 Classical numerical integration

The simplest example for one-dimensional integrals are classic numerical integration methods. They refer to they idea of the Riemann integral [19] and solve the integral by calculating the area under the curve. Those methods can be broadly categorized by the question whether they evaluate the integrand at equally spaced (Newton-Cotes type) or non-equally spaced (Gaussian quadratures) abscissas.

Newton-Cotes type formulae

Newton-Cote type formulae are used once it makes sense to approximate an integral over a finite interval by weighted values of the integrand at equally spaced abscissas. The simplest example is the trapezoidal rule:

$$\int_{x_0}^{x_0+\Delta x} = \frac{\Delta x}{2} [f(x_0) + f(x_0 + \Delta x)s] - \frac{(\Delta x)3}{12} f''(\xi) \tag{3.3}$$

where $x_0 \leq \xi \leq x_0 + \Delta x$. It gives a formula that approximates an integral over a finite interval $[x_0, x_n]$ by dividing the interval into n sub-intervals of length Δx that can then be estimated with the trapezoidal rule. The position of the ξ cannot be known without knowing the integral exactly. For this reason, the last term is usually neglected and introduces an error in the numerical evaluation that is proportional to $1/n^2$.

Gaussian quadrature

While Newton-Cotes type formulae are used if the value of the integrand is given at equally-spaced points, it can be more suitable to use other methods such as Gaussian quadrature if it is possible to change the points at which the integrand is evaluated. The main formula of Gaussian quadrature states that if a weight function *w(x)* exists on [a,b], then there exist also weights w_j and abscissas x_j for $1 \leq j \leq n$ such that

$$\int_a^b dx\, w(x) f(x) = \sum_{j-1}^{n} w_j f(x_j) + \frac{f^{2n}(\xi)}{(2n)!} \int_a^b dx\, w(x) \left[\prod(x)\right]^2 \tag{3.4}$$

Multi-dimensional integration

All the classic numerical integration techniques are are defined for one-dimensional integrals. In many problems, such as the phase space integral depicted above, multi-dimensional integrals occur. They can often not be solved analytically and have to be evaluated numerically. A possible solution would be to iterate the multi-dimensional integral by applying a one-dimensional integration rule in each iteration and combining the pieces at the end. However, the error of this approach scales as $N^{-2/d}$ with N as the number of function evaluations and d as the number of dimensions. This means that the number of required function calls for a specific numerical accuracy scales exponentially.

3.2 Monte Carlo techniques

As shown in the previous section, classical numerical integration techniques are inefficient for multi-dimensional integrals. The key feature of Monte Carlo integration is the independence of the number of dimensions and that the error scales only with the number of function calls like $1/\sqrt{N}$. Even though this is a big improvement

compared to its alternatives, a convergence by a rate of $1/\sqrt{N}$ is still too slow for a fast and accurate result. Thus, some ways to improve the efficiency of Monte Carlo integrations even further will be presented.

Monte Carlo integration

In order to show the power of Monte Carlo integration, we go back to the basic idea at the beginning of this chapter and consider an integral of a square-integrable function over the unit hypercube $[0,1]^d$ that depends on d variables $u_1, ..., u_d$. To keep it simple, we will label a point in the unit hypercube as $x = (u_1, ..., u_d)$ and the evaluation of the function at this point as $f(x) = f(u_1, ..., u_d)$. The integral

$$I = \int dx\, f(x) = \int d^d u\, f(u_1, ..., u_d) \tag{3.5}$$

can be estimated using Monte Carlo integration to

$$E = \frac{1}{N} \sum_{n=1}^{N} f(x_n) \tag{3.6}$$

which converges to the true value of the integral due to the law of large numbers. For a finite N, we can now estimate the error by calculating the *variance* $\sigma^2(f)$ of the function *f(x)*:

$$\sigma^2(f) = \int dx (f(x) - I)^2 \tag{3.7}$$

It can be shown that

$$\int dx_1 ... \int dx_N \left(\frac{1}{N} \sum_{n=1}^{N} f(x) - I \right)^2 = \frac{\sigma^2(f)}{N}, \tag{3.8}$$

indicating that the mean error in the Monte Carlo estimate is $\sigma(f)/\sqrt{N}$ on average. It is important to realize that Monte Carlo integration gives only a probabilistic error bound. This means that we can only give a probability that the Monte Carlo estimate lies within a certain range of the true value and not a deterministic error bound as in the trapezoidal rule.

Variance reducing techniques

It has been shown how the Monte Carlo error scales. This is both a huge advantage (due to the independence of the integral's dimension) and a problem (since it converges

relatively slow to the true value). In the following sections we introduce several techniques to improve the scaling behaviour even further.

Stratified sampling

A fundamental property of the Riemann integral is the fact that one can divide the full integration space into subspaces

$$\int_0^1 dx\, f(x) = \int_0^a dx\, f(x) + \int_a^1 dx\, f(x) \tag{3.9}$$

This idea can also be applied if the integral is evaluated using Monte Carlo techniques. One performs a Monte Carlo integration in each subspace, and adds up the partial result at the end. The division of the hypercube can be imagined similarly. The procedure can lead to a dramatic variance reduction compared to crude Monte Carlo if the subspaces and the number of points in each subspace are chosen carefully. This is indeed necessary, since an inappropriate choice can also lead to a larger variance.

Importance sampling

Importance sampling is a general technique for estimating properties of a particular distribution, while only having samples generated from a different distribution. Mathematically, importance sampling corresponds to a change of integration variables:

$$\int dx\, f(x) = \int dx\, \frac{f(x)}{p(x)} p(x) = \int dP(x)\, \frac{f(x)}{p(x)} \tag{3.10}$$

with

$$p(x) = \frac{\partial^d}{\partial x_1 ... \partial x_2} P(x) \tag{3.11}$$

If we restrict $p(x) \geq 0$ and $\int dx\, p(x) = 1$, then $p(x)$ can be interpreted as a probability density function. Now we *just* have to choose *p(x)* such that it approximates $|f(x)|$ reasonably well in shape and that it can be used to generate random numbers that are distributed according to *P(x)*.

Control variates

As with importance sampling one seeks an integrable function that approximates the function f to be integrated reasonably well, but this times the two functions are substracted rather than divided:

$$\int dx\, f(x) = \int dx (f(x) - g(x)) - \int dx\, g(x) \tag{3.12}$$

If the integral of g is known, the only uncertainty comes from the integral of $(f - g)$, which can have a smaller variance than f if g is chosen carefully.
There are several other methods that serve the same purpose, but they will not be discussed here.

Adaptive Monte Carlo techniques

The problem with all variance-reducing techniques described above is that they require some knowledge of the integrand in advance. Since we deal very often with functions that can not be solved analytically this knowledge is not generally available. For this reason, one usually prefers adaptive techniques, i.e. an algorithm which learns about the function as it proceeds.

Multi-channel Monte Carlo

If the integrand $f(x)$ has sharp peaks, crude Monte Carlo usually leads to poor results. Multi-Channel Monte Carlo offers a solution if the transformations for a single peak structure are known (each such transformation is known as a channel).

3.3 Random numbers

It is now clear that Monte Carlo Integration offers a tool for the numerical evaluation of integrals in high dimensions. Since random numbers play an important role in the Monte Carlo ansatz, we will finish this introduction with a brief discussion of their nature.
In the most general sense a computer is a fully deterministic machine and it can never generate truly random numbers. Instead, we have to use so-called *pseudo-random* or *quasi-random* numbers.

Pseudo-random numbers

Pseudo-random numbers are produced in the computer deterministically by a simple algorithm. They are not truly random, but appear to be random to someone who does not know the algorithm. By today's standards a good random number generator should satisfy criteria like *good distribution, long period, repeatability, long disjoint subsequences, portability and efficiency.* To test all those qualities and ensure that a given set of random numbers is adequate for a given problem, there are a certain

number of tests available (that will not be discussed here).

Quasi-random numbers

Quasi-random numbers are actually not random at all, but produced by a numerical algorithm that is designed to distribute them as uniformly as possible. A use of quasi-random numbers is valid nonetheless since the true randomness of the generated numbers is not that important in Monte Carlo integrations. It is more relevant that the integration region is sampled as uniformly as possible. This is why the use of quasi-random numbers can in fact reduce the errors in Monte Carlo generation.

4 The RAMBO phase space generator

Knowing the basics of Monte Carlo integration, one can now go back to the original physical problem, i.e. the generation of events from a multiparticle phase space. This problem is addressed in [14], where a method is proposed that generates phase-space points with uniform weights. The FORTRAN program that resulted from this paper has been used successfully in various HEP Monte-Carlo generators for many years. This thesis will provide a new implementation of this method in C++.
The most widely used approach to generate phase-space distributions is to implement the particle production *hierarchically* as a series of sequential two-body decays. In order to avoid the variable event weights that result from this strategy, RAMBO (short for RAndom Momenta Beautifully Organized) uses a *democratic* procedure that treats all particles equally.
The goal of the implementation is the ability to generate decays of a parent particle of given momentum $\mathbf{P}$ = (P,0,0,0) that result in four-momenta of the N daughter particles. In the given scenario, the phase space volume for N-body decay is given by

$$V(\sqrt{P^2}) = \int \delta^4 \left(P - \sum_i^N p_i \right) \prod_i^N [d^4 p_i \, \delta(p_i^2) \theta(p_i^0)] \tag{4.1}$$

where

- the four dimensional δ-function ensures energy momentum conservation,
- $\delta(p_i^2)$ ensures that the daughter particles are real (not virtual), and
- θ is a function which takes into account several detector effects and can be set to unity.

The power of the idea behind the RAMBO algorithm is the ansatz that once one has generated an arbitrary phase space region that does not necessarily satisfy energy-momentum conservation, one can get to a system that meets the requirements of these constraints by first generating all particle momenta isotropically and then boosting and rescaling them according to a certain conformal transformation. To give some more insights into the RAMBO algorithm, the general mathematical idea

is outlined first and then the procedure is described more detailed in Sec. 4.2.

4.1 Overview

We define a quantity

$$R_n \equiv \int \prod_i^N [d^4 q_i \, \delta(q_i^2) f(q_i^0) \theta(q_i^0)] \tag{4.2}$$

and assume that

$$Q^\mu = \sum_{i=1}^N q_i \tag{4.3}$$

is the vector sum of the all the random 4-vectors generated in our new system and

$$P^\mu = \sum_{i=1}^N p_i \tag{4.4}$$

is the vector sum of the desired "well behaved" 4-vectors of the true particles with zero mass. RAMBO offers the following conformal transformation from Q^μ to P^μ.

$$p_i^0 = \gamma q_i^0 + \vec{b} \cdot \vec{q_i} \qquad\qquad \vec{p_i} = x(\vec{q_i} + \vec{b} q_i^0 + a(\vec{b} \cdot \vec{q_i})\vec{b}) \tag{4.5}$$

Notice that the arrows indicate the 3-vectors and the index i represents the i^{th} daughter particle. The single terms are defined as

$$\begin{aligned} \vec{b} &= \frac{\vec{Q}}{M} \text{ with } M = \sqrt{Q^\mu Q_\mu} = \sqrt{Q^2} \\ \gamma &= \frac{Q^0}{M} = \sqrt{1 + b^2} \\ a &= \frac{1}{1+\gamma} \\ x &= \frac{\sqrt{P^\mu P_\mu}}{M} = \frac{\sqrt{P^2}}{M} \end{aligned} \tag{4.6}$$

In a second step, the p_i^μ are then transformed into new physical four-momenta of non-zero mass

$$K^\mu = \sum_{i=1}^N k_i \tag{4.7}$$

with mass m_i as follows:

$$k_i^0 = \sqrt{m_i^2 + \xi^2 p_i^{0\,2}} \qquad\qquad \vec{k_i} = \xi \vec{p_i} \tag{4.8}$$

4.2 The Algorithm

As said before, sequential algorithms generate weighted events. The ideal case gives unweighted events, meaning that all generated events are of equal probability.

Massless particles

The first goal is thus to generate independently n massless four-momenta q_i^μ with isotropic angular distribution, energies (q_i^0) and uniform weight. RAMBO does not use the phase space given in equation 4.1, but starts by defining the quantity

$$R_n \equiv \int \prod_i^N [d^4 q_i \, \delta(q_i^2) f(q_i^0) \theta(q_i^0)] = \left[2\pi \int_0^\infty x f(x) dx \right]^n \tag{4.9}$$

which can be interpreted as a phase-space like object describing a system of n massless four-momenta q_i^μ that are not constrained by momentum conservation but occur with some weight function f that ensures that the total volume stays finite. In a next step, the conformal transformation given in equation 4.5 relates the four-vectors q_i^μ to their physical counterparts p_i^μ by transforming the overall momentum of the set q_i^μ into the desired P^μ.

This is done with a sequence of random numbers $\{\{\rho\}_1, \{\rho\}_2, \ldots, \{\rho\}_N\}$ where N is the number of daughter particles and $\{\rho\}_i$ itself is a set of 4 random numbers $\{\rho_{i1}, \rho_{i2}, \rho_{i3}, \rho_{i4}\}$. The random numbers should be uniformly distributed in the closed interval (0,1), which means the numbers should never be 0 or 1 (boundaries excluded). This is achieved by using the ROOT [20] class TRandom2, a random number generator that is based on the maximally equidistributed combined Tausworthe generator by L'Ecuyer [15]. It has a period of 2^{88} (around 10^{26}) [16] and should be sufficient for our purpose.

Since it is a decay process, the massless three-vectors need to be distributed according to the density $q_i^0 e^{-q_i} dq_i^0$. This can be achieved using the method of exponential deviates with uniform random numbers. One can generate the following quantities:

$$c_i = 2\rho_{i1} - 1 \qquad\qquad \phi_i = 2\pi\rho_{i2} \tag{4.10}$$

Now we can generate the required $q_i^\mu = (q_i^0, q_i^x, q_i^y, q_i^z)$

$$\begin{aligned} q_i^0 &= -\ln(\rho_{i3}\rho_{i4}) & q_i^x &= q_i^0\sqrt{1-c_i^2}\,\cos\phi_i \\ q_i^y &= q_i^0\sqrt{1-c_i^2}\,\sin\phi_i & q_i^z &= q_i^0 c_i \end{aligned} \tag{4.11}$$

They enable us to address the problem using a Monte Carlo approach, which is highly favorable as depicted in 3.2.

Every generated Monte Carlo event has to be weighted. In this case, the weights are constant and can be calculated to

$$W_0 = \left(\frac{\pi}{2}\right)^{n-1} \frac{w^{2n-4}}{\Gamma(n)\Gamma(n-1)} \tag{4.12}$$

where w is the center of mass energy.

Massive particles

Contrary to hierarchical procedures where there is no fundamental difference between generating massless and massive particles, the democratic approach requires some changes due to the fact that they can obviously not be scaled without varying their masses. Nevertheless, it is possible to transform the momentum components of a set of massless momenta that obey momentum conservation in a way that the transformed momenta have non-zero mass and still satisfy the conditions of the given phase space. The corresponding transformation as given in eq. 4.7 is

$$k_i^0 = \sqrt{m_i^2 + \xi^2 p_i^{02}} \qquad \vec{k_i} = \xi\vec{p_i} \tag{4.13}$$

where ξ is the solution of the equation

$$w = \sum_{i=1}^{n} \sqrt{m_i^2 + \xi^2 p_i^{02}}. \tag{4.14}$$

Since the sum and the square root do not commute there is no general analytic expression for ξ, but it can be computed numerically very quickly and to high accuracy using an iterative algorithm.

Now, the weights are not constant any more but depend on the four-vectors of the daughter particles.

$$W_n = \left[\frac{1}{N}\sum_{i=1}^{n}|\vec{k_i}|\right]^{2n-3} \left[\prod_{i=1}^{n}\frac{|\vec{k_i}|}{k_i^0}\right] \left[\sum_{i=1}^{n}\frac{|\vec{k_i}|^2}{k_i^0}\right]^{-1} \tag{4.15}$$

Consequently, the efficiency of the weight distribution, which is defined as the average event weight divided by the maximum weight, goes down.

We now have both an algorithm that generates Monte Carlo events with massless momenta and weights W_0 and an algorithm that transforms those massless momenta into massive ones with weights W_m. Now we are able to generate any type of final state by subsequently applying these two algorithms and multiplying the corresponding weights, $W = W_0 \cdot W_m$.

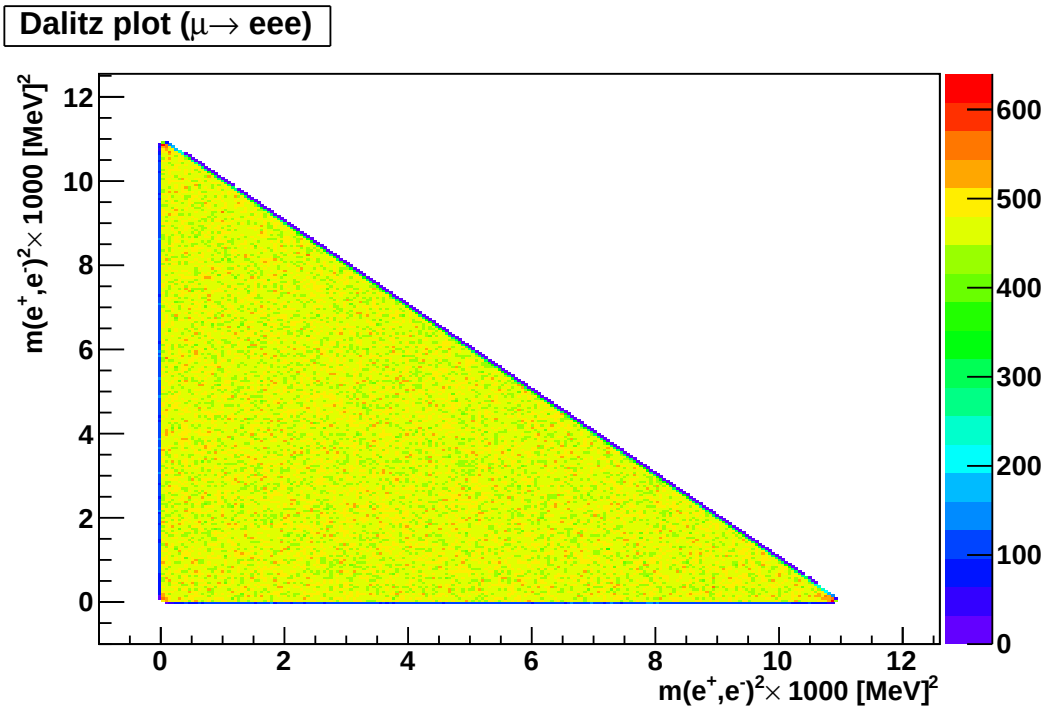

Figure 3: Dalitz plot

5 Results

5.1 Simulation

There are several ways to test the quality of the simulated phase space region, one of them being a so-called *Dalitz Plot* [18], a scatter-plot that can be used to depict the dynamics of a three-body decay. By choosing the squares of the invariant masses of two (different) pairs of the decay products as the axis, one can completely describe the kinematics in one plot. This has been done for $\mu \to 3e$ (Fig. 3) and $p\bar{p} \to 3\pi^0$ (Fig. 4). The results where exactly as expected. The triangles represent momentum conservation and the rounded corners are due to the fact that the possible number of combinations decreases to the extremes. The Dalitz Plot for $p\bar{p} \to 3\pi^0$ can be compared to a similar Plot from the Crystal barrel Experiment (Fig. 5) that has been done with real data, and the similarities are obvious. The simulated plots do not show any features though which is reasonable because we did not take the matrix elements into account. Other then that, the plots sufficiently consistent. This proves

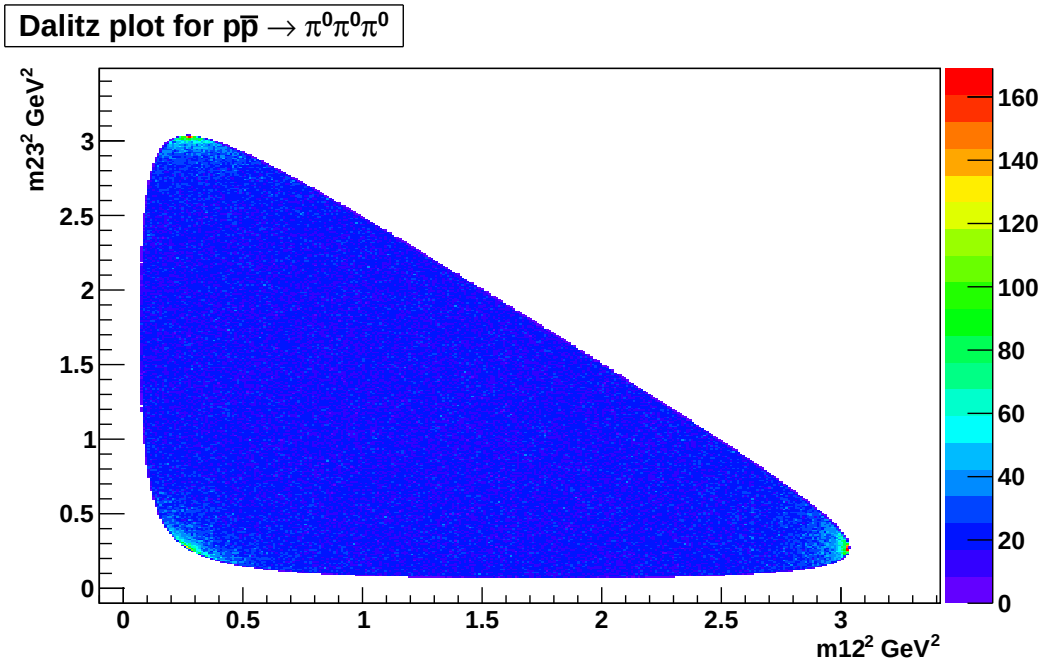

Figure 4: Dalitz plot (simulated)

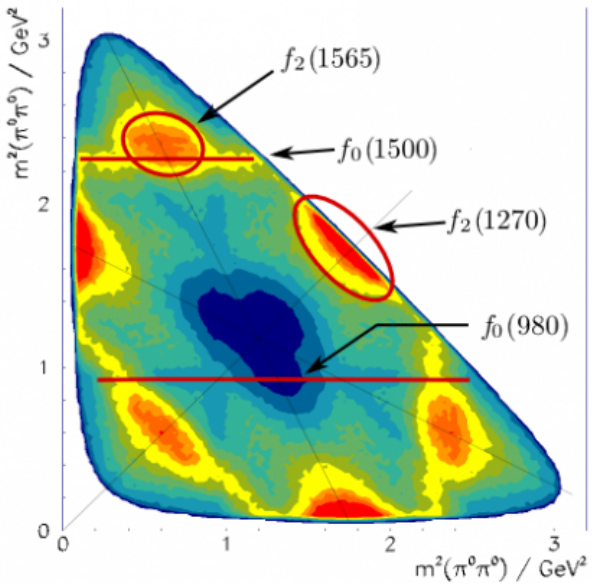

Figure 5: Dalitz plot(Crystal Barrel Experiment)

that the implementation of the RAMBO algorithm in C++ was successful.

6 Summary and Outlook

Summary

During the work on this thesis, a phase space generator has been implemented in $C++$ according to the ideas presented in [14]. It was possible to show the viability of the code using Dalitz Plots that were plotted with the data from the simulated event.

Outlook

The phase space generator can now be used for the simulation framework of the planned $\mu \to eee$ experiment to generate signal Monte Carlo. It might also be possible to combine the RAMBO procedure with the integration routine VEGAS from the CUBA library [21] in order to generate BSM backgrounds where we know the matrix element in principle.

References

[1] Sheldon L. Glashow, *Partial-Symmetries of Weak Interactions*, Nuclear Physics 22, 579 (1961)

[2] Steven Weinberg, *A Model of Leptons*, Physical Review Letters 19, 1264 (1967)

[3] Abdus Salam, *Weak and electromagnetic interactions*, Imperial College of Science and Technology, London (1968)

[4] K. Nakamura et al., *Particle Data Group*, Journal of Physics G37, 075021 (2010)

[5] R.R. Crittenden et al., *Radiative decay modes of the muon*, Phys.Rev. 121, 1823 (1961)

[6] Bertl et al., *Search for the decay* $\mu^- \to e^- e^+ e^-$, Nucl.Phys. B260, 1823 (1985)

[7] S.J. Freedman et al., *Limits on neutrino oscillations from anti-electron-neutrino appearance*, Phys.Rev. D47, 811 (1993)

[8] M.L. Brooks et al., *New limit for the family number nonconserving decay* $\mu^- \to e^+\gamma$, Phys.Rev.Lett. 83, 1521 (1999)

[9] Bellgardt et al., *Search for the Decay* $\mu^+ \to e^+ e^+ e^-$, Nucl.Phys. B299, 1 (1988)

[10] R.D. Bolton et al., *Search for Rare Muon Decays with the Crystal Box Detector*, Phys.Rev. D38, 2077 (1988)

[11] A. Schoening et al., *A Novel Experiment to Search for the Decay* $\mu \to eee$, Physics Procedia 00, 1 (2010)

[12] Particle Data Group, *Review of Particle Physics*, Volume 37, Chapter 27 "Passage of particles through matter" (2010)

[13] Stefan Weinzierl, *Introduction to Monte Carlo methods*, NIKHEF Theory Group (lecture) (2000)

[14] Kleiss, Stirling and Ellis *A new Monte Carlo Treatment of Multiparticle phase space at high energies*, Computer Physics Communications 40, 359 (2000)

[15] Pierre L'Ecuyer *Maximally equidistributed combined Tausworthe Generators*, mathematics of Computation 65, 203 (1996)

[16] *ROOT class TRandom 2*, details can be foudn here: http://root.cern.ch/root/html/TRandom2.html

[17] Bhabha et al. *The Scattering of Positrons by Electrons with Exchange on Diracs Theory of Positron* Royal Society (1935)

[18] R.H. Dalitz *Decay of τ Mesons of Known Charge* Phys.Rev. 94, 1046 (1954)

[19] B.Riemann *Ueber die Darstellbarkeit einer Function durch eine trigonometrische Reihe* Abhandlungen der Koeniglichen Gesellschaft der Wissenschaften zu Goettingen 13, 87 (1868)

[20] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[21] T. Hahn, *Cuba–a library for multidimensional numerical integration*, Computer Physics Communications 168 (2005)

List of Figures

I, Christoph Dressler, hereby declare that this thesis is entirely my own work except where otherwise indicated. It is being submitted for the degree of *Bachelor of Science* at the University of Heidelberg (Germany) and has not been submitted before for any degree or examination at this or any other University.

Date and place	Christoph Dressler

To my parents, Sabine and Christian Dressler, I gratefully dedicate this thesis. Thank you for always supporting me. It's been a long, hard road – but it was worth it.

I would like to thank Mr. Schoening
for being a great advisor. His ability to share knowledge in an incredible instructing and motivating way has made the work on this thesis a lot easier.

I would also like to thank Rohin
for all his clarifications and help during the last months. This thesis would not have been possible without his assistance.

Thanks to Moritz
for giving me the most vivid explanations ever. When I was lost in physical no man's-land he was there to save me, and I can't tell how much I appreciate it.

Thanks to Patricia, Arno, Robert, Botho, Gregor and Sebastian
for making the time in this group the best time of my physics studies.

Thanks to Florian
for reading this mess and throwing in those funny comments. Rambo strikes back.

And, above all, special thanks to Kika
for your never ending patience. Sorry for my temporary inability to enjoy anything – you really earned yourself a cookie and some quality time.

A Programm code

main.cc

```
#include "rambo.h"
#include <iostream>
#include "TLorentzVector.h"
#include "TFile.h"
#include "TH1F.h"
#include "TH2F.h"
#include "TPad.h"
#include "TVector3.h"
#include "TRandom2.h"
#include "TMath.h"
#include <vector>
#include <valarray>
#include <cstdlib>

using namespace std;

int main()
{
  TLorentzVector parentMomenta(0, 0, 0, 105);
  int n_daughter = 3;
  rambo RAMBOgeneral(n_daughter, parentMomenta);
  cout << "generating..." << endl;
  RAMBOgeneral.generate();
  cout << "drawing..." << endl;
  RAMBOgeneral.draw();
}
```

.

rambo.h

```
#ifndef rambo_h
#define rambo_h

/* Random Momenta Beautifully Organised(RAMBO) is a Democratic Phase
    Space generator
 */

#include <iostream>
#include "TLorentzVector.h"

#include "TFile.h"
#include "TH1F.h"
#include "TH2F.h"
#include "TPad.h"
#include "TVector3.h"
#include "TRandom2.h"
#include "TMath.h"
```

```
#include <vector>
#include <valarray>

class rambo
{
  public:
     rambo(int, TLorentzVector);
     ~rambo();
     void generate();
     void draw();

  protected:
     int num_daughter;
     TLorentzVector m_parentMomenta;
     std::vector<double> m_daughterMassContainer;

     std::valarray<TLorentzVector> m_daughterMomenta_random;
     std::vector<TLorentzVector> m_daughterMomenta_massless;
     std::vector<TLorentzVector> m_daughterMomenta_massive;

     std::vector<std::vector<TLorentzVector> >
         m_daughterMomentaInAllEvents;
     std::vector<TLorentzVector> m_daughterMomentaContainer;

     void GenerateQ();
     void GenerateP();
     void CalculateWeightsMassless();
     void CalculateWeightsMassive();
     int fak_n(int num
     );

     double m_weightMassless;
     std::vector<double> m_weightsInAllEvents;

           std::vector<TLorentzVector> GenerateMass();

};

#endif
```

.

rambo.cxx

```
#include "rambo.h"
#include "TMath.h"
#include <iostream>

using namespace std;

rambo::rambo(int n, TLorentzVector parent)
{
```

```
  num_daughter = n;
  m_parentMomenta = parent;
  m_daughterMassContainer.clear();
  for(int i = 0; i < num_daughter; ++i)
  {
    m_daughterMassContainer.push_back(0.511);
  }
  m_daughterMomentaInAllEvents.clear();
}

rambo::~rambo()
{}

//generate

void rambo::generate()
{
  int n_events = 100000;

  m_daughterMomentaInAllEvents.clear();
  for(int i = 0; i < n_events; i++)
  {
    this->GenerateQ();
    this->GenerateP();
    this->m_daughterMomenta_massive = GenerateMass();
    this->CalculateWeightsMassless();
    this->CalculateWeightsMassive();

    m_daughterMomentaInAllEvents.push_back(m_daughterMomenta_massive);
  }

}

//GenerateQ

void rambo::GenerateQ()
{
  TRandom2 rand(0);
  m_daughterMomenta_random.resize(num_daughter);

  for(int i =0; i < num_daughter; i++)
  {
          double rho1   = rand.Uniform(0,1);
          double rho2   = rand.Uniform(0,1);
          double rho3   = rand.Uniform(0,1);
          double rho4   = rand.Uniform(0,1);
          double c_i    = 2*rho1 - 1;
          double phi_i  = 2*TMath::Pi()*rho2;

          double q0     = -TMath::Log(rho3*rho4);
```

```
            double qx      = q0 * std::sqrt(1 - c_i*c_i) * TMath::Cos(
                phi_i);
            double qy      = q0 * std::sqrt(1 - c_i*c_i) * TMath::Sin(
                phi_i);
            double qz      = q0 * c_i;

            TLorentzVector Qi;
            Qi.SetPxPyPzE(qx,qy,qz,q0);
            m_daughterMomenta_random[i] = Qi;
  }
}

//GenerateP

void rambo::GenerateP()
{
  TLorentzVector Q_mu = m_daughterMomenta_random.sum();
  double M = std::sqrt(Q_mu.Mag2());
  TVector3 b = -Q_mu.Vect()*(1/M) ;
  double gamma = Q_mu.E()/M;
  double a = 1/(1 + gamma );
  double x = std::sqrt(m_parentMomenta.Mag2())/M;

  m_daughterMomenta_massless.clear();
  for(int i = 0; i < num_daughter; i++)
  {
    double qiE    = m_daughterMomenta_random[i].E();
    TVector3 qi_3Vec = m_daughterMomenta_random[i].Vect();
    double b_dot_q    = b.Dot(qi_3Vec);
    double p0         = x*(gamma*qiE + b_dot_q);
    TVector3 pi_3Vec = x*( qi_3Vec + b*qiE + a*( b.Dot(qi_3Vec) )*b);
    TLorentzVector aMomenta(pi_3Vec, p0);
    m_daughterMomenta_massless.push_back(aMomenta);
  }

  cout << "m_daughterMomenta_massless size: " <<
      m_daughterMomenta_massless.size() << endl;
}

//GenerateMass

std::vector<TLorentzVector> rambo::GenerateMass()
{
  cout << "generate mass" << endl;
       std::vector<TLorentzVector> MassiVeDaughterVector;

  if (m_daughterMomenta_massless.size() == m_daughterMassContainer.size
      () )
  {
    const int ContSize       = m_daughterMassContainer.size();
          MassiVeDaughterVector.resize(ContSize);
```

```
106         const int iterMax        = 1000;
            const double Accuracy    = 1e-5;

            std::valarray<double> E(ContSize);
            std::vector<double> XM2(ContSize);
111         std::valarray<double> P2(ContSize);

            for(int i =0; i < ContSize; i++)
            {
                  XM2[i]    = m_daughterMassContainer[i] *
                      m_daughterMassContainer[i];
116               P2[i]     = m_daughterMomenta_massless[i].E() *
                      m_daughterMomenta_massless[i].E();
            }

            double XMT =0;
            double ET  = std::sqrt(m_parentMomenta.Mag2());//Rest mass
                energy of parent
121
            for(int i =0; i< ContSize; i++)
            {
                  XMT  = XMT + m_daughterMassContainer[i];
            }
126
            double XMAX = std::sqrt( 1 - (XMT/ET)*(XMT/ET));
            double X    = XMAX;
            int n_iter  = 0;

131         while(n_iter < iterMax)
            {
                  double F0 = -ET;
                  double G0  = 0;
                  double X2 = X*X;
136
                  for(int i =0; i < ContSize; i++)
                  {
                      E[i] = std::sqrt(XM2[i] + X2*P2[i]);
                      G0   = G0 + P2[i]/E[i];
141               }

                  F0 = F0 + E.sum();

                  if(fabs(F0) <= Accuracy)
146               {
                      for(int i = 0; i < ContSize; i++)
                      {
                            TLorentzVector massive(
                                m_daughterMomenta_massless[i].Vect(), E[i
                                ]);
                            MassiVeDaughterVector[i] = massive
151               }
```

```
                    break;
                }
                else
                {
                    n_iter++;
                    X = X - F0/(X*G0);
                    std::cout<< "Iterations: "<<n_iter<<endl;
                }

                if(n_iter == iterMax)
                {
                    std::cout<< "Iterations: "<<n_iter<<endl;
                    std::cout<<"RAMBO DID not CONVERGE"<<std::endl;
                }
                std::cout <<"check 1" <<std::endl;
            }
            std::cout <<"check 2" <<std::endl;
    }
    std::cout <<"check 3" <<std::endl;
        return MassiVeDaughterVector;
        std::cout <<"check 4" <<std::endl;
}

int rambo::fak_n (int num)
  {
    if (num > 1)
    {
      return fak_n(num - 1)*num;
    }
    return 1;
  }

void rambo::CalculateWeightsMassless()
{
  double w     = m_parentMomenta.E();
  double pi        = 3.1415;

  double prod1  = pow((pi/2),num_daughter - 1);
  double prod2  = pow(w,2*num_daughter - 4);

  double W_0  = (prod1*prod2)/((this->fak_n(num_daughter - 1 ))*(this->
      fak_n(num_daughter - 2)));

  m_weightMassless= W_0;

}


//Calculating the massive weights

void rambo::CalculateWeightsMassive()
```

```
{
  double w = m_parentMomenta.E();

  double sum1   = 0;
  double sum1a  = 0;
  double sum1b  = 0;
  double sum2   = 0;
  double sum2a  = 0;
  double prod1  = 1;



    for(int i =0; i < num_daughter;i++)
    {
      //first sum
      double k_mag = m_daughterMomenta_massive[i].Vect().Mag();

      sum1a = sum1a + k_mag;

        //Second sum

        double k_mag2 = m_daughterMomenta_massive[i].Vect().Mag2();
        double k_i0 = m_daughterMomenta_massive[i].E();
        double k_ratio2 = k_mag2/k_i0;

        sum2a = sum2a + k_ratio2;

        //first product

        double k_ratio = k_mag/k_i0;
        prod1 = prod1 * k_ratio;
          }

    //calculation of the first sum
    sum1b   = (1/w)*sum1a;
    sum1  = pow(sum1b,2*num_daughter - 3);

    //calculation of the second sum
    sum2  = pow(sum2a,-1);

    double Wm = sum1*sum2*prod1;
    cout <<"Sum 1 : "<<sum1 <<endl;
    cout <<"Sum 2 : " << sum2 <<endl;
    cout <<"prod1 : " << prod1 <<endl;

    std::cout <<"Massive Weights: " << Wm <<std::endl;

    m_weightsInAllEvents.push_back(Wm);
    }
```

```
//Draw-function for Dalitz Plot for massive vectors (K)

void rambo::draw()
{

  cout << "init histograms" << endl;

  //Histograms of the daugher momenta

  TH1F* h1 = new TH1F("daughter1", "Daughter 1 Pt; MeV", 100,-10,110);
  TH1F* h2 = new TH1F("daughter2", "Daughter 2 Pt; MeV", 100,-10,110);
  TH1F* h3 = new TH1F("daughter3", "Daughter 3 Pt; MeV", 100,-10,110);
  TH1F* sumMomenta = new TH1F("SumMomenta","Magnitude of Sum of momenta
      ",100,-10,10);

        TH1F* momentaD1 = new TH1F("momentumDaugther1","Daughter 1
            Momentum",100,-10,110);
        TH1F* momentaD2 = new TH1F("momentumDaugther2","Daughter 2
            Momentum",100,-10,110);
        TH1F* momentaD3 = new TH1F("momentumDaugther3","Daughter 3
            Momentum",100,-10,110);

        //Dalitz Plot
        TH2F* Dalitz = new TH2F("Dalitz","Dalitz plot (mu->eee)"
            ,500,-1,15,500,-1,15);
  Dalitz->GetXaxis()->SetTitle("m12*m12");
  Dalitz->GetYaxis()->SetTitle("m23*m23");

        //Open a ROOT file
        TFile *outFile = new TFile("output.root","RECREATE");

        //Histogram of the weight
        TH1D *weights = new TH1D("weights","Rambo weights", 100000, 0,
            1);
        //not done yet

  cout << "filling histograms" << endl;
  for(int i = 0; i < m_daughterMomentaInAllEvents.size(); i++)
  {
    TLorentzVector firstDaughter = m_daughterMomentaInAllEvents[i][0];
    TLorentzVector secondDaughter= m_daughterMomentaInAllEvents[i][1];
    TLorentzVector thirdDaughter = m_daughterMomentaInAllEvents[i][2];

    double firstDaughterPt      = firstDaughter.Pt();
    double secondDaughterPt     = secondDaughter.Pt();
    double thirdDaughterPt      = thirdDaughter.Pt();

    h1->Fill(firstDaughterPt);
    h2->Fill(secondDaughterPt);
    h3->Fill(thirdDaughterPt);
```

```
            momentaD1->Fill(firstDaughter.P());
            momentaD2->Fill(secondDaughter.P());
            momentaD3->Fill(thirdDaughter.P());

            //Calculate Datliz Variables
            double m12_sqr   = (firstDaughter + secondDaughter)*(
                firstDaughter + secondDaughter);
            double m23_sqr   = (secondDaughter + thirdDaughter)*(
                secondDaughter+ thirdDaughter);

            m12_sqr = m12_sqr/1000000; //GeV^2
            m23_sqr = m23_sqr/1000000;

            //Sum Momena
            TVector3 Momentum_1 = firstDaughter.Vect();
            TVector3 Momentum_2 = secondDaughter.Vect();
            TVector3 Momentum_3 = thirdDaughter.Vect();

            double  sumVec = (Momentum_1 + Momentum_2 + Momentum_3).Mag()
                ;
            sumMomenta->Fill(sumVec);

                Dalitz->Fill(m12_sqr,m23_sqr);

                //Fill weights histogramm
                weights->Fill(m_weightsInAllEvents[i]);

        }
        cout << "Massive weights size: "<<m_weightsInAllEvents.size()<<
            endl;
        cout << "storing histograms on disk" << endl;

    h1->Write();
        h2->Write();
        h3->Write();
        Dalitz->Write();

        momentaD1->Write();
        momentaD2->Write();
        momentaD3->Write();

    sumMomenta->Write();
    weights->Write();
        outFile->Close();

}
```

xsifinder.cxx

```
#include "xsifinder.h"
#include "TF1.h"
#include "TLorentzVector.h"
#include "TVector3.h"
#include "TMath.h"

//works only for 3 particles
double Rootfunctor( double *x , double *param)
{
    int n_par    =3*2;
    double parentMass = param[0];

    //loop variables
    unsigned int lc = 0;
    unsigned int parIndex1 = 1;
    unsigned int parIndex2 = 2;
    unsigned int xindex =0;

    double sum_sqrt =0;
    while (lc <=n_par)
    {
        sum_sqrt  += TMath::Sqrt(param[parIndex1]*param[parIndex1] +
            param[parIndex2]*param[parIndex2]*x[xindex]*x[xindex]);
        parIndex1++; parIndex2++; xindex++; lc++;
    }
    double funcVal= parentMass - sum_sqrt;

    return funcVal;
}

xsifinder::xsifinder(double parentMass,std::vector<TLorentzVector>
    Daugther4Vector):
    m_f1(0),
    m_parentMass(parentMass),
    m_daugtherFourVectorCollection(Daugther4Vector),
    m_rootParamVector(0)
{
    unsigned int n_daughter = Daugther4Vector.size();
    unsigned int nParam = n_daughter*2;
    m_f1 = new TF1("f3",Rootfunctor,-100,100,nParam);
}

xsifinder::~xsifinder()
{}

void xsifinder::SetTF1Parameters()
{
    m_rootParamVector.push_back(m_parentMass);
    std::vector<TLorentzVector>::iterator vecIter;
```

```
    for(vecIter = m_daugtherFourVectorCollection.begin(); vecIter !=
        m_daugtherFourVectorCollection.end(); ++vecIter)
50  {
        TVector3 p  = vecIter->Vect();
        double E    = vecIter->E();
        double mass = E*E - p.Dot(p);
        m_rootParamVector.push_back(mass);
55      m_rootParamVector.push_back(E);
    }

    for(unsigned int i=0; i < m_rootParamVector.size(); ++i)
    {
60      m_f1->SetParameter(i,m_rootParamVector[i]);
    }
}
double xsifinder::getSolution()
{
65  this->SetTF1Parameters();
    //    return m_f1->GetX(0.,-100.,100.,1e-10,1000.); That's too many
          variables for my root version :(
    return m_f1->GetX(0.,-100.,100);
}
```