university of groningen / faculty of science and engineering / van swinderen institute for particle physics and gravity

Msc project

A fast tool to predict peaking and semi-peaking backgrounds in two body decay processes

Michiel Veen

Supervisor
dr. ir. C.J.G. Onderwater

Second examinator
prof. dr. A. Pellegrino

July 27, 2017

Abstract In rare decays background events are far more numerous than signal events. It is therefore essential to be able to predict backgrounds in the search for such decays. A fast tool was developed to predict peaking and semi-peaking backgrounds in two-body decays. Peaking backgrounds, due to misidentification, can be predicted and simulated in a matter of seconds, for any given decay. For semi-peaking backgrounds, both direct and cascade decays were considered. This tool can locate these backgrounds in several minutes.

Contents

1 Introduction **7**

2 Rare decays **9**
- 2.1 Theory . . . 9
- 2.2 Detection and analysis . . . 10
  - 2.2.1 Momentum Reconstruction . . . 10
  - 2.2.2 Select particle candidates . . . 11
  - 2.2.3 Reconstruct particle decay . . . 13
- 2.3 Types of backgrounds . . . 15
  - 2.3.1 Combinatorial . . . 15
  - 2.3.2 Peaking . . . 15
  - 2.3.3 Semi peaking . . . 16
- 2.4 Locating the backgrounds . . . 16

3 Database building **19**
- 3.1 Requirements . . . 19
- 3.2 Available databases . . . 20
  - 3.2.1 ROOT . . . 20
  - 3.2.2 Pythia . . . 22
  - 3.2.3 EvtGen . . . 23
- 3.3 Combining databases . . . 24
  - 3.3.1 Mapping the ROOT database . . . 25
  - 3.3.2 Mapping the Pythia database . . . 27
  - 3.3.3 Mapping the EvtGen database . . . 30
  - 3.3.4 Merging the three maps . . . 32
- 3.4 Writing combined database . . . 35
  - 3.4.1 Writing XML database . . . 36
  - 3.4.2 Writing ROOT database . . . 36
- 3.5 Remapping combined database . . . 39

4 Peaking backgrounds **41**
- 4.1 Simulation without misidentification . . . 41
- 4.2 Input parameters . . . 42
- 4.3 Selecting Peaking backgrounds . . . 44
  - 4.3.1 Background candidates . . . 45

4.3.2 Reducing the number of background candidates . . . 47
4.4 Simulating peaking backgrounds . . . 50
4.5 Validation . . . 53
4.5.1 D^0 to $e\mu$ . . . 53
4.5.2 B_s^0 to $\mu\mu$ . . . 53

5 Semi peaking backgrounds **57**
5.1 Partial reconstruction . . . 57
5.2 Cascades . . . 57
5.2.1 Cascade construction . . . 59
5.3 Finding the first daughter particle . . . 60
5.3.1 Case study: $e^+\mu^-$ . . . 62
5.4 Finding the second daughter particle . . . 63
5.4.1 Case study: $e^+\mu^-$ . . . 66
5.5 Writing and reading cascades . . . 68
5.5.1 Writing cascade file . . . 68
5.5.2 Reading cascade file . . . 71
5.6 Stopping criteria . . . 71
5.7 Invariant mass spectrum . . . 71
5.7.1 Calculating cascade endpoint . . . 74
5.8 Visualizing cascades . . . 75
5.8.1 Illustrating cascades . . . 76
5.8.2 Outlook: cascade simulation . . . 76
5.9 Validation . . . 77

6 Conclusion **79**

Appendices **85**

A Program functions **87**
A.1 ROOTcount . . . 89
A.2 Pythiacount . . . 90
A.3 FParticle . . . 91
A.4 Particle maps . . . 91
A.5 ROOTmap . . . 92
A.6 Pythiamap . . . 95
A.7 EvtGenmap . . . 100
A.8 Merge Databases . . . 104
A.9 Writing XML database . . . 108
A.10 Writing ROOT database . . . 110
A.11 Initializing the databases . . . 113
A.12 Remapping the database . . . 114
A.13 Making a histogram of a misidentification . . . 118
A.14 Detector effects . . . 119
A.15 Global parameters . . . 121
A.16 Locating peaking backgrounds . . . 122
A.17 Filename generation . . . 127

A.18 Input of misidentification function . . . 128
A.19 Misidentification particles . . . 130
A.20 BSM particle types . . . 132
A.21 2 Body decay conserved properties . . . 133
A.22 Mass difference signal and background . . . 134
A.23 Branching ratio of peaking background . . . 135
A.24 Histogram title names . . . 137
A.25 Boost distribution . . . 138
A.26 cascade struct . . . 139
A.27 Finding first cascade particle . . . 140
A.28 Finding second cascade particle . . . 144
A.29 Cascade file . . . 147
A.30 Invariant mass of cascade . . . 150
A.31 Visualizing cascades . . . 152
A.32 simulating cascades . . . 157

B Peaking backgrounds for B_s^0 to $\mu\mu$ **161**

Chapter 1

Introduction

For the past century, the Standard Model (SM) has manifested itself as the leading theory in elementary particle physics. It has passed many precision tests. These include predictions on the anomalous magnetic moment of the electron and the mass and width of the W bosons [1][2]. All elementary particles predicted by the SM have also been found, with the most recent conformation, the Higgs boson, being the last one [3].

Despite its enormous success, there are phenomena not explained by the SM. Among these are major observations such as gravity and the nature of dark matter and dark energy. These unexplained phenomena have led to various proposals for Beyond the Standard Model (BSM) theories and experiments. Several of these experiments can be found at the Large Hadron Collider (LHC) at CERN, Geneva. This particle collider produces proton-proton (pp) collisions at energies up to 13 TeV [4]. At these energies, particle decays heavily suppressed in the SM can be observed. These rare decays provide a possible probe of BSM physics.

Where there are rare decays, there are also common decays. These are the processes already appearing at low energy scales. At the high energies of the LHC, these decays are abundant. When looking for rare decays, these common processes create a background. To examine rare decay processes, it is therefore essential to understand the backgrounds caused by the, significantly more abundant, common decays.

To solve the problem of the background being far larger than the signal, (Monte Carlo) simulations are used (e.g. EvtGen, GEANT) [5][6]. These simulation can provide extensive and accurate predictions of the expected signals and backgrounds. However, due to this very precise computations, these simulations can take several weeks to complete. For the accuracy provided, this is an acceptable computation time. However, for exploration purposes, a fast tool is needed. Such a tool can be used to gain a quick insight in the expected backgrounds for given search parameters. Based on the results of such a study, a choice can be made whether it is desirable to start an accurate, computationally intensive simulation.

This thesis describes the creation of a fast tool to predict backgrounds in two body decay processes. This tool will provide the location of all backgrounds, given a specific (BSM) decay

process. To find all backgrounds, firstly, the origin of rare decays and their expected backgrounds have to be examined. This is done in chapter 2. Secondly, a complete particle database is required. The construction of such a database is described in chapter 3. Lastly the computation of the two types of background, called peaking and semi peaking backgrounds, is described in chapters 4 and 5, respectively.

Chapter 2

Rare decays

To predict backgrounds for rare two body decays, it is essential to understand the origin and to know the expected background types. This chapter will describe what is expected when examining rare decays and their backgrounds.

2.1 Theory

Conservation laws in the SM are related to their underlying symmetries. Examples of conserved quantities include energy, momentum and charge. When respected, such symmetries forbid certain processes. However, through loop diagrams, otherwise forbidden processes might occur, albeit at highly suppressed rates. Two examples of such processes are B_s^0 to $\mu\mu$ and D^0 to $e\mu$. The dominating loops needed for these processes to take place are shown in figure 2.1. It should be noted that, indeed, these loops heavily suppress the occurrence of these processes. The branching ratio for the helicity suppressed decay B_s^0 to $\mu\mu$ is $3.57{\cdot}10^{-9}$ [7]. The lepton flavor conservation violating decay D^0 to $e\mu$ will need a neutrino oscillation to occur. In the SM, this gives a branching ratio not feasible by any experiment [8] [9]. Because these processes are so heavily suppressed by the SM, they are very sensitive to BSM processes [10]. Therefore, these rare decays are an active field of research at many experiments.

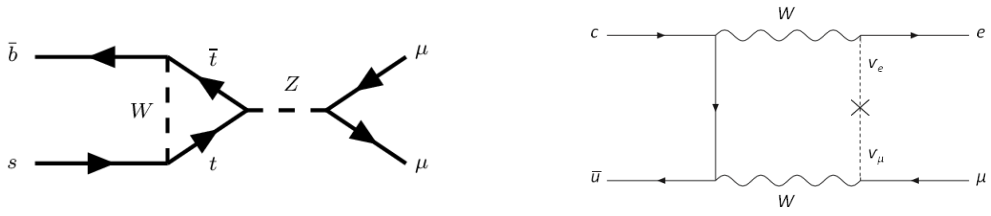


Figure 2.1: Leading order SM diagrams for the decays B_s^0 to $\mu\mu$ [11] (left) and D^0 to $e\mu$ (right)

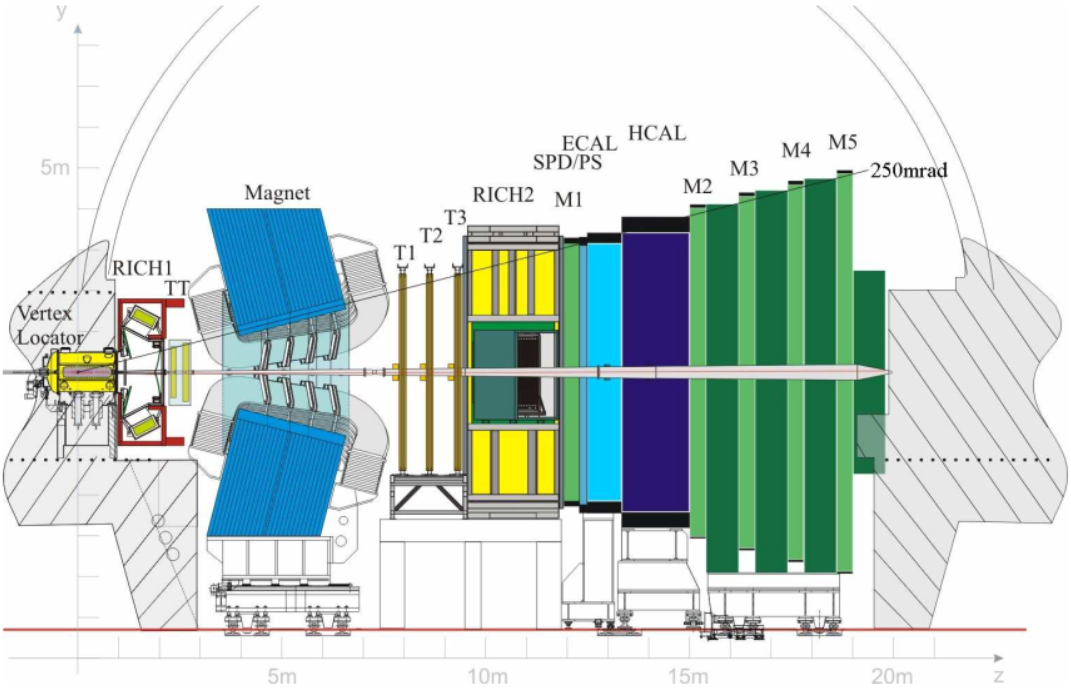


Figure 2.2: Layout of the LHCb detector [12]

2.2 Detection and analysis

Unless states otherwise, the information on the LHCb detector in this chapter comes from reference [12].

To understand the backgrounds of rare two body decays, it is essential to understand the detection of the rare decay. The rare decay that is being searched for will be called the signal decay hereafter. Each signal decay consists of a mother particle decaying into two daughter particles. As mentioned in section 2.1, such a signal decay might be forbidden by the standard model. Although this thesis described a general program, the LHCb detector will be used in this chapter to illustrate the detection of a two body decay. The layout of this detector is given in figure 2.2. Detection consists of identifying the daughter particles and reconstructing the mother particle.

2.2.1 Momentum Reconstruction

In the case of the LHCb detector, the momentum is reconstructed using hits in the Vertex Locator (VELO), Trigger Tracker (TT) and the trackers (T1-3). As seen in figure 2.2, the VELO and TT are located before the magnet, whereas the T1-3 are located after the magnet. Any charged particle passing these detectors will leave hits. By analyzing these hits, the trajectory of the particle and its bending in the magnet can be established. From the bending, the momentum and charge can be

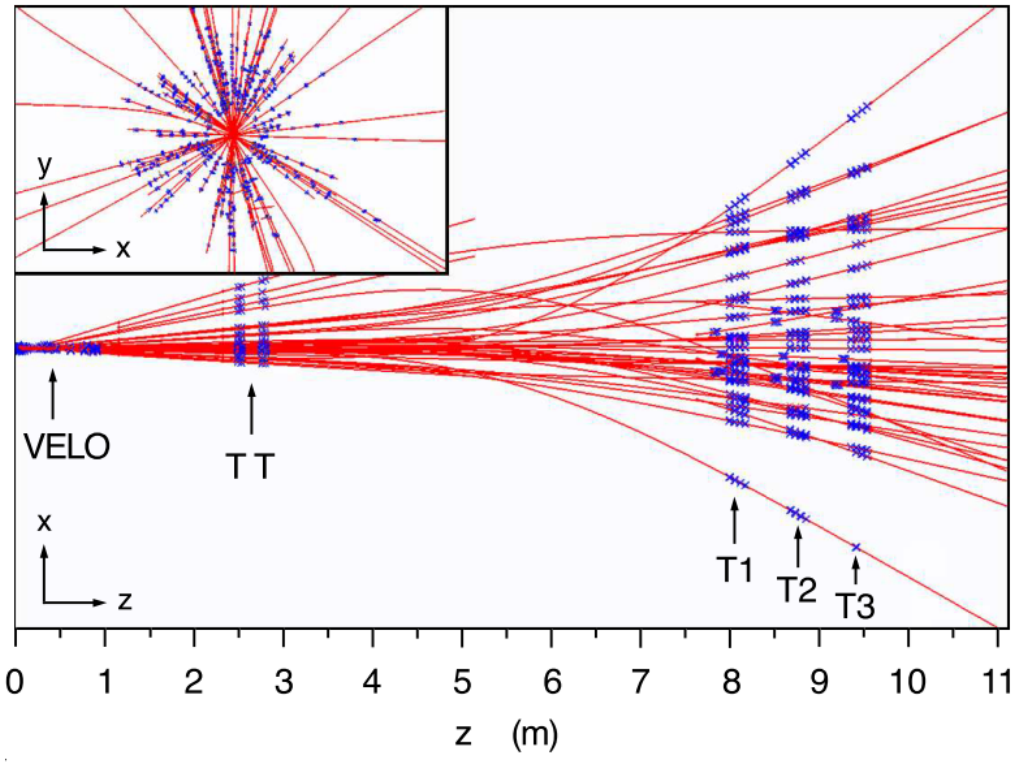


Figure 2.3: An illustration of reconstructed hits in an event. In the insert, the VELO tracks in the x, y plane are shown. [12]

obtained. For each particle decay, it is tested whether the tracks of the daughter particles originate in the same point. This point corresponds to the position where the mother particle has decayed, the decay vertex. Figure 2.3 shows several tracks reconstructed in a single collision.

Bremsstrahlung For electron tracks, an additional effect has to be taken into account. In the presence of matter, electrons experience a process called Bremsstrahlung. In this process, electrons traveling through material loose part of their energy by emitting photons. An illustration of this process is given in figure 2.4. Here, photons emitted in the magnet will go straightforward, whereas the electron track will be bend in the magnetic field. The energy of these photons (E1) will be missing from the electron energy. Photons emitted after the magnet will end in the same position as the electron. The energy of these photons is therefore measured together with the electron energy (E2).

2.2.2 Select particle candidates

Particle identification is done using the Ring Imaging Cherenkov (RICH) detectors, the electron and hadron calorimeters (ECAL, HCAL) and the muon system (M1-5). The RICH detector is

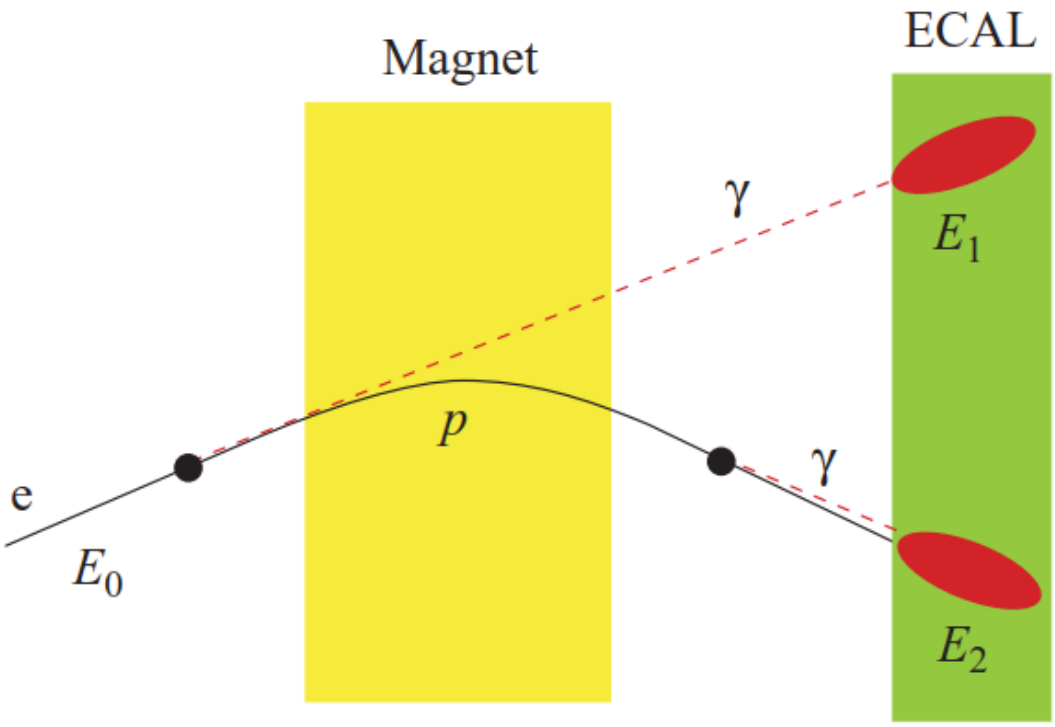


Figure 2.4: An illustration of Bremsstrahlung. [12]

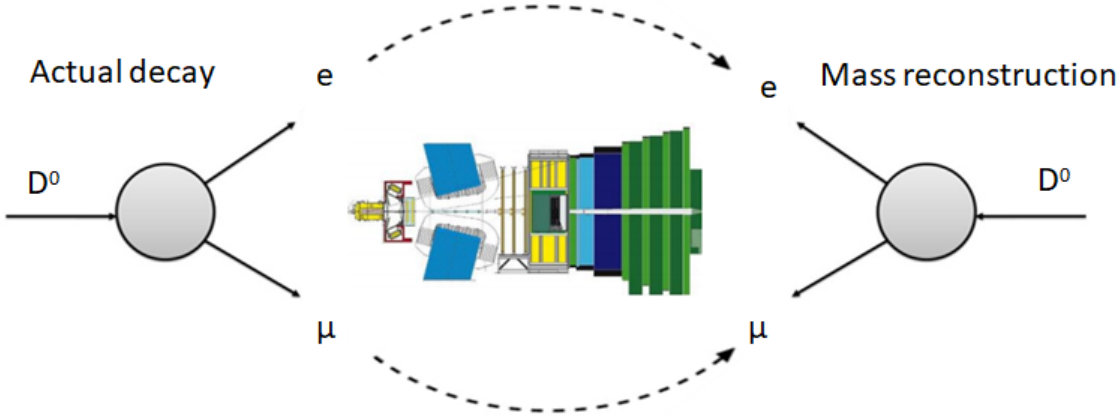


Figure 2.5: A D^0 decaying into e and μ, detected by the LHCb detector. On the left the actual, physical, decay is shown. On the right the decay is reconstructed using the mass and momentum of the particle candidates. LHCb detector copied from [12]

able to identify particles by the shape of their Cherenkov cones. However, this does not provide adequate information to identify particles with high certainty. As can be sen in figure 2.2, the first RICH detector is located at about 1 meter from the pp collision. This means only particles with a lifetime of several nanoseconds can be detected directly. For the HCAL this means charged pions and koans, as well as protons and neutrons [13] can be detected. These particles are detected by the shower they leave behind as they deposit their energy in the calorimeter. The ECAL is used to identify (anti-)electrons and photons. As was already illustrated in figure 2.4, both particles leave a similar mark in the ECAL. However, since photons are neutral and electrons charged, the curvature of their tracks in the magnet can be used to easily distinguish between these particle. Muons are identified in the muon system. A particle is considered a candidate if it leaves hits in a minimal number of stations. This number is between 2 and 4, depending on momentum.

2.2.3 Reconstruct particle decay

A combination of daughter particle candidates whose momenta point to a common vertex is considered a candidate decay. The detection of such a process is illustrated in figure 2.5. The mass of a particle is assigned based on its identification. The last selection on candidate decay is made using the invariant mass of the process. From the particle identification and momentum reconstruction, the invariant mass can be reconstructed. This can be done using equation 2.1, which gives the invariant mass in the center of mass frame of the mother particle.

$$W_{CM}^2 = E_{tot}^2 - p_{tot}^2 \tag{2.1}$$

Here, W_{CM} is the reconstructed mass in the center of mass frame and E_{tot} and p_{tot} are the total energy and momentum of the daughter particles. For two body decay the momenta of the daughter particles will be equal but opposite. Therefore, the total momentum will be 0. The invariant mass will then be given by the sum of the energies of the daughter particles, as shown in equation 2.2.

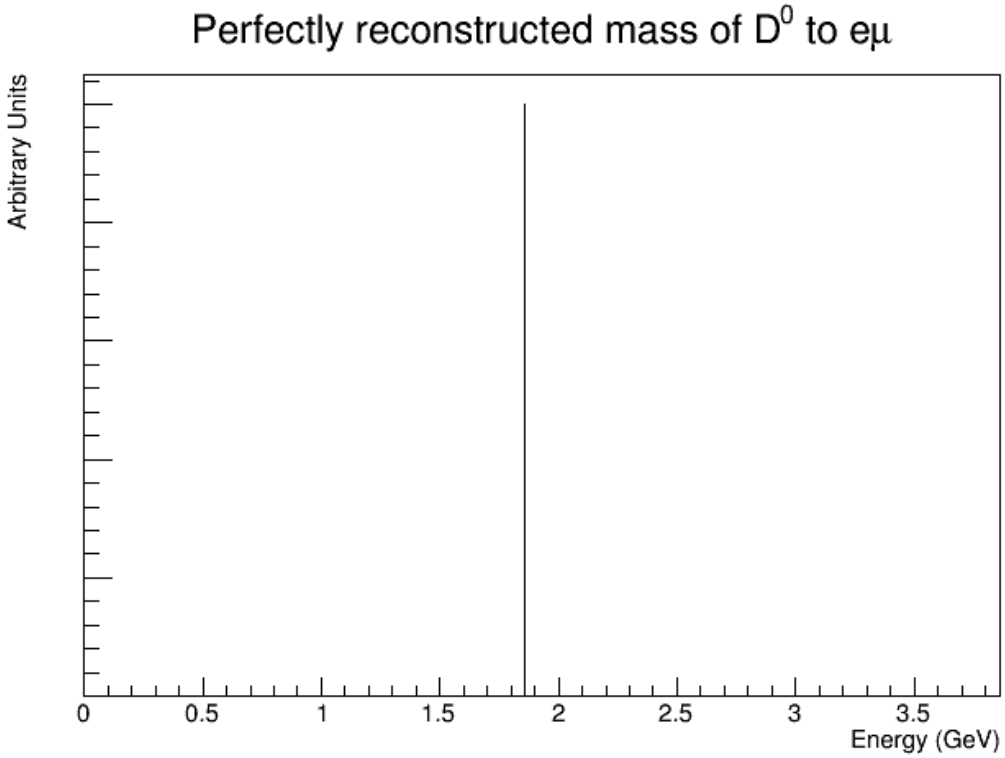


Figure 2.6: A histogram of the perfectly reconstructed mass spectrum of D^0, which has a mass of 1.86 GeV. In a perfect world, without measurement error or background processes, the reconstructed mass will always be a δ function, located at the exact mass of the mother particle.

$$W_{CM} = E_1 + E_2 = \sqrt{m_1^2 + p_1^2} + \sqrt{m_2^2 + p_2^2} \tag{2.2}$$

Here, W_{CM} is the reconstructed mass in the center of mass frame and $E_{1,2}$, $m_{1,2}$ and $p_{1,2}$ are the energy, mass and momentum of the two particles, respectively.

The invariant mass is used to determine whether a candidate decay originates from the mother particle that is being searched for. In the case of the D^0 decay, this would mean selecting decays which have a reconstructed mass at the mass of the D^0 meson (1.86 GeV). In a perfect world, where measurement errors do not exist, all candidate decays would reconstruct to the D^0 mass. If the reconstructed mass spectrum is then plotted, it would look like a δ function, as shown in figure 2.6. However, in the real world, there are processes, including scattering and detector resolution, which lead to a broadening of this function. Furthermore, incorrect combination can mimic the candidate decay. Section 2.3 describes several types of such backgrounds.

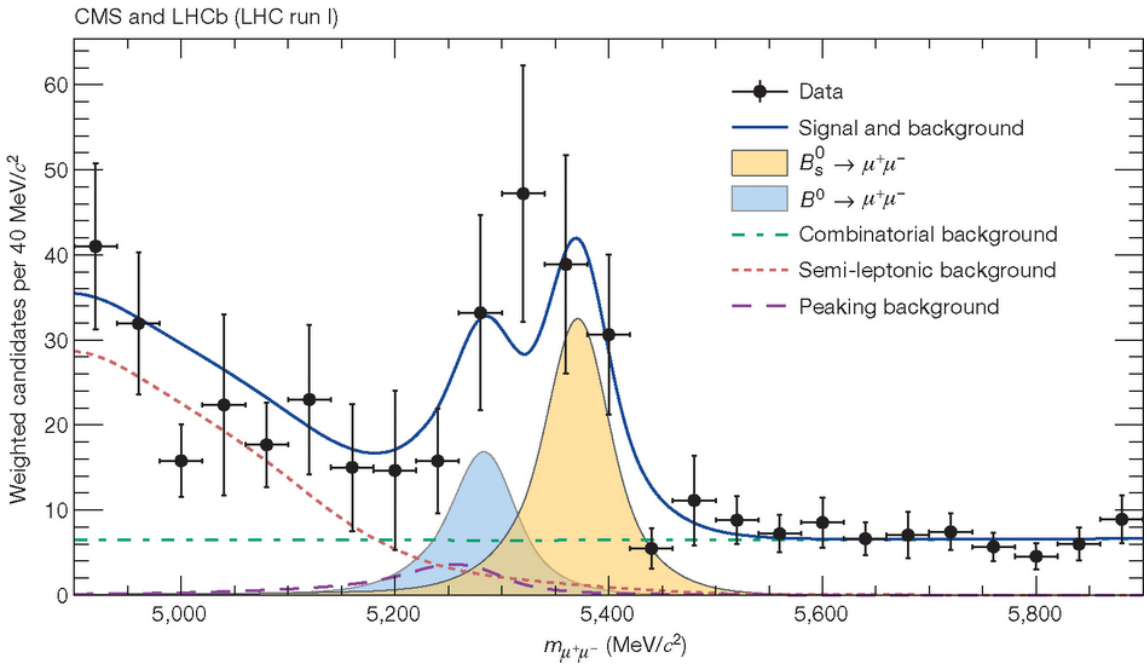


Figure 2.7: The dimuon invariant mass spectrum of $B^0_{(s)}$ to $\mu^+\mu^-$ showing combinatorial, peaking and semi peaking backgrounds [14]

2.3 Types of backgrounds

In figure 2.7 an analysis of the decay $B^0_{(s)}$ to $\mu^+\mu^-$ is shown. In this picture, 3 types of background are identified, these are the combinatorial, peaking and semi-leptonic backgrounds. To locate such backgrounds in a general decay, it is essential to understand the origin of these backgrounds.

2.3.1 Combinatorial

It can always happen that two unrelated daughter particles originate in a common vertex. These particles do not have to be from the same decay, just by chance they end up with their tracks pointing to the same vertex. As there is no relation between these particles, their invariant mass can have any value. This leads to a smooth background all through the spectrum. This background is called the combinatorial background, highlighted in green in figure 2.7. Because of the smoothness of this background, it is easy to parametrize. Therefore, the simulation of these backgrounds will not be part of this project.

2.3.2 Peaking

Another type of background observed in 2.7 is the peaking background. This naming is because these backgrounds resemble the peaks of the signal, in this case the $B^0_{(s)}$ to $\mu^+\mu^-$ peaks. Already in this figure, two peaks are shown, representing the decays B^0_s to $\mu^+\mu^-$ and B^0 to $\mu^+\mu^-$. When

looking for one of these processes, the other will constitute a background. Other peaking background arise through misidentification of particles. A misidentification can happen when a particle candidate is wrongly identified as one of the signal daughter particles. As mentioned, the mass is assigned to particles based on their identification. When a misidentification takes place, the wrong mass is assigned to a particle. For example, if a charged pion would leave enough hits in the muon system (see section 2.2.2), it would be identified as a muon. Therefore, the pion, with a mass of 140 MeV/c^2 would be assigned a mass of 106 MeV/c^2. Although such misidentifications have a small chance of occurring, it is important to remember that this project concerns rare decays. Therefore, common processes, happening in abundance, might yield significant hits, despite the small change of misidentification.

When the wrong mass is assigned to a particle, the assumption that equation 2.2 is the invariant mass in the center of mass frame is no longer correct, as both the boost and the mass(es) are wrong. Because of the misidentification, some energy is missing or incorrectly added to the invariant mass. As a consequence, the invariant mass will be wrong, as shown in equation 2.3.

$$W_{CM} = \sqrt{m_{1,mis}^2 + p_1^2} + \sqrt{m_{2,mis}^2 + p_2^2} \tag{2.3}$$

As the momentum is correctly measured, the previous assumption that the total momentum from equation 2.1 is 0 is still valid. The new value for the reconstructed mass will give the position of the peak of the peaking background. The function finding these peaking backgrounds is given in chapter 4.

2.3.3 Semi peaking

The last type of background arises when events are partially reconstructed. It happens that the two daughter particles are created in a decay which also creates other particles. If these other particles are not detected, this would resemble a decay to just the two daughter particles. When reconstructing such an event, the invariant mass of the two daughter particles is missing the energy taken by the undetected particles. Such events are given by the red line in figure 2.7. Here these events are named semi-leptonic, as they originate partially reconstructed decays which have muons, the signal daughter particles in this specific case. In general these backgrounds will be referred to as semi-peaking backgrounds. The function finding such semi-peaking backgrounds is given in chapter 5.

2.4 Locating the backgrounds

As mentioned, in the search for rare decays it is likely to encounter backgrounds much larger than the possible signal. Therefore it is essential to know beforehand what backgrounds to expect in the region of invariant mass that is being probed. The aim of this project is to make a fast program to predict which backgrounds are to be expected in a defined search region. Such a fast program will allow for quick searches to locate clean signal regions. An example of such a region is given in figure 2.8. Here the signal of J/Ψ to $\mu^+\mu^-$ is shown. As can be seen, only a single peak is visible, which can be associated with the signal.

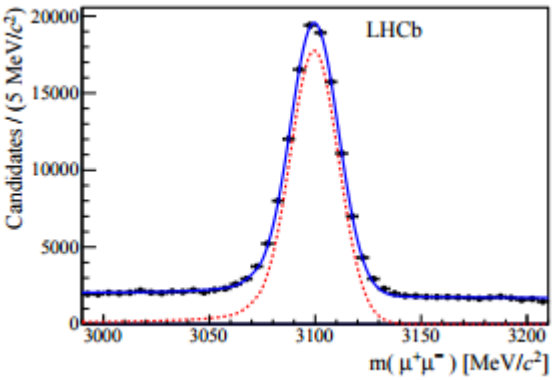


Figure 2.8: Example of a clean region in J/Ψ to $\mu\mu$ [15]

For a candidate process to be considered a background it is required to have significant probability of occurring. BSM processes are not expected to be abundant, as no such process has been detected (yet). Therefore, only SM allowed backgrounds are considered prominent enough to possibly appear as a background. If a BSM processes were to turn up as a measurement background, it would be reason to change the focus of the experiment and treat it as a signal. To determine whether a process is SM allowed the Particle Data Group(PDG) database is used [16]. It is assumed that a particle decay that has been measured can be found in the database. Therefore, any expected background can be found given a complete database.

To have the most complete collection of backgrounds in a given region of invariant mass, it is necessary to have the most complete database of particles and decay channels possible. Unfortunately, it is not possible to probe the PDG database directly. Instead, several open source version are present in specialized software packages. These are the ROOT [17], Pythia [18] and EvtGen databases [19]. To have the most complete database, it was decided to merge these three databases. This is described in chapter 3.

Chapter 4 will explain how to locate all peaking backgrounds, using this database. This is done by looking at all combinations of particles forming a two body decay. These combinations are constrained using the laws of physics and the particle properties from the database.

Finding the last background category, the semi peaking backgrounds, is described in chapter 5. This function locates both direct and cascade decays, contributing to partially reconstructed events.

The code of all the programs used in this project will be listed in appendix A. The start of this appendix shows a flowchart (figure A.1) of the programs used in the specific parts of the projects: database building, finding peaking backgrounds and finding peaking backgrounds.

Chapter 3

Database building

To find all possible backgrounds, it is essential to have a complete and detailed database of particles, their properties and decay channels. This chapter will describe the building of such a database. As mentioned, this will consist of merging three databases (ROOT, Pythia and EvtGen), all of which are based on the PDG database.

3.1 Requirements

All of the databases are structured as follows: there are particles with a given name and a unique ID, referred to as the pdg code of the particle. In addition there are particle properties, including, but not limited to, mass, lifetime and charge. Each particle can also have a corresponding list of decays, giving the decay channels in which the particle is allowed to decay. Each of these channels also holds properties, including, but not limited to, the branching ratio of the channel.

The requirements for the combined database are dictated by the conditions used to find peaking and semi-peaking backgrounds. To determine whether a given peaking or semi peaking background is to be expected the physics properties of the given particles listed in table 3.1 are used. For the entire process see chapter 4 on peaking backgrounds.

Table 3.1: Required particle properties

Property
Pdg code (unique ID)
Name
Mass
Charge
Lifetime
Spin
Color type (i.e. singlet, multiplet)

In addition it is important to know whether a particle is in the SM. For the semi peaking background not the list of particles, but the list of channels of each particle is used. The process of

finding these backgrounds in described in chapter 5. To compute these background, a complete set of decay channels for each particle is needed. In addition, the branching ratio of each background is required.

3.2 Available databases

Three open source version of the PDG database are used to create an as complete as possible database: ROOT, Pythia and EvtGen databases.

3.2.1 ROOT

CERN offers a software package called ROOT [17] with contains a subset of the PDG database. For this project ROOT version 5.34/36 has been used. The database is included in the file "pdg_table.txt" and can be access through the TDatabasePDG Class [20]. Particles are then accessed through the TParticlePDG class [21]. Each particle can have certain properties assigned to it. Table 3.2 is a list of all possible properties and whether or not they are used in the project. After this table follows some notes on several properties in the database.

Table 3.2: Used and Unused properties from the ROOT database

Used Properties	Unused properties
pdg code	Parity
Name	Isospin
Mass	I3
Charge	Strangeness
Lifetime (see lifetime note)	Charm
Width (see lifetime note)	Beauty
Spin	Top
List of decay modes	4^{th} generation quantum numbers (X,Y)
Class (lepton, etc. See class note)	Stability (see lifetime note)
	G3 Tracking Code

Notes on lifetimes The counting of particles in this section is done using the program `ROOT-count` given in appendix A.1. Lifetimes in ROOT are determined by three functions in the TParticlePDG class: lifetime, width and stable. Here lifetime is in seconds, width in GeV and stable can be false or true. With these three it should be possible to determine the lifetime of every particle. Giving particles a lifetime, or a width in case of resonances, and a stable to true in case of stable particles, such as electrons or protons. Unfortunately, the database is not complete on this point. In total there are 521 particles in the ROOT database. Of these 521, 303 particles do not have a lifetime defined (58%). All particles without a lifetime have their stable modifier set to true. There are no particles in the ROOT database that have a width and no lifetime or a lifetime and no width. Some examples of particles which are used in LHCb searches, without lifetime and width in ROOT are: *electron*, *tauon*, *proton*, D^0, J/Ψ. Some of these particles are stable, and cannot be assigned a lifetime, while other are unstable, and should have a lifetime. Because so many lifetimes are missing, the lifetimes in ROOT are not considered complete.

Notes on class The property ParticleClass of the ROOT particles contains the type of the particle. The possible classes are shown in table 3.3 with some examples of each class. Every ROOT particle has one of these classes.

Table 3.3: Particle classes in ROOT

Class:	Examples:
Generator	Specflav, b-hadron, cluster
Meson	π^0, ρ^0, π_diff+, K^0, J/Ψ
B-Meson	B^0, $B^{+/-}$, B_s, Υ
CharmedMeson	D^0, $D^{+/-}$, D^s, J/Ψ_diffr
Baryon	$Neutron$, Δ^+, Λ, Σ
B-Baryon	Σ_b, Λ_b
CharmedBaryon	Σ_c, Λ_c
Quark	u, d, $\bar{u}$, $\bar{d}$
Lepton	e, μ, τ, τ', $\nu_{e,\mu,\tau}$, ν'_τ
GaugeBoson	γ, Z0, Z'0, Z"0
Excited	d*, e*, μ*,τ*
Unknown	dd_1, LQ_ue, Δ^-, Υ', D^{*0}
Sparticle	∼d_L, ∼e_l-, ∼g

It has been assumed that no BSM backgrounds are to be expected. The ROOT particle class can help to identify particles which do not occur in the SM. However, there are several inconsistencies within the particle classes.

Firstly, the quarkonium states. One could argue that a state like $J/\Psi(c\bar{c})$ is either a charmed meson, for its c quarks, or a meson, as the "charmness" of the two quarks cancels out. It is named a meson in the ROOT database. However, the $\Upsilon(b\bar{b})$ is inconsistently called a B-Meson, despite its "beautyness" cancelling out. The J/Ψ_diffr, of which it is not clear what it is, on the other hand, is called a charmed meson. And the Υ' ($\Upsilon(2S)$), an excited state of the Υ, is listed as unknown. Another "diffr" particle, the π_diffr+, is than in the meson category, akin to the actual π meson.

Secondly, there are inconsistencies in the Δ baryon classes. The $\Delta^+(uud)$ is listed as a baryon. However, the $\Delta^-(udd)$ is listed in the unknown category. In this unknown category, there is also the BSM particle LQ_ue, a leptoquark connecting electrons and u-quarks.

In the gauge boson category there are both SM and BSM particles. All the SM gauge bosons, for example the photon (γ) and Z0 are present in this category. In addition also BSM gauge bosons are present in this category, for example the Z'0 and Z"0 bosons.

Lepton number As mentioned in the begin of this chapter, the function seeking peaking backgrounds used lepton number to check if a candidate background is SM allowed. For more information on this function see chapter 4. As lepton number is not a particle property in the ROOT database (or any of the other databases) the ROOT class lepton is used. In table 3.3 all particle of the lepton class are shown. All particles in this class are SM particles, except for τ' and ν'_τ. These

particles seem to correspond to potential 4^{th} generation leptons. Later on, the τ' and ν'_τ are taken out manually (see section 3.3 on combining databases). This is done by giving these a new class "Unscope" to show they are beyond the scope of the function.

Eventually, it was decided that only the generator, Excited and Sparticle classes will be used to pinpoint BSM particles and pseudo particles. These classes do not contain any SM particles. In addition to the aforementioned lepton class, also the quark class is used. This only contains the quarks from the SM, and is used to ensure there are no single quarks in the final state. All other classes are deemed too inconsistent to be used.

Missing particles In addition to the particle properties, it is also important to check if there are any particles which are not present at all in the database. Unfortunately there is no unbiased way to find what is not in the database. Particles that are not in the database are only encountered when they are searched for. Missing particles that were found in some tests of the program were $\Upsilon(3S)$ and $\Upsilon(4S)$.

3.2.2 Pythia

Initially, the Pythia database was used to obtain the lifetimes of the particles which do not have a lifetime in the ROOT database. The Pythia program [18] is used in the ROOT framework for dynamic particles and has its own particle database. When accessing lifetimes in ROOT it is suggested to use the Pythia decay program [22]. As only the database was required, the ParticleData file was taken from GitHub [23]. The database is structured in XML format. As with the ROOT database, each particle has a list of decay channels assigned to it. Each particle can also have any number of the properties listed in table 3.4 [24]. In this table, BW stands for Breit-Wigner distribution.

Table 3.4: Used and Unused properties from the Pythia database

Used Properties	Unused properties
pdg code	mMin: lower limit of the BW
Name	mMax: upper limit of the BW
Mass in GeV	
Charge in units of $\frac{1}{3}e$	
Lifetime in mm/c	
Width in GeV	
Spin type (2S+1, in $\hbar$)	
Color type (i.e. singlet, multiplet)	

There are also several particle flags are mentioned in [24], but not found in the database found in [23]. In addition many particles have a property named antiName, which simply gives the name for the antiparticle, for example the proton has name p+ and antiName pbar-, corresponding to an anti-proton. This way particle and antiparticle are a single entry in the Pythia database. Also, the channels in the Pythia database have some properties, i.e. onMode and meMode, which are associated with the Pythia program itself. These properties are unused. What is used, are the branching ratios and the products of each channel.

To read the XML format of the Pythia database within the ROOT framework the TXMLEngine class is used [25]. An example of this use can be seen in appendix A.2, which contains a program to check the completeness of the Pythia database in terms of particle lifetimes. In total there are 345 particles in the Pythia database. This cannot be compared with the 521 in the ROOT database, as in the ROOT database particle and antiparticle are counted separately. Of the 345 Pythia particles, 289 do not have a lifetime defined (84%). However, in the Pythia database resonances do not have a lifetime, but a width instead. If only particles are counted which have neither a width nor a width, the result is 101 particles (29%). This is still too much to only contain stable particles. However, it is an improvement over the ROOT database, which was missing 58% of the lifetimes. It should also be noted that there are many BSM particles in the database which do not have a lifetime. Some unstable particles, encountered in the making of this project, with missing lifetimes in both the ROOT and Pythia database include: η, Σ_b^0 and B^{*0}.

While testing for peaking backgrounds in Υ decays, it was found that in the Pythia database there are no Υ(3S) and Υ(4S). These particles were also missing in the ROOT database. Whether more particles are missing from these databases is not known. In terms of completeness the Pythia is a essential addition to the ROOT database, especially when it comes to lifetimes. For this project these databases work to find all peaking background given a certain decay. However, for finding semi-peaking backgrounds, see section 5, not only the particles themselves, but mainly the channels are used. To have a complete set of channels one more database will have to be added.

3.2.3 EvtGen

The EvtGen program is specialized in B-decays [19]. Therefore, many channels and properties of B particle physics can be accessed through this database. The databases from EvtGen version R01-06-00 [26] are used. To omit having to install the EvtGen program in its entirety, only two files from the source tarrball are used. The first is the evt.pdl file. This file contains all the particles from the EvtGen database. The second file is the DECAY_2010.XML file. This is a file that has been converted to XML format from the DECAY.DEC file normally used in the EvtGen program. In this decay file there can be separate channels for particle and antiparticle. The properties from the evt particle file used by this program are listed in table 3.5

Table 3.5: Used properties from the EvtGen database

Property
pdg code
Name
Mass in GeV
Charge in $\frac{1}{3}e$
Lifetime in mm/c
Width in GeV
Spin in $\frac{1}{2}\hbar$
Pythia ID: gives the corresponding ID in the Pythia database

Only one property was not used: the maximum width of the particle. For the channels in the decay file more properties than in the previous databases are available. These include sources for

each channel, the branching ratio, the names of the daughter particles and several parameters for the EvtGen program. The availability of the decay channels in the EvtGen database were the main reason to include it. These are more complete than the other databases. This is most noticeable when looking at the number of channels for B-particles. A comparison is shown in table 3.6.

Table 3.6: The number of decay channels for several B-particles in each database

Particle	ROOT Database	Pythia Database	EvtGen Database
B^0	37	891	962
B_s^0	40	247	257
B^+	37	736	794

From this table it can be concluded that the ROOT database should not be used to access B-particle channels. The difference between the Pythia and EvtGen databases is not as large as the difference of these two databases with respect to the ROOT database. However, in every case the EvtGen database has more entries than the Pythia database. Therefore the EvtGen database is taken to be the most complete in terms of particle channel listings. For the ROOT and Pythia databases the Υ(3S) and Υ(4S) were taken as cases to check whether all particles are present. Both these particles were not present in these databases. However, these are in the EvtGen database. Therefore, the EvtGen is taken to be the most complete in term of particles.

For many of the properties that are not used from the ROOT, Pythia and EvtGen databases holds that these are unique to the respective database. However, for each property holds that it can be added if so desired. The property will then only be added for particles that are also in the database from which the property was taken. For particles from the other databases, it will have to be added manually. One example of such a particle property that is used in the program is the ROOT ParticleClass. In the following chapter the properties that are used in the program will be combined into one database.

3.3 Combining databases

To combine the databases, first of all, all three databases have to be in common format. To achieve this, the databases will be converted into a c++ map [27]. This map will use the unique id of each particle, the pdg code. This pdg code will be mapped to a particle struct, which will have to be made. Then there will be three maps, one for each database. These will be combined by looping over each map and adding each particle, channel and property to a combined map.

The particles in the map are given by the newly created particle struct. The code for this struct is listed in appendix A.3. An particle object made with this struct will have the properties listed in table 3.7.

The decay channels of the particles are listed as a vector of a newly made channel struct, which is also given in appendix A.3. One single channel contains the properties given in table 3.8. The flag in both the particles and the channels is 0 at default, but can be set to 1 for objects outside the scope of this project. It should be notes that each daughter particle of a given channel object is again a particle object, containing its own set of channels.

Table 3.7: Properties of a particle object

Property	Description
Pdg code	A unique ID
Name	
Mass	Given in GeV
Charge	Given in $\frac{1}{3}$ e, to make it an integer number
Spin	Given in $\hbar$
Particle type	Based on the ROOT ParticleClass property
Color type	For example: singlet, triplet
A scope flag	Can be either 0 or 1
Channels	A list of the particle's decay channels

Table 3.8: Properties of a channel object

Property	Description
Branching ratio	
Mother particle	Pointer back to the mother particle of the channel
Daughter particles	A list of daughter particle objects
A scope flag	Can be either 0 or 1

In addition to the map from id to particle, a second map is made. This map maps the particle names to their respective id's. This is done because the names of the particles can appear different depending on the database that is being used. One example of such an occurrence is the $\bar{K}^{*0}$. In the EvtGen database this is called "anti-K*0". in the Pythia database however, it is named K*bar0. In the ROOT database, it again has a different name: K*0_bar. Since the names of particles can be used as input for the program of this project, all 3 versions of the name are mapped to the pdg code of $\bar{K}^{*0}$. For both maps see appendix A.4.

3.3.1 Mapping the ROOT database

The function mapping the ROOT particles to the standard library map is given in appendix A.5. The function works as follows. First the database that is in ROOT is to be read into the program, this is done using the TDatabasePDG class. The structure of the database can be seen in 3.1. This database is then converted into a list of all the particles. With the database as a list of particles and an iterator for this specific list it is then possible to loop over the list of particles. This way each particle can be accessed one at a time. The particles in the ROOT database are given in the TParticlePDG class. This will have to be converted to newly created FParticle struct (appendix A.3) to be able to compare the ROOT database with the other two databases. This chapter will go through each of the functions used in the function to map ROOT particles.

Adding particles to the ROOT map To convert a ROOT particle into the new particle the function `PopulateParticleROOT` is used. In this function any properties from the ROOT database, as described in chapter 3.2.1 can be converted into new particle properties. Only the properties that are used in the rest of this project are transferred from one particle to the other.

```
#-------------------------------------------------------------------
#    i   NAME.............  KF AP   CLASS      Q        MASS     WIDTH  2*I+1 I3 2*S+1 FLVR TrkCod N(dec)
#-------------------------------------------------------------------
   1 d                    1  1 100 Quark      -1 4.80000e-03 0.00000e+00 -100 -1 -100 -1   -1   8
#-------------------------------------------------------------------
#    decay  type(PY6)    BR     Nd         daughters(codes, then names)
#-------------------------------------------------------------------
         1   102   0.00000e+00   2       21        1                                    g         d
         2   102   0.00000e+00   2       22        1                                    gamma     d
         3   102   0.00000e+00   2       23        1                                    Z0        d
         4   102   0.00000e+00   2      -24        2                                    W-        u
         5   102   0.00000e+00   2      -24        4                                    W-        c
         6   102   0.00000e+00   2      -24        6                                    W-        t
         7   102   0.00000e+00   2      -24        8                                    W-        t'
         8   102   0.00000e+00   2       25        1                                    h0        d
   2 d_bar              -1       1     0
   3 u                    2  1 100 Quark       2 2.40000e-03 0.00000e+00 -100 -1 -100 -1   -1   8
#-------------------------------------------------------------------
#    decay  type(PY6)    BR     Nd         daughters(codes, then names)
#-------------------------------------------------------------------
         1   102   0.00000e+00   2       21        2                                    g         u
         2   102   0.00000e+00   2       22        2                                    gamma     u
         3   102   0.00000e+00   2       23        2                                    Z0        u
         4   102   0.00000e+00   2       24        1                                    W+        d
         5   102   0.00000e+00   2       24        3                                    W+        s
         6   102   0.00000e+00   2       24        5                                    W+        b
         7   102   0.00000e+00   2       24        7                                    W+        b'
         8   102   0.00000e+00   2       25        2                                    h0        u
   4 u_bar              -2       3     0
```

Figure 3.1: The first few entries in the ROOT database. There are the top and down quark and their anti-particles. This illustrates the structure of the database.

However, by changing this function and the particle struct one can add any desired property. The properties that are taken over directly from the ROOT database are: the pdg code, the particle mass, the particle spin. The ParticleClass from the ROOT database is renamed particle type. The charge in ROOT is already given in integer units of $\frac{1}{3}$ e, as desired by the new particle struct. As there are no particles other than color singlets in the ROOT database, the color type of the ROOT particles is always set to the singlet value 0.

Checking ROOT lifetimes For the lifetimes the function `FindROOTLifetime` is used. This function first checks if the particle has a lifetime. If there is a lifetime, it is assigned to the new particle. If there is no lifetime, the function seeks for a width. as mentioned in chapter 3.2.1 there are no particles in the ROOT database which have a width and no lifetime. This check is therefore done to completely include the ROOT functionality, with possible future width added to the ROOT database.

ROOT names A separate function, named `name_handler_ROOT`, is used for the names in the ROOT database. Because of technical reasons, the name $Z"0$ (with 1 " character) was changed to $Z''0$ (with 2 $'$ characters). One format in which the new database will be rewritten (see chapter 3.4.1) is XML. In XML format, the " character is used to localize text. Therefore, "$Z"0$" breaks the structure of an XML file. For the entire process of writing the rewritten database, see chapter 3.4. This function also checks if the word "tech" is present in the name. This is to take out any Technicolor [28] entries in the database. These entries are listed with the type "Unknown" by default. As discussed in chapter 3.2.1 the "Unknown" type does not contain a consistent set of

particles. Therefore, particles with "tech" in the name are reassigned the type "Technicolor".

Once each particle has all the desired properties assigned to it, it is inserted into the map. This map links the pdg code of the particle to the particle itself. As discussed earlier in this chapter, a second map is made alongside the particle map. This map pairs the names of the particles with their pdg code. This will be used when combining all three databases. For the merging of the databases, see chapter 3.3.4.

Add channels to ROOT particle The map from pdg code to particle is then used to fill the decay channels of each particle. This is done in the `fillChannels` function. This function loops over every particle in the map that was made and finds back its partner from the ROOT database. The ROOT particle is used to find the channels of the new particle. It would be ideal to not have to loop over all the particles twice, once for filling the particles and once for filing the channels. However, the channels are given with a list of daughters, which have to be mapped again to their corresponding particle struct in the map. These daughters are accessed by their pdg codes in the ROOT program. Therefore it is necessary to first have every particle within the map, before going to fill the channels.

For each particle in the map, the channels are filled using the function `findROOTChannels`. This function accesses the ROOT decay channels. These are stored in the TDecayChannel of the TParticlePDG class. A ROOT channel is then converted to the new channel struct, as given in appendix A.3. For every channel the mother particle and branching ratio can be added directly. For the list of daughter particles, a new function, `fillROOTDaughters`, is called. The function loops over the daughters of the ROOT channel. For each daughter it links the pdg code to the new particle struct via the particle map. To easily compare the various particle channels from the three databases, the daughter particles of each channel are sorted by their pdg codes.

At this point the function has made a map of all the particles in the ROOT database, including their respective channels. With the particles in these channels again being in the map. To compare with the other databases these have to be brought to this format as well.

3.3.2 Mapping the Pythia database

The functions to map the Pythia database are given in appendix A.6. The method used is similar to the mapping of the ROOT database. First all particles are written in terms of the new particle struct. Then a map is created, mapping the pdg code to the particles. This chapter will go over the functions that are used, following the flow of the program.

As mentioned in section 3.2.2, describing the Pythia database, it is written in an XML structure. An example of the database is shown in figure 3.2, showing the particle $pi^{+/-}$. To read this database in the ROOT program, the build in TXMLEngine class in used. This allows for looping over specific parts of the database, called XML nodes. For example, it is possible to loop over all the particles of the database, without having to go over the channels. This is because the channels are in a different node than the particles. To be able to use this ROOT functionality, a slight alteration to the database file was made. The copyright statement on the end of the file was outside of the main node "chapter". Therefore, the file could not be accessed in ROOT. To fix this the last two statements in the file were swapped. This function directly probes the Pythia database file,

discussed in chapter 3.2.2. This database is fed into the function, and a loop over the particle nodes is constructed. For each node, a particle is constructed using the new particle struct.

```
<particle id="211" name="pi+" antiName="pi-" spinType="1" chargeType="3" colType="0"
          m0="0.13957" tau0="7.80450e+03">
 <channel onMode="1" bRatio="0.9998770" products="-13 14"/>
 <channel onMode="1" bRatio="0.0001230" products="-11 12"/>
</particle>
```

Figure 3.2: The charged π meson, as given in the Pythia database. This illustrates the structure of the Pythia database, with particle and channel nodes

Adding particles to the Pythia map To create the particle is the Pythia map the function `PopulateParticlePythia` is used. The properties of each particle node are given to the new particle. The pdg code(id), charge and mass are taken directly out of the database. For the lifetime, name and spin of the particle, additional funtionallity was needed. These are described below. The color type of the particles will be copied from the Pythia database. Non-singlet color particles are BSM and not expected to be a background. These are assigned the type "ColorMultiplet". As the Pythia database does not contain information on the particle type the rest of the particles are give the type "PythiaParticle" by default. The Pythia database has a combination of particle and antiparticle, by assigning the property "AntiName" to each particle which has a corresponding antiparticle. If the particle has this property the function `PopulateParticlePythia` return true. If it does not have an antiparticle, it returns false. In the map particle and antiparticle are separate particles, linked by the fact that their pdg codes have opposite sign. If an antiparticle is found the function `AntiParticlePythia`, described below in the paragraph Pythia antiparticles, is called.

Pyhtia spin The spin in the Pythia database is given as $2S + 1$, with S in units of $\hbar$. However, the particle which is being created, has spin in units of $\hbar$ only. To convert the new spin is defined as $\frac{S_{Pythia}}{2} - 1$. For particles with undefined spin (i.e. BSM and pseudo particles) the Pythia value is set to 0. If such a particle is encountered, the spin of the new particle is also set to 0. To recognize such a particle later, it is also assigned the type "Unscope".

Pythia lifetime and names Two properties, the name and the lifetime of the particles, are treated in separate functions, named `PythiaName` and `PythiaLifetime`. As with the ROOT database, there are Technicolor particles [28] in the Pythia database. These are given the type "Technicolor" by finding if particles have "tc" in their name. In addition, supersymmetric particles are located by looking for "$\sim$" in the name. The lifetimes ($\tilde{\tau}$) in the Pythia database are given in mm, as $c \cdot \tau$. In addition some particles do not have a lifetime, but instead have a width (Γ). For the complete database, these values are taken together in one lifetime property. This new lifetime is given in seconds. For particles which have a lifetime in mm the conversion to seconds is given by:

$$\tau[\mathrm{s}] = \frac{\tilde{\tau}[\mathrm{mm}]}{c} \cdot 10^{-3} \frac{\mathrm{m}}{\mathrm{mm}} \tag{3.1}$$

where c is the speed of light in $\frac{\mathrm{m}}{\mathrm{s}}$. This corresponds to:

$$\tau[\mathrm{s}] = \tilde{\tau}[\mathrm{mm}] \cdot 3.333 \cdot 10^{-12} [\frac{\mathrm{s}}{\mathrm{mm}}] \tag{3.2}$$

to convert lifetimes from mm to seconds. For resonances, not the lifetime, but the width is given to show how long the particle lives. This width is given in units of GeV. To convert from width to lifetime in seconds the following equation is used:

$$\tau[\mathrm{s}] = \frac{\hbar}{\Gamma[\mathrm{GeV}]} \tag{3.3}$$

with $\hbar = 6.582 \cdot 10^{-25}\,\mathrm{GeV} \cdot \mathrm{s}$.

Despite flagging particles with undefined spin, technicolor charge or supersymmetry, there are still BSM and pseudo particles unidentified as such. The last method that was used to identify these particles, is based on pdg code. In the Monte Carlo numbering scheme [29] it can be seen that the only SM allowed particle with 7 digits or more in the pdg code are atoms. Since Pythia does not have any atoms in the database, pdg codes with more than 6 digits are BSM and pseudo particles. It has been checked that the highest SM allowed pdg code is 6 digits. This was found to be $\Upsilon(2S)$, which has a code of 100553.

At this point the properties of the particle node are properly fed to the new particle. The particle is then added to the map connection pdg code and particle. As mentioned in chapter 3.3, a second map is made connecting the name to the pdg code of each particle. This is done as there can be different names for each particle. In the case of the Pythia database, a third map is to be constructed. This map connects a particles pdg code to the XML node in which the particle was found. This map will be used later to find the channels for this particle.

Pythia antiparticles If an antiparticle was found by the function `PopulateParticlePythia`, an antiparticle will have to be add to the particle map. This is done by the function `AntiParticlePythia`. And the name of this antiparticle will have to be added to the map of names. The process of creating an antiparticle is similar to that of creating a particle. The same particle node is used to access the properties of the antiparticle. The mass, lifetime and color type are the same for particle and antiparticle. The spin and charge have opposite sign to the particle. This also holds for the pdg code. The particle that is chosen as antiparticle is given the negative pdg code of the particle to which it corresponds. The name of the antiparticle is given by the Pythia database, as a property called "AntiName". The antiparticle, if a color singlet and not a Technicolor particle, is given the type "PythiaAntiparticle".

Assigning channels to Pythia particles If the map of all particles, including antiparticles is complete, the channels can be constructed. As with the ROOT database, a loop over the entire map is done to revisit all particles and add channels to each. Also with the Pythia database this can only be done afterwards, as the daughter particles of each channel will have to be in the map already to assign them to the channel. In contrast to the ROOT database, an extra map is used, this map gives the corresponding node in the XML file for each particle. Anti particles are in the same node as their corresponding particle. Each particle node contains an underlying

node, which holds the channels for that particle. To add the channels to each particle the function `findPythiaChannels` is used. This function loops over all the channel nodes within a single particle node. For each node a channel is created for the particle. Three properties have to be added to the channel: the mother particles, the branching ratio and the daughter particles. The mother particle is given by the previous function and passed on to this function. The branching ratio can be found directly in the channel node. The products are given as one long name of all the pdg codes, as shown in picture 3.3.

```
<particle id="511" name="B0" antiName="Bbar0" spinType="1" chargeType="0" colType="0"
          m0="5.27940" tau0="4.58700e-01">
 <channel onMode="1" bRatio="0.0001330" meMode="22" products="12 -11 -211"/>
```

Figure 3.3: The channel B^0 to ν_e, e^+ and π^- as given in the Pythia database. Shown are the properties of B^0 and the properties of the channel.

Pythia channel daughter particles To convert this long name containing all the daughter particle pdg codes into particles, two different functions are used. In addition to a standard function (`mapPyhtiaproducts`), a separate function is made for antiparticles (`mapPythiaproductsAnti`). In both functions the name is split into numbers, using the fact they are separated by spaces. These numbers are then the pdg codes of the daughter particles. For each pdg code, the map connecting pdg code and particle is used to assign the corresponding particle struct. For the anti particle, it has to be taken into account that the decay channel, in comparison of the particle, also contains of the anti particles of the codes that are given. To go from particle to antiparticle, the sign of the pdg code can be flipped. However, some particles are their own antiparticle, and will not have a corresponding negative pdg code. To take this into account, the function first checks if the negative pdg code is present in the map. If not, it is assumed the particle is its own antiparticle. Therefore, the positive pdg code will be used to find the daughter particle. To compare this database to the other database, all particles of each channel are sorted by their pdg codes.

At this point both the ROOT and Pythia databases are converted into maps with identical structure. The final database that is used is the EvtGen database, which again has a different structure.

3.3.3 Mapping the EvtGen database

The functions described in this chapter are given in appendix A.7. The final database that will be mapped is the EvtGen database. As with the previous two databases, first the particles of the database are added to a map. Then, this map is used to construct the channels of these particles. As discussed in chapter 3.2.3 this database consists of two files, one for particles, and one for channels. The channel file has an XML structure and will be accessed in a way similar to the Pythia database. The particle file has a .pdl extension which cannot be accessed using a ROOT class. Therefore, a new function is written to access this database. This function makes use of the fact that the structure is governed by deliminator separated values [30]. The deliminator in the case being tabs. As an illustration the first few values from this file are shown in figure 3.4.

The function `EvtGenDatamap`, that maps the EvtGen particle database, starts by opening the file. It then loops over every line in this file. For every line that is not commented, the function

```
*--------------------------------------------------------------------------------------------------------------------------------------------------
* 5/10/2013 Updated by R. Godang. The format and convention are based on the current evt.pdl and PDG 2012
*--------------------------------------------------------------------------------------------------------------------------------------------------
*                 name                  id      mass/GeV    width/GeV     max_Dm/GeV    3*charge   2*spin    lifetime*c/mm    PythiaId
add  p Particle  K_4*+                    329  2.0450000e+00  1.9800000e-01  2.0000000e-01     3     8  0.0000000e+00          0
add  p Particle  h_b(2P)                110553  1.0255000e+01  0.0000000e+00  0.0000000e+00     0     2  0.0000000e+00     110553
add  p Particle  b                          5  5.0000000e+00  0.0000000e+00  0.0000000e+00    -1     1  0.0000000e+00          5
add  p Particle  anti-nu_e                -12  0.0000000e+00  0.0000000e+00  0.0000000e+00     0     1  0.0000000e+00        -12
add  p Particle  D_2*0                    425  2.4626000e+00  4.9000000e-02  3.0900000e-01     0     4  0.0000000e+00        425
```

Figure 3.4: The first few entries of the particle file of the EvtGen database. Note that it is reminiscent of a deliminator separated file.

creates a particle with the properties of that line. As with the previous databases, two maps are made. The first map connects pdg code and particle, such that these can be compared to the other databases. The second map contains the names of the particles as given in the EvtGen database.

Adding particles to the EvtGen map The function `PopulateParticleEvtGen` adds the properties of each line to the newly created particle. The properties as listed in one line of the EvtGen database are separated by spaces. Each of the words in this line has been assigned a number, according to their position in the line. So the first word is 1, the second word is 2 etc. When separating the words, the program counts how many words have been found. By looking at the number of the current word it can be identified which property is being read. The properties can then be directly assigned to the particle. Note from figure 3.4 that the second property, particle type, is always set to "Particle" in the EvtGen database. A few properties need a slight remodeling to fit the structure of the new particle. In the EvtGen database, not the spin, but twice the spin is listed. Therefore, before being assigned to the particle, the spin is halved. The EvtGen database disagrees with the Pythia database on some of the pdg codes of the particles. To account for this, one of the properties of each particle is the "Pythiaid". This contains the pdg code of the particle in the Pythia database. If the particle is not present in the Pythia database, this value is set to 0. To make comparing the particles possible, this project used the Pythia pdg code if the particle exists in both databases. The lifetimes of the EvtGen database are similar to those of the Pythia database, described in chapter 3.3.2. Again, there are lifetimes in mm and widths in GeV. So the same formula's (equation 5.2 and 5.3) are used to determine the lifetime of the particle. As there are no color multiplet particles in the EvtGen database, the color type is always set to the singlet value: 0. Once all the particles of the EvtGen database have been generated and mapped, the channels are next.

Assigning EvtGen channels As mentioned, the channels of the EvtGen database are held in a separate file. A part of this file is shown in figure 3.5. As discussed in chapter 3.3.3 the XML version of this database is used. Therefore, much alike the Pythia case, the TXMLEngine class of ROOT can be used to read these channels. Besides the channels, the decay file also holds a list of particle aliases. The function `findEvtGenChannels` first goes through all 1^{st} order nodes, containing the alias and channel "decay" nodes. The aliases are added to the map of names. For each "decay", corresponding to a single particle, there is a list of possible channels. The program will then go through these channels and for each node in the file construct a new channel for the particle. To construct such a channel the branching ratio, mother particle and daughter particle are needed. The branching ratio can be access from the file. For the mother and daughter particles,

the names are given. The mother particle is the title of the "decay" node. Using a combination of the map of name to pdg code and the map from pdg to particle, the mother particle is found. The daughter particles are given as one single string of text The function `mapEVtGenproducts` separates the individual daughter particle names. Then the map of name to pdg code is used to find the pdg code, and the map from pdg code to particle is used to find the corresponding particle. The last step is to sort the daughter particles based on their pdg codes.

```
<!--   Vector Mesons Updated PDG 2008 -->
    <decay name="D*+">
        <channel br="0.6770" daughters="D0 pi+" model="VSS"/>
        <channel br="0.3070" daughters="D+ pi0" model="VSS"/>
        <channel br="0.0160" daughters="D+ gamma" model="VSP_PWAVE"/>
    </decay>
    <decay name="D*-">
        <channel br="0.6770" daughters="anti-D0 pi-" model="VSS"/>
        <channel br="0.3070" daughters="D- pi0" model="VSS"/>
        <channel br="0.0160" daughters="D- gamma" model="VSP_PWAVE"/>
    </decay>
```

Figure 3.5: An illustration of how the channels are written in the EvtGen database. Shown are the D^{*+} and D^{*-} channels

Now all three databases are mapped to a similar structure. In the next chapter these will be compared and combined into a new and as complete as possible database.

3.3.4 Merging the three maps

To merge the three database, one has to be chosen as a basis to compare to the other two. Of the three databases used, the EvtGen database is found to be most complete. Therefore, it is assumed to be the most up to date database. The map of the EvtGen database is taken as the basis, with which the Pythia and ROOT maps will be compared. The merging consists of four parts. Firstly the particles and their properties are compared using the pdg codes. Secondly, the channels of these particles are compared, using the fact that all daughter particles are sorted by their pdg codes. Thirdly, the names of all the databases will be combines in a single map from name to pdg code. Lastly, there are some manual additions to the program, which are not found in any of the databases.

The code to merge the maps can be found in appendix A.8. As the EvtGen map is taken as a basis, the code begins by looping over the Pythia map and comparing values. If the particle of the Pythia map is not present in the EvtGen map, it is simply added to the EvtGen map. If a particle from the Pythia map is also present in the EvtGen map the function the two particles are merged. After this, a similar loop is performed over the ROOT map, merging when particles are in both databases. Both merging functions will now be discussed.

Merging EvtGen and Pythia particles The input of the function `merge_EvtGen_Pythia_particles` are the particles of the Pythia and EvtGen maps with the

same pdg code. In the EvtGen map no color has been assigned to the particles. If any particles should have a color type other than singlet (e.g. quarks), then the Pythia particle holds this information. Therefore the color type of the Pythia particle is also applied to the EvtGen particle. As was mentioned in chapter 3.3.3, the type of a EvtGen particle is always set to "Particle". Therefore, if the particle is also present in the Pythia database, its type is copied into the EvtGen particle. Two generic functions, on lifetime and decay channels, are called. These will be described later in this chapter.

Merging EvtGen and ROOT particles To merge EvtGen with ROOT, the function `merge_EvtGen_ROOT_particles` compares particles from the ROOT map and the EvtGen map. This function is called after the merger between the EvtGen and Pythia maps has been completed As mentioned in chapter 3.2.1 the particle type from the ROOT map is, to some extent, used to identify BSM particles. At this point the particle type is given by the type assigned in the Pythia or EvtGen database, depending on which database the particle came from. These mean some particles can have the generic type "Particle" from the EvtGen database or "Pythia(Anti)Particle" from the Pythia database. If the particle has such a generic type, the ROOT database might help further defining the type of particle Therefore, such types are overwritten by the type given for the ROOT particle. The two other functions in this function are the generic functions governing Lifetime and channels. These will be discussed next.

Combining lifetimes To avoid undefined behavior in the program, the function `check_lifetimes` makes sure that all particles have a specified lifetime. An example of undefined behavior that might otherwise occur is determining a particles lifetime through its width, when there is no width defined. Even when both values are set to 0, the lifetime will be calculated by dividing $\bar{h}$ by the width, causing undefined behavior by dividing by 0, instead of dividing by an undefined number. This function checks, for particles that have no lifetime, or a lifetime of 0, if the other database, first Pythia, then ROOT, has a value for that lifetime. If no lifetime is found, the lifetime is set to 0 seconds. It has been checked that all particles that still have a lifetime of 0 in the final version of this database are either objects beyond the scope of this project (e.g.: pomeron, pi_diffr, He3) or particles with a very short lifetime (e.g.: B_c*+, Lambda(2625)+, Omega_c*0). It is therefore sufficient to let these particle have a lifetime of 0.

Merging the channels The function `mergechannels` merges the list of channels for a specific particle given the lists of channels of that particle in two databases. This function is used in both the merge of the EvtGen particle map with the ROOT particle map and the merge of the EvtGen map with the Pythia map. For the merging of the channels two sets of channels are defined. The primary channels are the basis, and are assumed to already be in the decay list of the particle. The secondary channels are the channels from another particle map, which need to be compared to the primary channels. For the end result it does not matter which channels are chosen to be the primary channels. The EvtGen channels are taken as the basis. The merging of the channels is done by first looping over the secondary channels. These are compared to the primary channels, which can be either the Pythia or the ROOT channels. For each of the secondary channels, all the primary channels are checked. If the two channels are the same, the secondary channel is ignored, as it is already in the combined map. If the secondary channel does not match any of the primary channels, it is added to the list of primary channels.

Comparing channels While mapping each of the databases. (sections 3.3.1, 3.3.2 and 3.3.3). The products of each channel were sorted using their pdg code. This is necessary to be able to compare the channels of each particle. The function `compare_particlevectors` return true or false, telling if two specific channels have exactly the same products or not. To do so the mismatch function [31] is used. This function compares two lists of particles, using their pdg codes. It gives the first entry where the two lists do not match. If all the values match, the two channels contain exactly the same daughter particles. Therefore they are essentially the same channel, only from different databases. such a channel is then not added to the new particle map, as it is already in there.

The name maps While mapping each of the databases the names of each particle was also mapped to its pdg code. This was done because each database might have a different name for the same particle. As the names of the particle can be used as input in the functions seeking (semi-)peaking backgrounds (chapters 4 and 5) it is important that the different names for each particle are in a single map. The function `combineNames` loops over the map of names of a secondary database (in this project the ROOT and Pythia map) and for each name checks if it is present in the primary map (in this project: the EvtGen map). If the name is present, it is the same in both databases. In this case nothing needs to be done. If the name is not in the primary map, then it will need to be added to that map. This way all possible names for each particle are stored in a single map.

Manual additions The last function that is used in merging the maps contains several manual additions that could not be extracted from any of the databases. Here it is important to note that none of the particle properties will be altered. This is because the end product is meant to be a general database containing all the properties from the specialized databases. Altering particle properties to suit this projects needs would create, again, a specialized database. This function adds several particle types. The particle type is only present in the ROOT map, and even there it is not consistent (section 3.2.1). In the Monte Carlo numbering scheme [29] it can be found that particles with pdg code in the range from 31 to 100 are BSM and pseudo particles. To make sure these do not end up in the end result, they are given the particle type "Unscope". This way the program is able to recognize such entries. This is also done for the 4^{th} generation lepton and its neutrino

In addition to flagging these particles with a new type, two more additions were done. To the map of names the Upsilon(1S) is added as an alternative way to find the Upsilon particle. This was done to have consistency in the Upsilon line of particles. Now these are named Upsilon(1S) to Upsilon(4S). Note that also the original Upsilon, and the Upsilon' for Upsilon(2S), from the ROOT database are still in this names map. Therefore, these names can be used as input in finding backgrounds (chapters 4 and 5). The last manual addition concerns stable particles. These particle cannot be assigned a lifetime, as they do not decay. Therefore, in term of lifetime, stable particles cannot be distinguished from BSM particles, which also do not have a defined lifetime. To make this distinction, a lifetime of $3.14 \cdot 10^7$ seconds is given to these particles. This roughly corresponds to the amount of seconds in a year. Because this value is chosen to fit this program, it slightly diminishes the generality of this database. However, the value is chosen to be so nonphysically large that any use of the lifetimes will effectively mean a infinite lifetime. Future versions of the program might alter this value, examples of alternatives to flag stable particles include a negative lifetime or adding a new particle property, telling whether a particle is stable or not.

Notes on particle properties Many of the particle properties mentioned in the start of this chapter were not compared between the different databases. These include properties like spin and charge, which are given in integer numbers and therefore very precise. No instances were found where the charge of a particle was not in the database. Other values cannot be determined with the certainty of integer numbers. The main example of this is the particle mass. The value of the mass for each particle might vary slightly in each of the databases. The mass is not cross referenced in each database. Instead, the particle mass given in the database in which the particle was first found is given to the particle. The order in which the databases are checked is: first EvtGen, then Pythia, then ROOT. This order is chosen based on the completeness of the given databases, starting with the most complete. It should also be noted that the difference in mass between the databases cannot be very large. It is assumed that the uncertainty in the mass is much lower than the uncertainty in the position of the background of this project. For the lifetimes a similar argument can be made. For example, the lifetimes are used to determine whether particles reach the detector, this is more a matter of the order of magnitude of the lifetime, rather than the exact value.

The pdg codes for known particles are given by established rules given by the Particle Data Group [29]. There is some disagreement between the EvtGen database on one side and the Pythia and ROOT databases on the other side. However, this does not affect this project, as the EvtGen database also holds the Pythia pdg code for each particle. For some other "particles" in the database, no well established numbering scheme is present. This does lead to some misunderstanding between the databases. For example, the Blackhole in Pythia has the same pdg code (40) as the pomeron in the ROOT database. Because neither of these particles in used in the project, this is of no concern.

Note on channel properties Besides the particles, also the channels hold some properties. The main property that can be checked is the branching ratio of each channel. One check that was performed was whether all the branching ratios of the particles add up to 1. It was found that this is not always the case. However, it was also found that the separate databases do not have a total branching of 1. For example, the total of all the B+ channels in the Pythia database is 1.27251. It is thought that this is due to channels existing in the database in multiple formats. For example, the channel to $\bar{u}$, $\bar{d}$, u and u (branching = 0.003) is a different channel than the one to $\pi^0(u\bar{u})$ and $\pi^+(u\bar{d})$ (branching = $5.5 * 10^{-6}$). Because of this, it is assumed that particles in the combined database can have branching ratio's which add up to more than 1. In that case it is assumed that similar channels are in the decay list multiple times. A method of finding which channels are in such a list multiple times is not known. However, in the example previously shown, only one channel, the pion channel, has end products which can be detected. Therefore, if the process is considered a background, it will only be found once.

At this point all three particle maps are combined into a single map. However, to go to a combined database this map needs to be converted back to a database format.

3.4 Writing combined database

The particle maps are written in two particle database formats. Once in XML format and once in the format of the ROOT database. Both will be used to find backgrounds. The functions

used to write the combined particle map in database formats are given in appendices A.9(XML) and A.10(ROOT).

3.4.1 Writing XML database

The XML format of the combined database is similar to the Pythia database. The advantage of writing the combined database in XML format is that any property can be added to the database, if so desired. Another advantage is that the XML database can be accessed in ROOT, as shown in section 3.3.2 on mapping the Pythia particles. The combined map consists of two parts, a map of names to pdg codes and a map of pdg codes to particles. The function to read both maps performs a loop over each map. The first map that is checked is the name map. For each name in the map the corresponding pdg code is looked at. If, in the particle map, the particle with that pdg code does not have that exact name, it is considered an alias. At the first part of the combined XML file all aliases are listed, together with the name that the particle has in the actual database and its pdg code.

The particles map is then added to the database. All particle properties mentioned at the start of this chapter are then added in the particle XML node. These properties are: pdg code, name, spin, charge, mass, lifetime, color type and particle type. Within each particle node multiple channel nodes can be added. The function loops over each channel and add the property branching ratio to the node. Further into the XML structure, each channel node also contains several product nodes. Each of these product nodes contains the names and pdg code of the daughter particles in this channel. An example of the rewritten database can be seen in figure 3.6 where the π^+ particle and its channels are shown.

```
<particle id="211" name="pi+" spin="0" charge3e="3" mass_GeV="0.13957" lifetime_s="2.60124e-08" color_3times="0" type="Meson" >
<channel branchingratio="0.999877" ><product id="-13" name="mu+"/><product id="14" name="nu_mu"/></channel>
<channel branchingratio="0.000123" ><product id="-11" name="e+"/><product id="12" name="nu_e"/></channel>
</particle>
```

Figure 3.6: An illustration of the rewritten XML database. Shown are the π^+ meson and its channels

3.4.2 Writing ROOT database

Besides the XML database, a second version of the rewritten database is written. This version is written in the format of the ROOT database. See section 3.2.1 for a description of this database. The main advantage of writing the database in this format, is that ROOT is optimized for it. Therefore, it is the fasted way of accessing and looping over the entire database. The major disadvantage is that is does not provide the freedom of adding any particle property desired. When adding a new particle to the ROOT database, the program is limited by the AddParticle function. With this function particles can be added to an existing database. The input of this function is shown in figure 3.7 [20]. When compared to the input needed for this program, there are three properties that cannot be added in the ROOT database: lifetime, spin and color type. For lifetime, the width is used, which can be accessed in the add particle function. It is possible to convert from lifetime to width. This would require equation 3.3 to be rewritten into 3.4:

$$\Gamma = \frac{\hbar}{\tau} \tag{3.4}$$

However, as mentioned in chapter 3.3.4 on merging lifetimes there are some particles without defined lifetimes. As these particles life for a very short time, setting the effective lifetime to 0 works. However, when converting to width, dividing by 0 leads to undefined behaviour, as the effective width of these particles would be ∞. Therefore is was decided to simply enter the lifetime in the rewritten ROOT database, as if it was the width. This does limit the applications of the ROOT version of the rewritten database outside this project.

In the XML database lifetime is written in seconds. This does create the challenge that the lifetimes in the two rewritten databases do not hold the same value. This is important to keep in mind as the two databases can be used simultaneously. For the color type, it has been made sure while building the Pythia map (section 3.3.2) that all non color singlet particles were given the type "ColorMultiplet". Since particle type can be added to the new particle, the ROOT database now has the necessary color information. The only property that cannot be added to the ROOT database, and still is needed for this project, is the particle spin. This greatly limits the usefulness of the ROOT database.

```
TParticlePDG * TDatabasePDG::AddParticle ( const char * name,
                                           const char * title,
                                           Double_t     mass,
                                           Bool_t       stable,
                                           Double_t     width,
                                           Double_t     charge,
                                           const char * ParticleClass,
                                           Int_t        PDGcode,
                                           Int_t        Anti = -1,
                                           Int_t        TrackingCode = 0
                                         )
```

Figure 3.7: The function "AddParticle", from the ROOT software. The possible properties that can be assigned to a new particle in the ROOT database are dictated by this function.[20]

Unfortunately, it is not possible to open an empty ROOT database (given by the TDatabasePDG class). Nor is it possible to edit a particle already in the database. Therefore it is not possible to directly write a new database, using the build-in functionality of ROOT. To circumvent this issue, a new plain text file is created and immediately closed again. This created a new, blank text file with the title of the rewritten ROOT database. This empty file is then read as if it is a ROOT database file, creating a blank database, where any particle can be added.

In a ROOT database, particle and anti-particle are coupled. Each antiparticle has a name and the negative pdg code of its particle. For example, the π^- particle has only the name pi- and

pdg code -211. Whereas the π^+ particle, with pdg code 211, holds the properties and channels of the two particles. See figure 3.8 for the $\pi^{+/-}$ mesons in the rewritten database. For information on the particle properties, see chapter 3.3.1 on mapping the ROOT database. When accessing an antiparticle, these properties are also assigned to the antiparticle. The ROOT database automatically accounts for the change in sign for the charge. It also considered that the products of the channels should be converted to their respective antiparticles, if they exist. Because the ROOT database has a coupling between particle and anti-particle only particles with a positive pdg code will be considered while looping over the particle map. Each particle is added using the Addparticle function. For each particle it is checked if an antiparticle exists in the map. If it does, its name is added as an antiparticle. After a particle is added into the ROOT database, it is immediately loaded from that database in the ROOT TParticlePDG format. This is done to add the channels to that particle.

```
  115 pi+                 211   1 100 Meson  3 1.39570e-001 2.60124e-008 0 0 0 -1 0 2
#-------------------------------------------------------------------------
#    decay  type(PY6)    BR     Nd         daughters(codes, then names)
#-------------------------------------------------------------------------
         1     0   9.99877e-001    2       -13     14    mu+         nu_mu
         2     1   1.23000e-004    2       -11     12    e+          nu_e
  116 pi-                -211   116 0
```

Figure 3.8: An illustration of the rewritten ROOT database. Shown are the π^+ meson and its channels

To add the channels to the ROOT particles the function AddDecayChannel is used [21]. While looping over the channels of a specific particle, the branching ratio can be added directly into this function. The format of the daughter particles is slightly different compared to the particle map. The ROOT database will only require the pdg codes of the daughter particles. In addition, the number of particles is needed to complete the ROOT channel. These properties can be accessed relatively simple from the channel listings as used in the combined particle map. With the channels added to the particle, the database is complete, and is written into the empty text file.

Besides the particle file, a second file is created, this file is in XML format and contains the aliases of the particles. This is done in the same manner as in the previous case, where the entire database was written in XML format. In future versions of this program, the alias function might be separated from the two particle map functions.

Updating databases The versions of the databases used are: R01-06-00 for the EvtGen database [19], launched in 2010, Pythia 8, also from 2010 [32], and ROOT v5.34/36, which was launched in 2016 [33]. It should however be noted that the newest database does not necessarily hold the newest value. In fact, when looking at the mass of the Higgs boson, ROOT gives a value of 80 GeV, Pythia has 115 GeV, and EvtGen has 125.7 GeV. As the EvtGen value is the correct one [3], it could also be assumed that this is the most recent database. When a new version of one of the databases is desired, the rewritten, combined database has to be updated. For this purpose, the function `initialize_database` , given in appendix A.11, is used. This function maps the three databases, performs the merge and writes a XML and ROOT version of the new database. When updating the database, the file names of new version of the three basis databases

should be taken into consideration. In the current version of the program, four files are used: the pdg_table.txt from ROOT, the ParticleData.xml from Pythia and the evt.pdl and DECAY2010.xml from EvtGen. The first three of these file will probably not change name in future version of their respective databases. However, the EvtGen decay file has a date in the filename, and might change. Therefore, if updating the EvtGen map, in the function mapping the channels (appendix A.7), the filename should be changed.

3.5 Remapping combined database

Another possibility, next to rewriting the database, is using the combined map from section 3.3.4 directly. Saving this map in such a way it can be loaded instantaneously is unexplored in this project. However, two function were made to remap the database after writing it into the two formats of the rewritten database. These functions are given in appendix A.12. The function to map the XML database is very similar to the Pythia database. For how to map a XML database, see that section (3.3.2). A slight difference is that for mapping the rewritten database, the particle properties are already given in the desired format. Therefore, everything can be loading into the particles directly. Another difference is the structure which contains three XML nodes: particle, channel and products. The function loops over these three nodes. With the other function in appendix A.12, the ROOT database is remapped. This is done is the same manner as the original ROOT database (section 3.3.1). Also for this database holds that for mapping the rewritten database, the particle properties can be copied directly into the particles. The only exception being the width, which will have to be converted into lifetimes, using equation 3.3. Initially, these functions were used in the background functions (chapters 5 and 4). However, as it is very time consuming to remap the database, the current version used the databases directly. Remapping takes about 2.5 minutes, compared to several seconds for locating peaking backgrounds directly from the database (see section 4.5). One unexplored suggestion is to write the particle map to a binary file, enabling a quick reload of the map into the program. Despite these functions not being used to search backgrounds, it is still possible to recreate the map if needed.

Now there is an as complete as possible, and as detailed as possible database. This database has two formats, a XML and ROOT format. Both will be used in the functions seeking (semi-)peaking backgrounds. The next chapters will describe how this rewritten database is used to locate backgrounds in two body decay processes. The peaking backgrounds in chapter **??** are found using both the XML and ROOT versions of the database. The semi peaking backgrounds in chapter 5 only require the ROOT database.

Chapter 4

Peaking backgrounds

In chapter 2.3 three kinds of background were mentioned. The combinatorial, peaking and semi peaking backgrounds. As mentioned, the combinatorial can be easily approximated by a smooth function. To find the other backgrounds, two functions were written. This chapter describes the function to find peaking backgrounds. Chapter 5 describes the function to find the semi-peaking backgrounds.

4.1 Simulation without misidentification

Before constructing a program to find the backgrounds, it is essential to know what the spectrum will look like if there is no misidentification. For this, a given decay will be simulated. Thereafter, the mass and momentum detection will be simulated. Lastly, the decay is reconstructed, using the detected values. This reconstructed mass is then plotted in a histogram. In chapter 2.2.3 two detector effects were mentioned that affect the reconstruction. These are the detector resolution and Bremsstrahlung (for electrons). To implement detector effects up to high precision is beyond the scope of this project. Therefore, a simplified implementation of these processes is used. To simulate the correctly identified decay, the program that simulated the misidentification is used. The particles are then "misidentified" as themselves, thus, properly identified.

Detector effects The code of the function implementing the detector effects is given in appendix A.14. This function looks at a certain particle, defined by its rest mass. This particle has a Lorentzvector, holding the momentum in the x, y and z direction, as well as the energy of the particle. With this information the energy of particle is "detected". Combined with the energy of the other daughter particle, this value gives the invariant mass of the decay products.

The detector effects are governed by a random number generator. This can generate a random distribution given a certain function [34]. The detector resolution is given by redefining the energy of the particle using a Gaussian function. The actual energy is taken as the center of this Gaussian distribution. The width is given by the standard deviation σ. By default this is set to 0.001 times the energy, but this can be changed at will (see 4.2). The "measured" energy is then a random number picked from this Gaussian. This uncertainty in energy is a very basic way to simulate the

uncertainty in the momentum measurement.

Bremsstrahlung is also taken into account in this function. The energy lost for each particles is proportional to its energy(E) and mass(m)[35], as shown in equation 4.1.

$$E_{lost} \propto \frac{E^2}{m^4} \tag{4.1}$$

Because of this mass dependence, Bremsstrahlung will only have a measurable effect in the case of electrons. For other particles it is still applied, but not expected to have a noticeable effect in the reconstructed mass spectrum. To simulate the Bremsstrahlung, a proportionality factor for this energy loss will have to be defined. For each particle in the function this energy loss is given by an exponential function. From a distribution, characterized by $Ae^{-\frac{t}{\alpha}}$, a number is picked. Here, A and α are constants. To change the amount of Bremsstrahlung, α can be user defined (see chapter 4.2). The effective energy loss is then given by equation 4.2.

$$E_{lost,effective} = r \cdot \frac{E^2}{m^4} \tag{4.2}$$

Where r is the random number from the exponential distribution mentioned above. As mentioned the exact Bremsstrahlung spectrum is beyond the scope of this project. The default values of A (10^{-15}) and α (0.005) value are therefore chosen to generate invariant mass spectra which resemble real spectra. If a more physics bases approach to Bremsstrahlung is required, it is recommended to use specialized programs for that process. The amount of Bremsstrahlung is determined by the parameter α. The higher α, the more Bremsstrahlung.

In the sample spectrum D^0 to $e\mu$ figure 2.6 changes to the histogram shown in figure 4.1. The detector resolution widens the peak of the invariant mass. The Bremsstrahlung adds a tail on the left side of the spectrum, because of the energy lost.

4.2 Input parameters

Once the signal histogram can be constructed, the misidentification candidates have to be found. The first step is to define the input parameters. There are two kinds of input parameters in the misidentification function. Global parameters and the input of the function itself. Global parameters are the parameters defined outside the function. This is done in the file `globals`, which is given in appendix A.15. In this function three categories are defined: "constants", "globals", and "cascade". In the first category $\hbar$ is defined, to convert lifetime and width. The second category, "globals", holds parameters which are used in both the function seeking peaking backgrounds and the function seeking semi-peaking background (chapter 5). The last category "cascade" is used only in the semi peaking function.

Global parameters For all parameters in this function holds that they can be changed by the user of this program, if so desired. Among the parameters that are used in the "globals" function are the two parameters already discussed in section 4.1, `alpha_Brem` and `sigma`. In addition there are four more parameters. These are called `detector_tau`, `deltaW`, `minBR` and `Nevents`. The detector lifetime (tau) corresponds to the particle lifetime that is needed to reach the first detector.

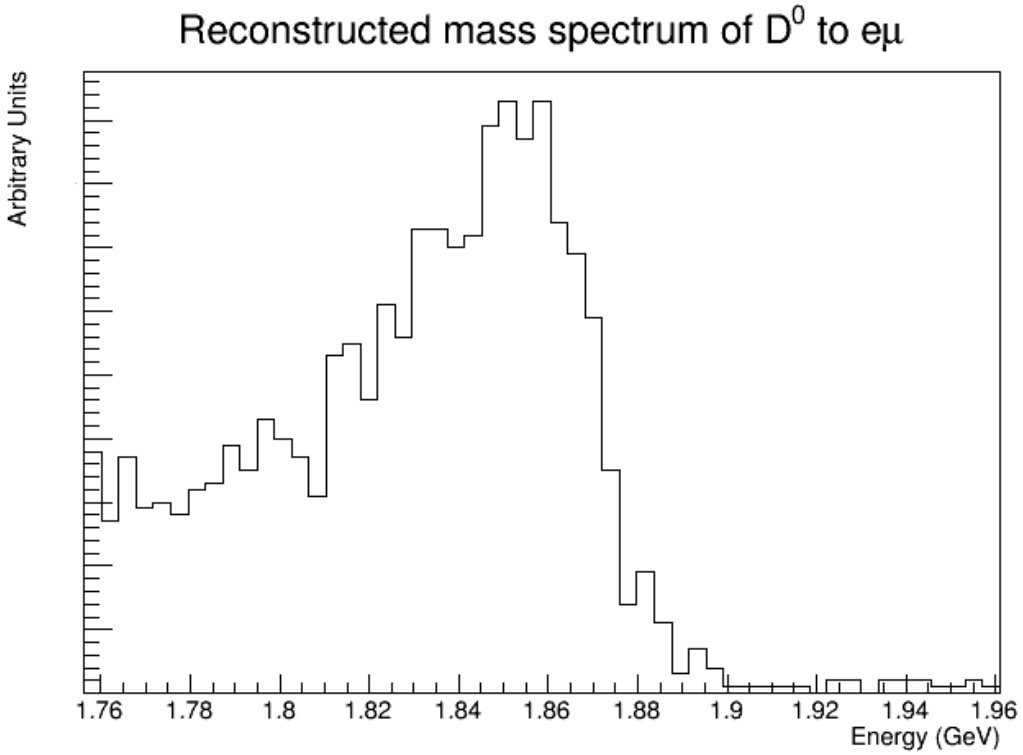


Figure 4.1: A histogram of the reconstructed mass spectrum of D^0 to $e\mu$ with extremely simplistic detector resolution and Bremsstrahlung effects added to the decay

For the LHCb detector this is set to 1ns. For relativistic particles ($\gamma = 3$, 94% of light speed) this corresponds roughly to 1m of flight. This value was chosen, because at 1m the first particles are detected in the LHCb detector (see chapter 2.2). Particles which have a lifetime much lower then this lifetime will decay before reaching the detector. Therefore, these will not be considered a possible background. `deltaW` defines how much the invariant mass of a background can differ from the expected value. This is given in GeV. For example, take a search for a D^0 meson, which has a mass of 1.86 GeV, and take a ΔW of 0.1 GeV. This will generate all backgrounds with reconstructed mass in the region between 1.76 and 1.96 GeV. Such a region will be referred to as a search region. For the D^0 results see section 4.5 on validation. The value `minBR` gives the minimum branching ratio at which a background is included is the result. As mentioned in chapter 3.3.4 the branching ratio's of the different databases can differ between the different databases. Because of this the default value is to include all possible backgrounds, regardless of the branching ratio. To do so the minimum value is set to -1. As branching ratio's are only 0 or positive, none will fall below -1, and all will be accepted. Despite the default value not changing the outcome, this functionality is in the program. This is to facilitate any desire to put constraints on the branchingratio's. The last value, `Nevents`, determines how many events are generated per background. Note that adding more events will increase the time the function takes to run.

Function parameters The input parameters of the misidentification function itself (appendix A.13) are five masses. As explained in chapter 2.3.2, peaking backgrounds arise due to a misidentification. The five input masses are the masses of the particles involved in the decay that is actually simulated, and the two masses assigned to the daughter particles after (mis)identification. The decay involves three particles, a mother particle and two daughter particles. After the decay the two daughter particles are assigned the mass that is given for the misidentification particles. It is important to note that the misidentified particles will be the signal in the end, despite not being in the original decay. With this function it is possible to simulate a single peaking background, see section 4.4 for the process of simulating these backgrounds. As an end result, all possible peaking background given a certain 2 body decay are expected. The next step is listing which misidentifications are possible.

4.3 Selecting Peaking backgrounds

To select all peaking backgrounds given a specific decay, it is essential to know the kinds of peaking backgrounds that can be expected. Chapter 2.3.2 explains how misidentification leads to these peaking backgrounds. Such an event concerns two groups of particles. The mother and daughter particles of the decay that is searched for, referred to as the signal. And the particles of the decay that actually happens, referred to as a background candidature.

It is even possible to have a peaking background without misidentification. Chapter 2.2.3 describes how two daughter candidates are recombined. The reconstructed mass is then used to identify the mother particle. It can happen that a particle with a mass very close to the signal mother particle produces the exact daughter particles of the signal. If these are then *correctly* identified, the reconstructed mass might look like the signal. Figure 4.2 illustrates an example of such occurrence, which is found in the search for $B^0_{(s)}$ to $\mu\mu$. Figure 4.3 shows this decay in the 2015 CMS and LHCb analysis [14] compared to the 2017 LHCb analysis [7]. The 2017 analysis is also used as validation case in section 4.5. When comparing these results, it is essential to note the overlap in peaks in the 2015 analysis. This overlap is mostly gone in the 2017 version. This is because of the improved resolution of the analysis, allowing the peaks to be separated.

Using the databases constructed in chapter 3, these kind of background can be easily found by searching for channels which exclusively have the daughter particles of the signal. One existing program to find these kinds of background is *fstate* [36]. As mentioned, the resolution of the measurement is essential in avoiding these backgrounds. By changing the parameters of the search function (section 4.2) it is possible to define which decays are considered a candidate. This is governed by two parameters. The resolution is given by width (σ) of the measurement uncertainty. The parameter ΔW determined which backgrounds candidates are in the search region.

Most peaking background will occur due to particles being misidentified. To locate these backgrounds it is essential to search the database for particles which can be misidentified as the signal particles. For the search for $B^0_{(s)}$ to $\mu\mu$ the mean contributions are from pions and kaons, identified as if they were muons. An example of this is shown in figure 4.4, where a B^0_s decays to a kaon and a pion. These are subsequently misidentified as muons.

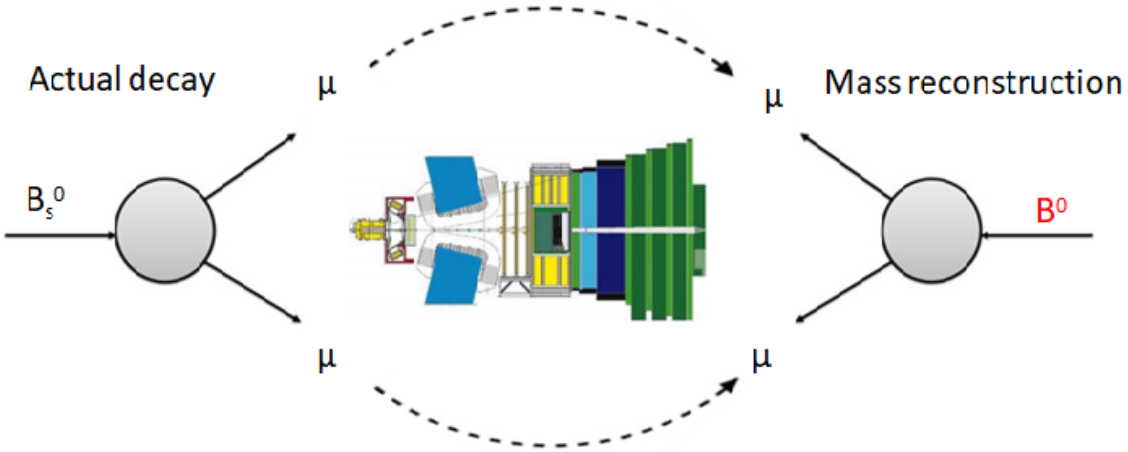


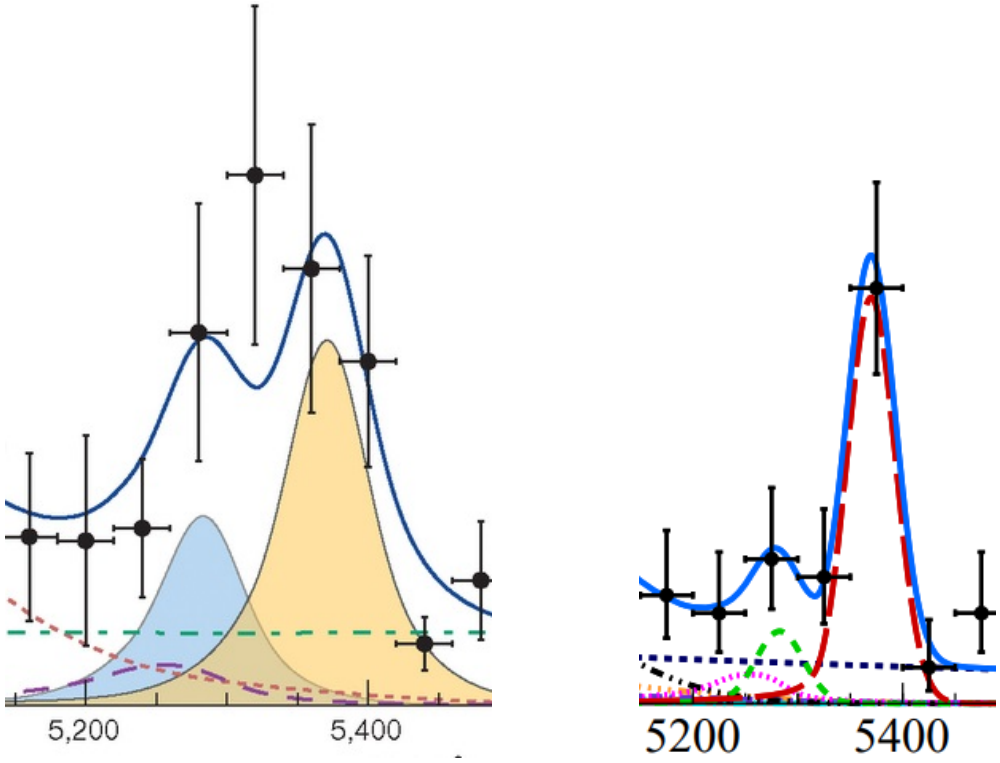
Figure 4.2: Because the mother particle is identified based on the reconstructed mass of the daughter particles, backgrounds might occur. This figure illustrates B_s^0 to $\mu\mu$ identified as B^0 to $\mu\mu$ in the LHCb detector

A combination of the situations in figures 4.2 and 4.4 is also possible. Here, both the daughter particles are misidentified, and the assumption on the mother particle is wrong. The most general case of a misidentification is shown in figure 4.5. In this picture the general decay X to ab is shown as a background, where X, a and b can be any SM particle. The following paragraph will describe the main function of the peaking backgrounds. This is the function to determine for which combination of X, a and b will be considered a background.

4.3.1 Background candidates

The main function to find peaking backgrounds is given in appendix A.16. This function loops through the database and combines 3 random particles to create a background candidate. Given the database created in chapter 3 there are 1276 particles in total. If all BSM particles are excluded, 1030 particles are left. Combining all particles in random groups of 3 would give over 1 billion combinations. To find which backgrounds candidates are significant enough to be considered a background, several constraints will be used. These constraints will be based on physics and help decrease the number of combinations.

The background candidates will have to be compared to a signal. This signal is defined by the user of the main function `MisID`. The resulting backgrounds will then be shown in a pdf file with the name that was used as filename input. As several characters are needed to open and close the pdf file in ROOT a function handling this name is needed, listed in in appendix A.17. The first is to use the names of the particles involved in the signal decay. For example: `MisID("B0", "mu-", "mu+", "B0mumu")` will generate a file B0mumu.pdf containing all peaking backgrounds of B^0 to $\mu\mu$. The second possibility is to use the pdg codes of the particles as an input parameter. In



(a) CMS and LHCb

(b) LHCb

Figure 4.3: A comparison between zooms of the $B^0_{(s)}$ to $\mu\mu$ mass spectra in figures 2.7 and 4.8 to illustrate the importance of resolution. In figure (a), the 2015 CMS and LHCb [14] reconstructed mass spectrum of the decay. Here, the yellow peak is B^0_s to $\mu\mu$ and the blue peak is B^0 to $\mu\mu$. In figure (b), the reconstructed mass spectrum of LHCb in 2017 [7]. Here the red dotted line gives B^0_s to $\mu\mu$ and the green dotted line gives B^0 to $\mu\mu$. In both cases, the purple line gives the peaking background. As can be observed, figure (a) shows much more overlap between the $B^0_{(s)}$ peaks than (b)

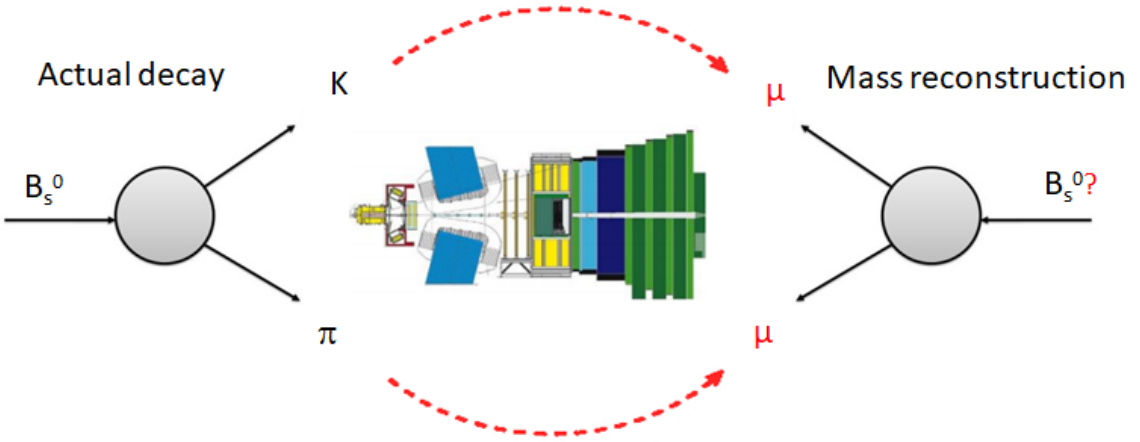


Figure 4.4: An illustration of B_s^0 decaying to $K\pi$, where the K and π are both misidentified as a μ

the case of B^0 to $\mu\mu$ this would look like: `MisID(511, 13, -13, "B0mumu")`. As the mother particle is only used to define the invariant mass of the signal, it is also possible to express this in a number. For example, the input for searching for a 5.5GeV invariant mass in $\mu\mu$ would be `MisID(5.5, "mu-", "mu+", "mymumu")`. The mother particle is always listed first in this function. However, the order of the daughter particles does not matter. The last value is always the filename of the result.

Whatever the input, the next function used is the `MisIDInput` function, listed in appendix A.18. This function makes sure that the names or pdg codes given as input actually exists in the database. As names might differ between the databases the map of names decried in 3.3 is used to make sure all possible variation of a particle name are included. If a name or pdg code is not found, this function send out a message telling what is missing and cancels the search for peaking backgrounds.

4.3.2 Reducing the number of background candidates

To convert the database from text to a list of particle, the function `particledata` is used. This is listed in appendix A.19. This function creates a FParticle (appendix A.3) for each particle in the XML database. this is done in a similar fashion as reading the Pythia database (chapter 3.3.2). While listing the particles from the database, two constraints can already be applied. The first constraint is taking out all BSM particles. The function doing so is listed in appendix A.20. This is done using the particle type that was defined while constructing the database in chapter 3. Table 4.1 shows the disallowed particle types. The second constraint is based on charge conservation. It is assumed that the charge of the particles is always correctly identified. As the backgrounds have to be SM allowed, this means that the charge of the mother particle has to be the sum of the daughter particles for each background. For the signal, this is not a necessity. The function makes three list of particles, based on the charge of the three signal particles.

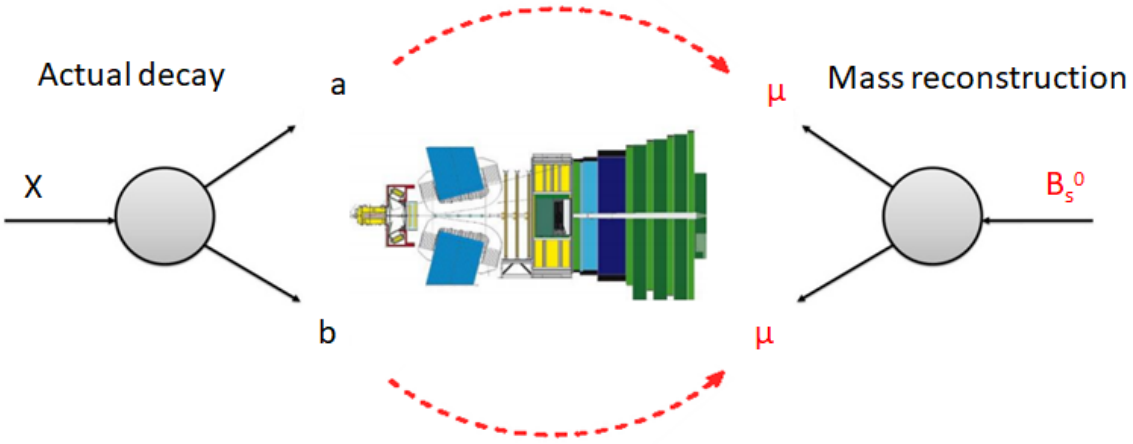


Figure 4.5: A general illustration of misidentification using the LHCb detector. X decaying to a and b, with are misidentified as B_s^0 to $\mu\mu$. X, a and b can all be any SM allowed particle.

Table 4.1: Types of non SM particles

Type	Note
Technicolor	
Unscope	Assigned while building database
Sparticle	
ColorMultiplet	Any non-singlet particle
Quark	Color confinement: no isolated quarks
Generator	Pseudo particles (e.g. cluster, jet)

Using these three lists, background candidates are constructed by taking a particle from each list and making all possible combinations. These combinations are made in the function `PCombine`, also listed in appendix A.16. This function perform a number of checks on the physics of each combination. This is to test if the mother particle can indeed decay into the daughter particles it was combined with. Each of these checks will be mentioned below. These checks are performed in the function listed in appendix A.21.

Energy conservation The primary check of every background candidate is whether energy conservation is respected. This is done by checking if $m_0 < m_1 + m_2$. Here, m_0 is the mass of the mother particle, and $m_{1,2}$ are the masses of the daughter particles. If this is the case, the background, which is the decay which physically should occur, cannot exist without violation energy conservation. It is then not taken into account as a background candidate.

Angular momentum conservation The second conservation law that is used is angular momentum conservation. This function is the reason the ROOT database cannot be used to find

peaking background candidates. This is because the spin of the particles is used, which cannot be added to ROOT (see chapter 3.4.2). The function to check the angular momentum, called `spinCheck`, used another function. This secondary function, `iCheck`, checks if a given spin is an integer spin. The function itself uses this to check whether the two daughter particles have the same "integerness". If they do, the mother particle must be an integer spin particle, if not, it must be half integer spin. Table 4.2 shows which combinations of particle spin are allowed. Here, half stands for half integer spin.

Table 4.2: Spin combinations for peaking backgrounds

mother spin	daughter1	daughter2	allowed?
half	half	half	No
half	half	integer	Yes
half	integer	half	Yes
half	integer	integer	No
integer	half	half	Yes
integer	half	integer	No
integer	integer	half	No
integer	integer	integer	Yes

Lepton number conservation The final conservation test is lepton number conservation. The lepton number of each particle is extracted using the particle type. All particles with the lepton class have a positive lepton number. All antiparticles with this class have a negative number. For any other class this number is 0. For a background candidate to be allowed, this property has to be conserved throughout the decay. Note that a similar approach on baryon number was not possible as the particle type does not specify quark content. See chapter 3.2.1 for the inconsistencies in baryon types.

Lifetime Besides the conservation laws, for each particle it is also essential to know if it is detectable. To test this, the lifetime of the particles is used. If a particle does not life long enough to reach the first detector, it cannot be detected directly. By default this minimum lifetime is set to 1ns. As mentioned in section 4.2, this corresponds to about 1m of flight, or more. This value was chosen to resemble the LHCb detector (chapter 2.2), but can easily be assigned any value (see section 4.2).

Search region For each physically allowed background that is also detectable one final check has to be performed. This checks if the reconstructed mass of the peaking background will end up in the search region. This search region is defined by the user in section 4.2. For a peaking background candidate to be in this region, the difference in invariant mass between the signal and the background has to be smaller than the ΔW defined by the user. The function controlling this is listed in appendix A.22. In this function, six masses are used. There are two two-body decays involved, each with three masses. The two decays are the background candidate decay and the signal decay. The three masses in each decay are the mass of the mother particle (mm) and the masses of the two daughter particles (m1 and m2). The invariant mass of the signal decay, as given in equation 2.2 will be compared to the invariant mass of the misidentified decay, given in equation

2.3. For these invariant masses, the momenta and rest masses of the daughter particles are needed. The rest masses are given by the database. The momenta can be determined by looking at the center of mass of the two-body decay. In equation 4.3 the momentum of a daughter particle is given. The momentum of the other particle is always equal but opposite.

$$p1 = \frac{\sqrt{(mm^2 - (m1 + m2)^2)(mm^2 - (m1 - m2)^2)}}{2 \cdot mm} \tag{4.3}$$

Branching ratio If a branching ratio (BR) limit is defined, the BR of each candidate background will also have to be determined. In the eventual simulation of the background (section 4.4) the BR will be given in the legend of each background. Therefore this final check is in the `MisID` function (appendix A.16) itself. To find the BR of a candidate, the decay channels of each particle are needed. These are accessed using the ROOT database. The function doing so is listed in appendix A.23. This function goes through all the channels of the mother particle and checks if there is a channel with 2 daughters that have the exact pdg code of the daughter particles. If it finds such a channel, the BR is listed with this channel. If no channel was found, the BR is set to 0. Note that it might also occur that the channel was found with the BR already set to 0. This function does not distinguish such channels from decays not in the database. It is not known whether 0 BR channels refer to theoretically possible, but undetected, channels, or whether these refer to very small BR.

Once all constrains are applied, the background candidates that are still left are accepted as peaking background for the given signal, in the given search region. To visualizer these backgrounds a simulation of the misidentification is performed.

4.4 Simulating peaking backgrounds

With the functions presented in this chapter, all peaking backgrounds of any decay can be found. To most informative way to present such results would be a histogram, such as figure 4.1. Although simulation of events is not really the scope of this program, a invariant mass spectrum of each peaking background is also generated. This can be used to visualize and validate the resulting backgrounds

The function in A.13 both simulates the event and constructs a representative mass spectrum. To simulate particle decay's the ROOT class TGenPhaseSpace [37] is used. This can generate any decay given the mass of the mother particle, the number of daughter particles and the masses of the daughter particles. From this, the Lorentzvectors of each daughter particle can be extracted. Three phenomena are taken into account before reconstructing the event again. Firstly, the decay is simulated in the rest frame of the particle. To go through a more realistic scenario, a boost of that frame is performed. This will be discussed below. Secondly, the misidentification takes place. This is done by redefining the energy of each particle and assigning the wrong mass, as shown in equation 4.4.

$$E = \sqrt{mass_{mis}^2 + p_x^2 + p_y^2 + p_z^2} \tag{4.4}$$

Here, $mass_{miss}$ is the mass that is wrongly assigned to the particle. $p_{x,,y,z}$ gives the momenta of the particle in all directions. Note that these momenta are determined by the *actual* mass of the particle. Therefore the new energy is a mix of correct and mis-identification. The last phenomenon that is applied to the particle are the detector effect discussed in section 4.1.

Mass reconstruction The event is reconstructed using the mass and momenta of the two daughter particles. The mass of the particle has already been (mis)identified in the energy, as mentioned above. The energy of the mother particle can then be reconstructed by adding the energies of the two particles:

$$E_{mother} = E_{daughter1} + E_{daughter2}. \tag{4.5}$$

Note that this reconstructed energy is still a frame dependent property and not yet the invariant mass. The momentum of the mother particle is also needed to reconstruct the mass of the mother particle. This is constructed by adding the momentum vectors of the daughter particles

$$\vec{p}_{mother} = \vec{p}_{daughter1} + \vec{p}_{daughter2}. \tag{4.6}$$

The mass of the mother particle can then be found as

$$m_{mother} = \sqrt{E^2_{mother} + |\vec{p}_{mother}|^2}. \tag{4.7}$$

Which corresponds to the invariant mass of the two daughter particles. The number of decay that are simulated can be chosen in the function listed in A.15, as described in section 4.2.

Histogram For each decay, the reconstructed mass value is put into an histogram. This will be the eventual result of each peaking background. In section 4.5 some examples can be seen. In such a histogram the signal, in black, is shown together with the peaking background, in red. In the legend the BR for each peaking background can be found. As the peak of the signal might shift due to Bremsstrahlung effects, a line is given at the mass of the signal mother particle. This line illustrates how close the peaking background peaks to the desired reconstructed mass value. Each histogram will be assigned a title telling which peaking background is shown. This title will have the following form: "Reconstructed Invariant Mass Spectrum of [signal mother] to [signal daughter1] and [signal daughter2] misidentified as [background mother] to [background daughter1] and [background daughter2]". Here, the names in [] brackets are filled in by the program. To improve the ease at which the title can be read, the function in appendix A.24 makes sure that all Greek letters are localized. For example, the first letter of the Greek alphabet will be α instead of alpha. Because the title is written in LaTeX format, this function also makes sure all lower cases (_), which determine subscript in LaTeX, are replaced by hyphens (-).

Boost Appendix A.25 lists the function used to generate a distribution for the boost of the particles. This is because the particles are usually not in rest when detected. To account for the relativistic velocities of the particles the Lorentz boost (γ) distribution is characterized by the representative function $e^{\frac{1}{\gamma-10}}$. This function is chosen purely to have a distribution of relativistic particles with $\gamma > 10$. To apply this boost to the frame it is converted into $\beta = \frac{v}{c}$. Where v is the velocity of the particle and c the speed of light. This is done by

$$\beta = \sqrt{1 - \frac{1}{\gamma^2}}. \tag{4.8}$$

For the result this should not matter, as the physics that was applied is Lorentz invariant, as well as the reconstructed mass. A crucial reservation to make here is that misidentification typically does violate Lorentz invariance. This is because assigning the wrong mass to particles is not a physical process. Especially if there is an asymmetry in the misidentification, this effect can be noticeable. An example of this phenomenon is given in figure 4.6. This figure shows $K\pi$ misidentified as $e\mu$ at an invariant mass of about 9.3 GeV. No detector effects (i.e. uncertainty and Bremsstrahlung) are applied in this misidentification. However, at relativistic boosts, the δ function of the reconstructed mass smears out. This creates a long tail left of the signal invariant mass. This effect will also relocate the peaking value of the peaking background. For large γ values (>10) this peaking value will converge. When going from a γ of 10 to 100, the peak does not shift significantly anymore.

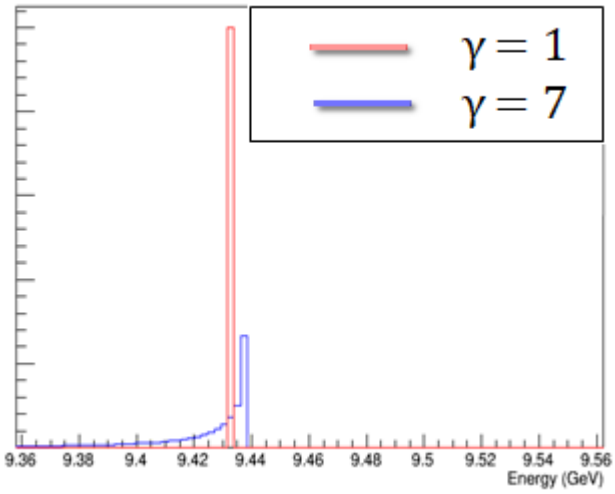


Figure 4.6: $K\pi$ misidentified as $e\mu$ at arbitrary invariant mass. Here, no measurement resolution or Bremsstrahlung is taken into account. The red line shows the misidentification in the rest frame. The blue line is the misidentification in a boosted frame.

This effect is also observed in the peaking backgrounds found in this function. It is not explored how this signal shape is effected by spin dependent decays. When locating a peaking background, this smearing is also not taken into account. Therefore, choosing a search region of a just few MeV's is not recommended. This is because backgrounds taken to be outside the search region might shift into the region due to this effect. At larger search regions, for example the default region of 0.1 GeV, the effect is not large enough to smear backgrounds into the search region.

BSM Signals The two body decay that is used as a signal does not have the limitations of the backgrounds. This does not have to be a SM allowed process. Examples of conservation laws that

can be violated in the signal include CP, lepton flavor and charge. For signals violating energy conservation, for example a photon decaying to two electrons, the invariant mass difference (appendix A.22) cannot be predicted. Therefore, all peaking backgrounds will be included, regardless of the defined search region.

Normalization The peaking backgrounds that are found are not normalized with respect to each other. To do so, more information per particle would be required. Examples are detector acceptance, to determine the likely hood of misidentification for each particle. Another example is the yield of each mother particle. In combination with the branching ratio of each channel these factors suppress the backgrounds found in this program. The resulting peaking backgrounds should therefore be treated as the worst case scenario.

4.5 Validation

Two case studies where performed to validate the results against larger simulations and data. These are the searches for D^0 to $e\mu$ and B_s^0 to $\mu\mu$. In this chapter, all parameter have the default value, as given in appendix A.15.

4.5.1 D^0 to $e\mu$

When running `MisID("D0", "e-", "mu+", "D0emu")` the resulting file gives 74 peaking backgrounds. As mentioned, note that this should be considered the worst case scenario. To be able to compare the results to the actual seach of D^0 to $e\mu$ [8], the branching ratio was limited. Of the 74 peaking backgrounds found, 2 have a nonzero branching ratio. These are D^0 to $\pi\pi$ and $\bar{D^0}$ to $\pi\pi$, both with a branching ratio of $1.397 \cdot 10^{-3}$. To find all peaking backgrounds, with no limit on branching ratio takes about 18 seconds. With the limit imposed, this is reduced to about 3.5 seconds. In figure 4.7 the resulting background peak is compared to the full LHCb simulation. Despite the simulation of the background not being the scope of this project, this comparison shows very good agreement.

4.5.2 B_s^0 to $\mu\mu$

In chapter 2.3 the decay $B_{(s)}^0$ to $\mu\mu$ (figure 2.7) was used to illustrate peaking and semi peaking backgrounds. In figure 4.8 the same decay can be seen, with the separate backgrounds identified. The peaking backgrounds in this picture are given by h^-h^+, where h can be either a pion or a kaon [7]. In addition, the two decays B_s^0 to $\mu\mu$ and B^0 to $\mu\mu$ can be considered each others peaking background. To check the peaking backgrounds, the function `MisID("B_s0", "mu+", "mu-", "Bmumu")` is run, with the search region given by $\Delta W = 0.2$ GeV. This function finds 97 peaking backgrounds. To constrain the number of backgrounds, the minimum branching ratio is set to a small number (for example: 10^{-20}). This filters out all decays with branching ratio equal to 0. This leaves 25 decays, listed in appendix B. As with the D^0 to $\pi\pi$ decay, described above, many backgrounds are listed with the mother both as particle and antiparticle. Also, $\bar{B_s^0}$ to $\mu^-\mu^+$ is considered a background. Therefore, only 12 of the 25 backgrounds have to be listed to give a complete overview. The remaining backgrounds are shown in table 4.3, with their respective branching ratios(BR). To find all backgrounds regardless of branching ratio takes about 26 seconds. With the

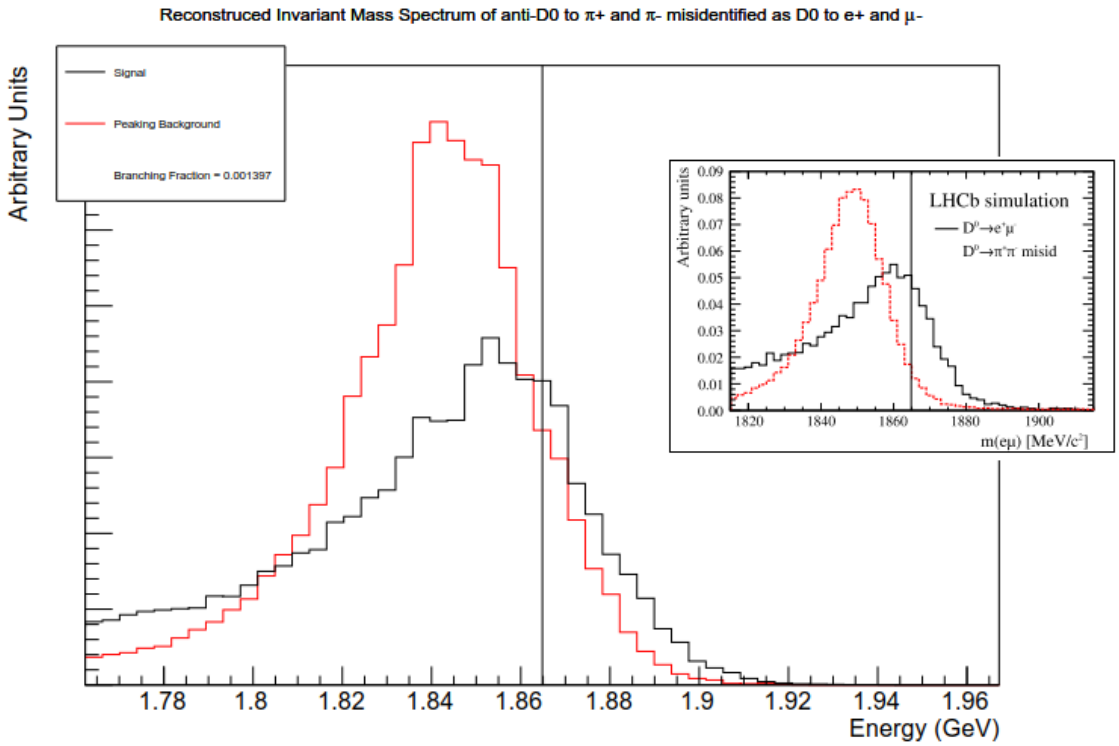


Figure 4.7: In red, the invariant mass spectrum of the decay D^0 to $\pi\pi$, the only peaking background found for D^0 to $e\mu$. In black, the invariant mass spectrum of D^0 to $e\mu$ correctly identified. The line gives the D^0 mass. In the inset, a full simulation of LHCb [8] is shown as validation.

constraint, finding the remaining backgrounds takes about 9 seconds.

When comparing these backgrounds to the spectrum in figure 4.8, the pink peak (B to h^-h^+) can be identified as B^0 to $K\pi$ or B_s^0 to KK. The reconstructed mass spectrum of these decays can be seen in figure 4.9.

The decay B^0 to $K\pi$ does not show up in the results. This is because this decay has not been identified yet. In figure 4.8 a hint to the existence of such a decay is hinted. In the results found, this process is one of the 72 decays with branching ratio equal to 0. In addition to the peaking backgrounds, the figure shows several semi peaking background. As can be seen, these originate from partially reconstructed decays. The following chapter will describe a function to identify such decays.

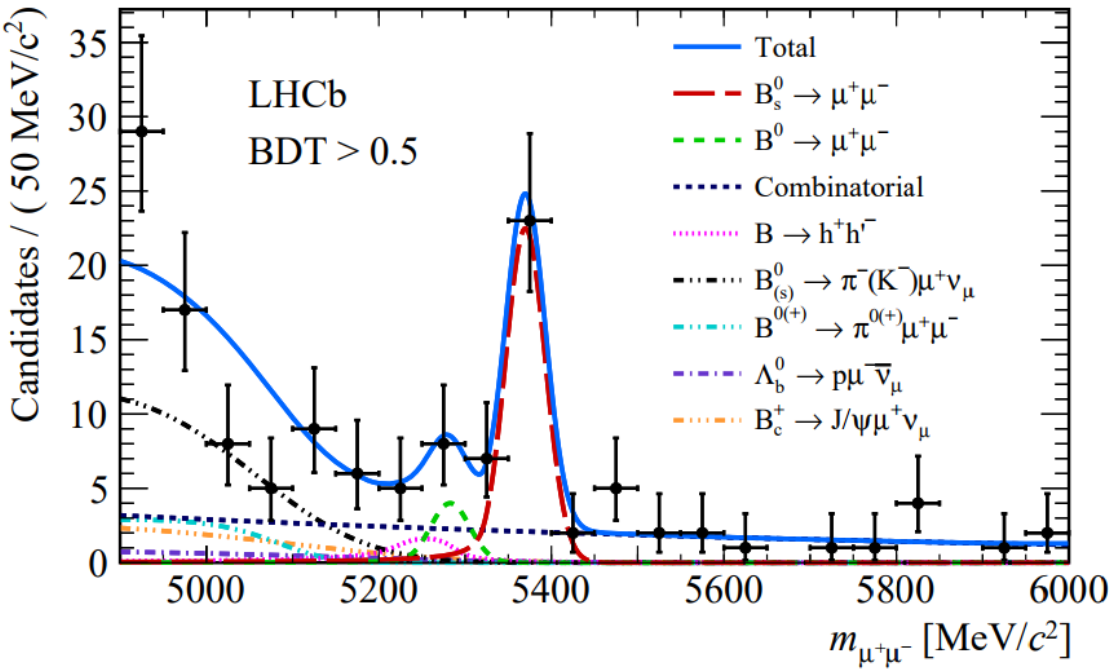
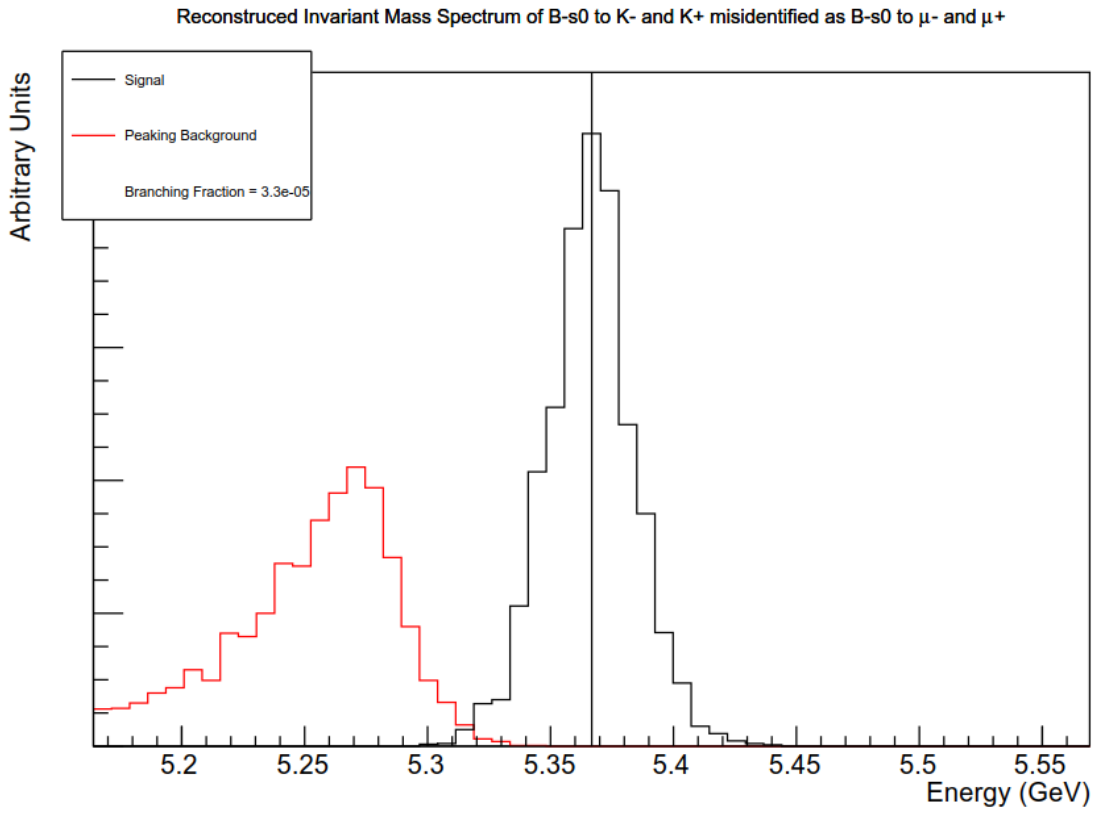


Figure 4.8: The reconstructed mass spectrum of the decay $B^0_{(s)}$ to $\mu\mu$. The individual backgrounds are identified in the spectrum. [7] This is the same decay as figure 2.7

Table 4.3: Peaking backgrounds of B^0_s to $\mu^-\mu^+$, as identified by the program

Decay	BR(10^{-6})
Λ_b^0 to $\bar{p}\pi^+$	3.8
Λ_b^0 to $\bar{p}K^+$	6
Λ_b^0 to K^-p	6
Λ_b^0 to π^-p	3.8
B_s^0 to $K^-\pi^+$	4.9
B_s^0 to K^-K^+	33
B_s^0 to $\pi^-\pi^+$	0.01
B_s^0 to π^-K^+	4.9
B^0 to $K^-\pi^+$	19.4
B^0 to K^-K^+	0.4
B^0 to $\pi^-\pi^+$	5.13
B^0 to π^-K^+	19.4



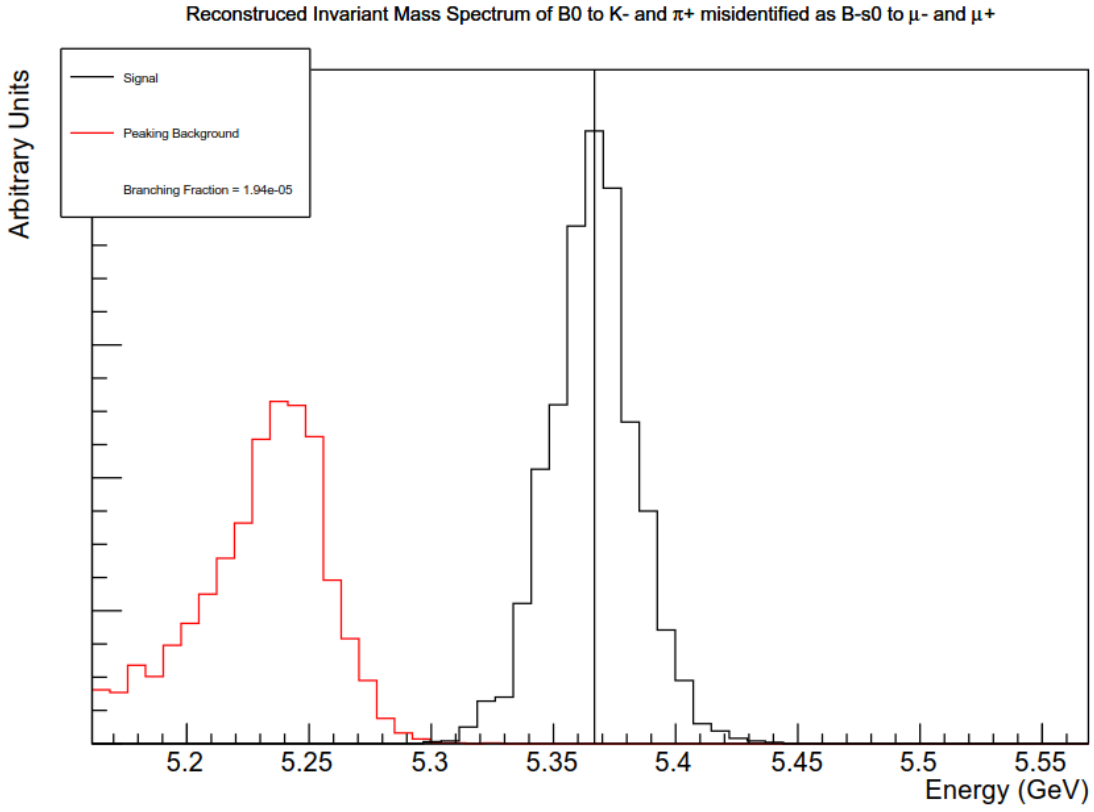


Figure 4.9: The two most likely candidate backgrounds for the peaking background B to h^-h^+, shown in figure 4.8. On the top is B_s^0 to KK, on the bottom B^0 to $K\pi$. In both cases, the kaons and pion are misidentified as muons.

Chapter 5

Semi peaking backgrounds

5.1 Partial reconstruction

At this point, there is a program to describe all peaking backgrounds given a specific decay. When looking at the reconstructed mass spectrum of B_s^0 to $\mu\mu$ (figures 2.7 and 4.8), many semi peaking backgrounds are shown. As discussed in chapter 2.3, these arise due to partially reconstructed decays. For example, two such events, identified in figure 4.8, are B^0 to $\pi^0\mu\mu$ and B^+ to $\pi^+\mu\mu$. In both cases the pion is not detected, causing the two muons to be identified as B to $\mu\mu$. In the general case, any number of particles might be undetected. The only requirement is that both daughter particles both have to be created.

The semi peaking background shown in figure 4.8 appear to be half of a peaking background. With the other half just outside the reconstructed mass region. When looking at the reconstructed mass of partially reconstructed events, it can be seen that these backgrounds are the endpoints of broader structures. For example, figure 5.1 shows the reconstructed mass spectrum of Λ_b^0 to $K_s^0 p\pi^-$. Here it can be seen that the begin and end points of the spectrum depend on which particle is undetected. This chapter will describe a function to find which partially reconstructed events appear in a given search region.

5.2 Cascades

Processes might occur where a particle decays into a particle with a very short lifetime. Once the secondary particle then decays, and the momenta of the daughter particles are reconstructed, the secondary decay vertex is indistinguishable from the production vertex. Such a stepwise decay is referred to as a cascade decay. It is possible to have a cascade of decays, in which the signal particles are among the end products. Figure 5.2 gives illustrations of two such events. In the reconstruction such events will look like partial reconstructions.

In this chapter, the function locating semi peaking backgrounds is described. This function will look at both the direct decays and the cascade decays. In contrast to the function seeking peaking backgrounds (chapter 4), which was based on the particles of the database, this function primarily

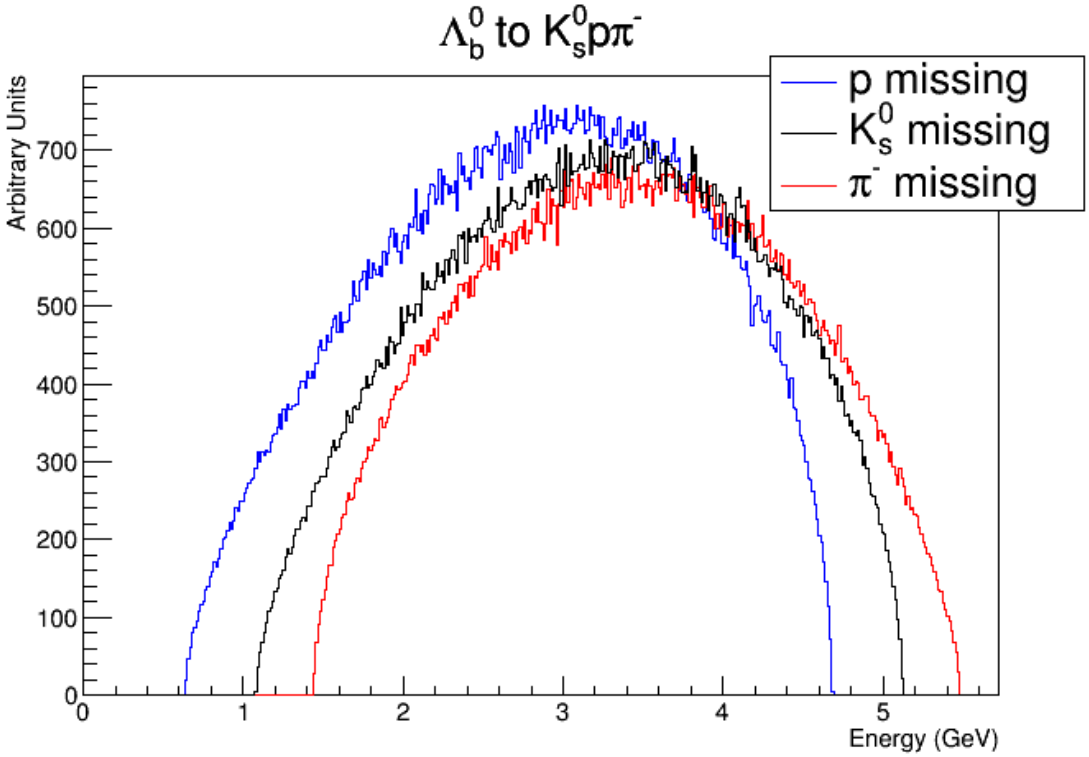


Figure 5.1: The partially reconstructed mass spectrum of Λ_b^0 to $K_s^0 p\pi^-$. There are 3 cases shown, in each case one of the daughter particles is not detected. The reconstructed mass of the two other particles give the shape of the spectrum.

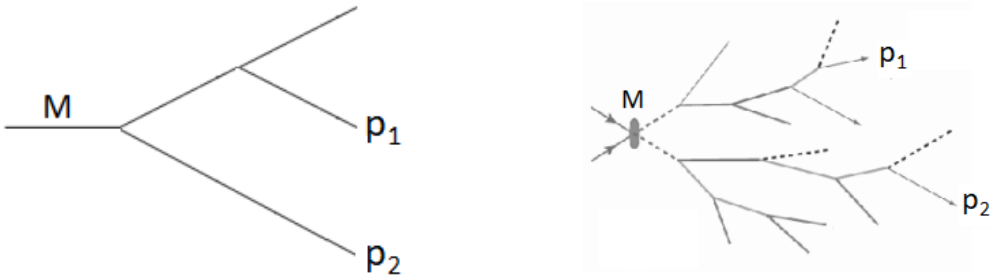


Figure 5.2: Illustrations of cascade decays in the search for a generic decay with daughters $p_1 p_2$. Here, M can be any SM allowed particle. On the left, the most simple cascade decay, where a primary decay gives p_2 and a secondary decay gives p_1. On the right, a very complicated series of decay is illustrated (picture from [38]), which eventually results in both p_1 and p_2. All end legs that are not assigned a particle illustrate undetected particles.

uses the decay channels. Because the spin of the particles is not used, the ROOT version of the rewritten database is used (see chapter 3.4.2). The direct decay can be found by simple checking these channels for both daughter particles. For the cascade decays, a more complicated approach is needed. For this approach, the function will be split up in two parts. First finding the first daughter particle. Secondly, if that has been found, finding the other daughter particle. The chapters below will describe this process. Alongside this description, the case study of a decay to μ^- and e^+ will feature in sections 5.3.1 and 5.4.1.

5.2.1 Cascade construction

The function seeking semi peaking backgrounds accepts any two body decay as signal decay, in both pdg codes as particle names. The function (appendix A.27) starts the same way as the peaking background function (chapter 4.3.1): by checking if the particle names or pdg codes exist is the database. All particles and decay channels in this function are accessed via the ROOT database. The heaviest daughter particle of the input decay is then chosen to be the first particle. This is done because, in general, the heavier particle will appear in less decay channels. Therefore, the maximum number of channels can be eliminated from the start. To be able to save a cascade, a cascade struct has to be defined. This struct is listed in appendix A.26. This contains another struct to describe each subsequent decay step of the cascade. The parameters of these structs are listed in table 5.1, and are explained in more detail below.

Table 5.1: Properties of a cascade object

Cascade properties	Step properties
List of steps	Mother particle
Number of undetected particles	Decaying daughter particle
Predicted invariant mass	Decay channel
Combined branching ratio	ID of channel in ROOT
Number of steps to reach p_1	
Number of steps to reach p_2	
Particle database	
Path to the database file	
Input decay	

Cascade object As can be seen, besides the cascade steps, a cascade object has three physics properties. Firstly, the number of undetected particles, which will be used as a stopping criterion in section 5.6. Secondly, a prediction on the invariant mass of the cascade, this is explained in detail in section 5.7. Thirdly, the combined branching ratio of all the channels in the cascade. The number of steps for the 2 particles, $p_{1,2}$, give the number of decay steps from the mother particle of the entire cascade down to the two eventually detected particles. The rewritten ROOT database is saved by both a pointer to the in-program database and the path where the .txt file that was used to access the database is located. Lastly, the decay that is given as input (the signal decay) is saved in each cascade object. When a new cascade object is created each property is set to 0, except for the branching ratio. This is set to 1 by default.

Cascade step object As mentioned, a list of steps is saved in each cascade object. For each step, a new struct is made, containing the information needed to simulate that step. The mother particle is the particle that decays in the specific step. The decaying daughter is the daughter particle which will either be detected or decay in the following step. The decay channel of the mother particle to the daughter particle is identified both by a pointer to the channel itself and by the ID which ROOT assigned to the channel.

Cascade parameters In chapter 4.2, several global parameters were defined which can be defined by the user. All of these parameters are also applied to the cascade backgrounds. In addition, there are two parameters specific to the cascade backgrounds, as can be seen in the function in appendix A.15. The first is a maximum lifetime. Below this value particles decay quick enough to appear as a single point decay. Any particle which lives longer than this value cannot decay in a cascade background. By default this is set to the same value as the minimum lifetime (1ns). This means particles will either reach the detector or decay. This lifetime can be decreased to exclude vertices which occur centimeters after the decay vertex. It might also be increased to include in-flight decaying particles, for example muons into electrons. The other parameter is the maximum number of particles missing. Section 5.6 described that a cascade background candidate is only included if 3 or less particles are undetected. Increasing this number will greatly increase the time needed to locate cascade backgrounds. However, decreasing it, shortening the time for the function, might disallow physical backgrounds.

Conceptual approach To locate all backgrounds, two recursive functions will be used. Each of these functions will seek one of the daughter particles of the signal decay. When both daughters are found, the resulting cascade will be written in a file. By writing the cascade to a file at the moment it has been found, the same cascade struct can be used to determine all cascades. This allows to gradually build and remove parts of the series of decays throughout the function. The resulting cascade file can be used to determine which cascades fall within the user defined constrains and can be considered a background.

5.3 Finding the first daughter particle

Mother particle candidates With a particle database, a cascade object defined, and a signal input the cascade can be build. Finding the first daughter particle starts by looking at potential mother particles. A priori, this can be any particle in the database. In the function listed in appendix A.27, this is given by the `find_mother` function. This function loops over all the particles in the database. This is done in the same way as the mapping of the ROOT database in chapter 3.3.1. For every particle, a check, called `c_condition`, is performed. This function is used to check if a particle can be a mother particle. It consists of 3 checks: a lifetime check, a BSM check and a mass check.

The lifetime of the mother particle is compared with the user defined maximum lifetime for cascade particles. If the mother particle lives too long to be part of a cascade decay, it is not considered a potential mother. Secondly, it is checked whether the particles are SM allowed. This is done in the same way as with the peaking backgrounds in chapter 4.3.2. The same function (appendix A.20)

is used, again excluding all non-SM particle types (see table 4.1). The last check is a comparison between the mass of the potential mother particle and the mass of the daughter particle. If the mass of the daughter particle is larger than the mass of the mother particle candidate, the latter is no longer considered a candidate.

Examine decay channels For each accepted mother candidate, the next step is looking at the decay channels. This is done in the next function, named `find_channels`. This loops over all the decay channels of the mother particle. For every individual channel, a new loop over the daughter particles of that channel is performed. For every daughter three checks are conducted.

The most important check is whether the daughter particle has the same pdg code as the signal daughter particle. In that case, the signal particle will be considered found. The second check is whether all particles in the decay are within the scope of this function. This is the same check, based on particle type, that was done for the mother particle. Lastly, the check if energy conservation is respected is performed. Despite all channels coming directly from one of the databases, some violate energy conservation. If the sum of the mass of all the daughter particle is larger than the mass of the mother particle, energy conservation is violated.

The function only continues if all three checks are passed. That is, if the daughter particle is found, there has been no energy violation, and all particles in the channel are within the scope of the function. Any decay channel passing all checks will trigger the next function. Such a channel will be referred to as a trigger channel.

Daughter particle found A channel triggering the function `found_product` will have to be added to the cascade object. As this object will eventually contain several channels, this does involve some bookkeeping on its properties (5.1). The trigger channel is added to the object as a cascade step object. Also, the number of undetected particles, the combined branching ratio and the number of steps have to be updated.

The number of daughters of the trigger channels, minus one, is added to the number of undetected particles. The one not undetected particle corresponds to the daughter particle that was searched for. The combined branching ratio is updated by multiplying it with the branching of the channel. For cascades with no channels, the default value was set to 1. Therefore, it will now be the branching ratio of the only channel it contains. The number of steps to get to the first daughter particle (p_1) is increased by 1. This corresponds to the channels that was added to the cascade.

The next step is to check if the channel also contains the other daughter particle of the signal decay. The process to find this particle is described in section 5.4. Any decays added to the cascade by this function will be removed after it finishes. The cascade will then again consist of the trigger channel

Mother particle candidates For each trigger channel, the function `find_mother` will be applied. This was the function used in the beginning of this chapter. By creating this recursion, the next channel of the cascade can be found using the previously defined functions. The entire function is reapplied with a new first daughter particle (p_1) defined. This new

daughter particle is the mother particle of the trigger channel. Because the function is recursive, the input particles in a given iteration of the function are not necessarily identical to the input particles of the cascade object. For the cascade this input is the signal decay, which does not change throughout the function.

With this recursion, any second iteration will find a mother particle for the mother particle of the daughter particle in the first iteration. Applying this multiple times, allows for a bottom up approach to finding any series of decays resulting in the signal daughter particle.

As explained in the end of chapter 5.2.1, only a single cascade object is used. A the function `found_product` can be triggered by several channels in the same mother particle, it is of vital importance to undo any changed in the cascade object. Not doing so would result in a cascade containing several channels which do not constitute a series of decays. Therefore, after the second daughter is searched for and the function is applied recursively, the last cascade step is removed from the cascade object. This also means all properties altered will have to be reset. Therefore, the number of steps and number of missing particle that were added before are subtracted again. Also the original branching ratio is reset. As some channels in the database have a branching ratio of 0, dividing by the branching of the trigger channel is not an option. Instead, this is done by saving the original value and reapplying it to the cascade.

5.3.1 Case study: $e^+\mu^-$

To illustrate the process, the search for a cascade resulting in $e^+\mu^-$ will be discussed. For a decay to $e^+\mu^-$, the muon is the heaviest particle of the two. Therefore, it will be classified as the first daughter particle. The next two steps involve checking all channels of the mother particle candidates. This case study focuses on one of the trigger channels for the muon: J/Ψ to $\mu^-\mu^+$, shown in figure 5.3. The μ^- of this channel is then the daughter particle. The first response is to check if the remaining daughter particles, in this case μ^+, of the trigger decay can decay into the other daughter particle of the signal, e^+. This happens with the function `find_daughter` (appendix A.28), and is discussed in section 5.4.1.

The second step is to recursively run the function. This time looking for a channel containing J/Ψ. One mother particle candidate for the J/Ψ is the B_c^+ meson. Figure 5.4 shows one of the trigger channels for this mother particle. Here, a B_c^+ decays into J/Ψ, e^+ and ν_e. Again, in section 5.4.1 the (in this case very simple) search for e^+ will be described. The search for a mother particle for B_c^+ will not be described here, but due to the high mass, not many trigger channels are expected. As mentioned in this chapter, after the recursion and searching for the second daughter particle, the last step is removed from the cascade. In this case the channel B_c^+ to J/Ψ, e^+ and ν_e. This leave the cascade with the channel J/Ψ to $\mu^-\mu^+$. Note that all cascades with the now deleted channel have already been found in the `find_daughter` function. The moment the function `found_product` ends, the loop in the function `find_channels` will go to the next channel. A second trigger channel for the B_c^+ can then be added to the cascade object. Figure 5.5 shows such a channel. In this instance B_c^+ decays into J/Ψ and D^+. Section 5.4.1 will also discuss getting e^+ from this D^+ meson.

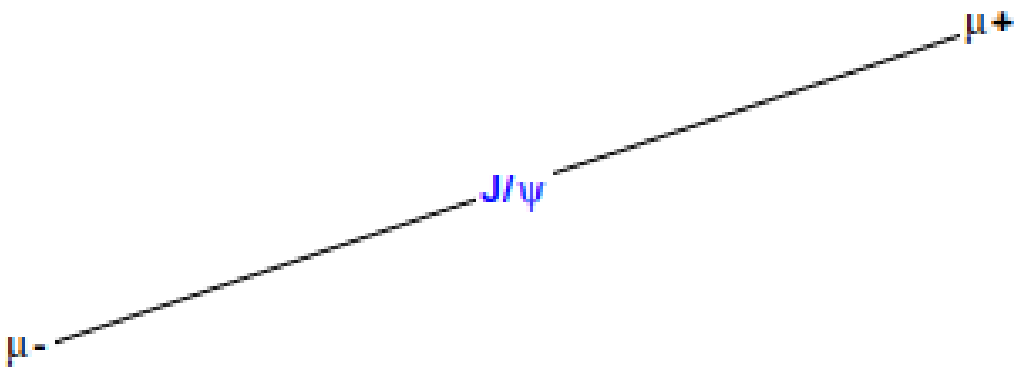


Figure 5.3: The decay channel J/Ψ to $\mu^-\mu^+$. This is one of the trigger channels for the search to any decay to $e^+\mu^-$

5.4 Finding the second daughter particle

As mentioned in section 5.3, for every trigger channel the function in appendix A.28 is applied. For such a channel, the first daughter particle of the signal decay will have been found. The next step is to examine of the second daughter particle is present in either the trigger channel, or in a cascade of the decay channel of one of the undetected products of the trigger channel.

The function consists of two parts, called `find_daughter` and `find_daughter_stepwise`. The first function, `find_daughter`, is the function that is applied in the function finding the first daughter particle. This function begins by subtracting 1 from the number of missing particles. This is to account for a second particle in the channel either decaying or being detected. Because of the same reason, the number of steps to the second daughter particle (p_2) is increased by 1.

Second daughter in trigger decay This first function is to check the simplest case, where the second daughter particle is produced in the triggered channel, alongside the first daughter particle. The first function loops over all the daughter particles of this channel. One of the arguments for this function is the number that is assigned to the first daughter particle in the list of decay products. This either corresponds to a detected particle, or a particle which has to decay, in order to create the first daughter particle. As a particle cannot decay twice or be detected twice, this function ignores the product at the given number. It is important to note, if the first daughter particle appears twice in the channel, that only one particle will be ignored. If the channel also contains the second daughter particle, a decay resulting in both particles has been found. If the daughter particle appear multiple times in the channel, the rest of the occurrences will be ignored. To add the second daughter particle to the cascade, a new cascade step is added, with this particle as the

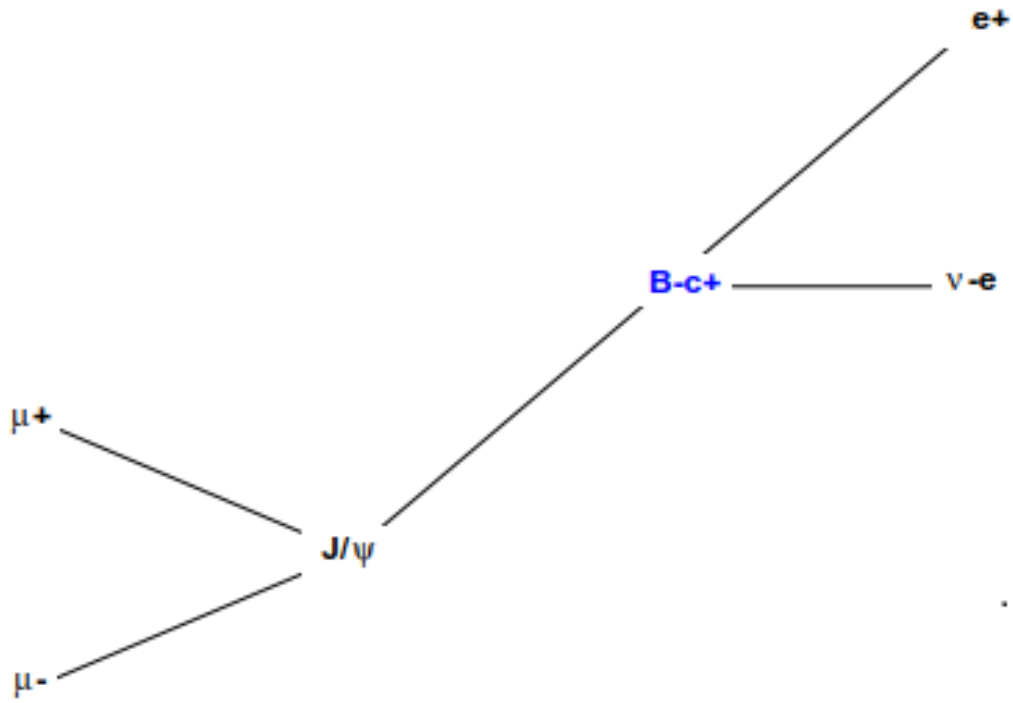


Figure 5.4: The decay channel J/Ψ to $\mu^-\mu^+$ is one of the trigger channels for the search to any decay to $e^+\mu^-$. This J/Ψ can be part of the cascade decay involving the decay channel B_c^+ to J/Ψ, e^+ and ν_e.

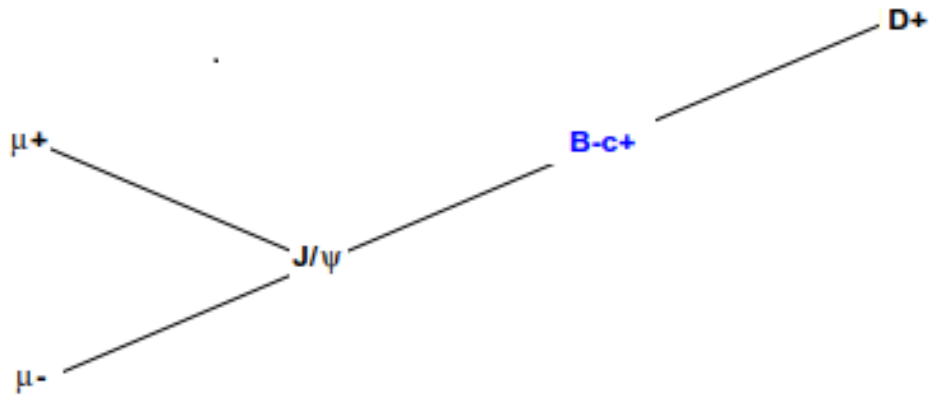


Figure 5.5: The decay channel J/Ψ to $\mu^-\mu^+$ is one of the trigger channels for the search to any decay to $e^+\mu^-$. This J/Ψ can be part of the cascade decay involving the decay channel B_c^+ to J/Ψ and D^+.

decaying daughter (see table 5.1). The trigger channel is therefore twice in any cascade object, once with the first daughter particle, and once with the second daughter particle. Since the number of steps and missing particle have been updated, no bookkeeping is required. The branching ratio does not have to be updated, as the channel was already in the cascade

This decay will then be written to a file, as described in section 5.5. For the easiest case, a decay which produces both signal daughter particles in a single channel, neither the function seeking the first daughter particle, nor the function seeking the second, will have to be applied recursively. After a linear search, the decay can be written to the file directly.

Second cascade It is also possible that one of the particles in the trigger channel can produce the second daughter particle by one or multiple decays. The approach for the first daughter particle was a bottom up approach where a loop over the entire database had to be performed. For the second one, a top down approach can be applied. This involves taking a single potential mother particle and looping over its channels. Every daughter particle in that channel can again be a potential mother particle. Therefore, every step is saved in the cascade object, before checking if the second daughter particle is in the channels.

For this the second function, `find_daughter_stepwise` is used. Before this function is applied, the check `c_condition` is applied. This check was already described in the beginning of section 5.3. In this case, it checks if any particle can be a potential mother for the second daughter particle. The function is first used to check the daughter particles in the trigger channel. For each potential mother, a new step is added to the cascade. The stepwise function is then applied. After this the step that was previously added is removed again, to allow for the next potential mother to be added.

The stepwise function looks at every decay channel of the decaying daughter of the last step added to the cascade. First, a series of tests are performed on each channel. These involve the maximum number of missing particles, energy conservation, a BSM check and whether the second daughter particle is in the channel. The first check can be performed without looking at the daughter particles of the channel. The only information needed, is the number of daughter particles. There are two ways to find the second daughter particle, either directly, or via a chain of decays. Either way, only one particle of the current channel will be detected. Note that, in this instance, a particle decaying in at least one detected particle is itself also counted as a detected particle. Therefore, the number of daughter particles, minus 1, is added to the number of missing particles in the cascade. If the new number of missing particle is more than the maximum allowed number, the specific decay channel can be ignored completely.

For the other checks, it is nessesary to loop over the decay products. While performing this loop, it is checked if any of the products have the same pdg code as the second daughter particle. Secondly, the particle type SM check (appendix 4.1), used several times before, is also applied here. Lastly, the masses of the decay products are not allowed to be larger than the mass of their mother particle. For a channel that passes the SM and energy conservation check, a new step can be added to the cascade. Therefore the missing particles are increased by the number of particles in the channel, minus the one detect red particle. Also, the branching ratio is updated by multiplying with the branching ratio of the channel.

To add the new steps to the cascade, a second loop will have to be performed over the channel. In this loop every product is examined. The function `c_condition` is again used to check if the mass of the potential mother is smaller than the mass of the second daughter particle. In addition, it is checked if the lifetime is small enough for the potential mother to decay within a cascade. For every product that passes these checks, a new step is added to the cascade. The stepwise function is then applied with the updated cascade function. This will check if the daughter particles of the mother candidate are themselves also mother candidates.

A new step is also created if, in the first loop, the second daughter particle was found. In this case a series of decay ending in both signal daughter particles has been found. Therefore, the cascade is considered found, and written to a file, as described in section 5.5. The next paragraph described how this function behaves in the case study of a decay to $e^+\mu^-$.

5.4.1 Case study: $e^+\mu^-$

The first part of this case study of cascades to $e^+\mu^-$ was given in section 5.3.1. The function seeking the second daughter particle was called three times in this paragraph. The first cascade object only consists of J/Ψ to $\mu^-\mu^+$ (figure 5.3), which is part of the the other cascades. These are: B_c^+ to $J/\Psi(\mu\mu)$, e^+ and ν_e (figure 5.4) and B_c^+ to $J/\Psi(\mu\mu)$ and D^+ (figure 5.5).

For the first decay decay, J/Ψ to $\mu^-\mu^+$, the μ^- has already been detected. Therefore, this function start by checking if the second daughter particle, which is e^+, is among the other decay products. This is not the case, as the only other daughter particle in this case is the μ^-. The next step is to check whether an e^+ can be produced in the decay channels of μ^+. It is possible

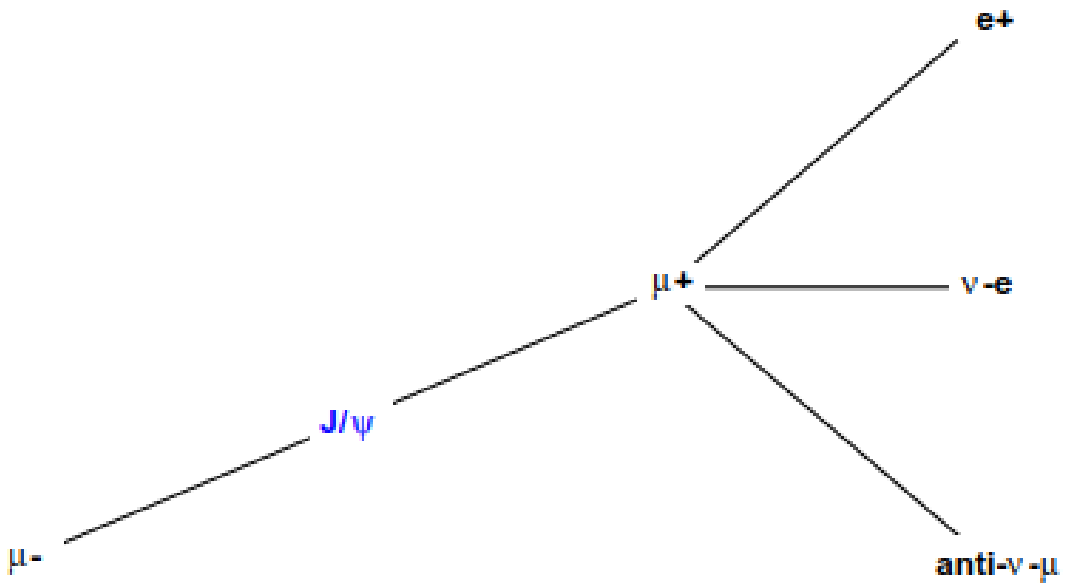


Figure 5.6: An example of a cascade resulting in $e\mu$. Here, a J/Ψ decays into two muons. Thereafter, one of the muons decays into an electron and two neutrino's. Note that this can only happen if time allows for the $/mu$ to decay.

to have a μ^+ decay into e^+, ν_e and $\bar{\nu_\mu}$. This would result in a cascade decay to $e\mu$, as illustrated in figure5.6. However, this can only happen if time allows for the μ to decay. This is given by the cascade lifetime, which can be adjusted, as described in section 5.2.1. By its default value, muons live too long to be able to decay before reaching a detector. However, to account for in flight decaying muon, this value might be raised above the muon lifetime ($2.2 \cdot 10^{-6}$ s).

The second decay, B_c^+ to $J/\Psi(\mu\mu)$, e^+ and ν_e, undergoes the same process. Here, the J/Ψ already decays into two muon. Therefore, is cannot also decay into electrons. The check to see if there is an e^+ among the other daughter particles is successful in this case. Therefore, the decay will be saved in a file (section 5.5). The neutrino is not checked for potential channels producing e^+. This is because it is apparent from the neutrino mass that it is impossible to produce an e^+. In the third decay, B_c^+ to $J/\Psi(\mu\mu)$ and D^+, the only other daughter particle is a D^+. As the e^+ is not found, the channels of the D^+ are checked to see if it can be produced. One of the decay channels which produces e^+ is the decay into e^+, ν_e and π^0. With the D^0 decaying via this channel, another cascade to $e\mu$ is created, seen in figure 5.7. Once this cascade is written to the file, the next step is to check the other particles of the channel for decays to e^+. This is done to represent a situation

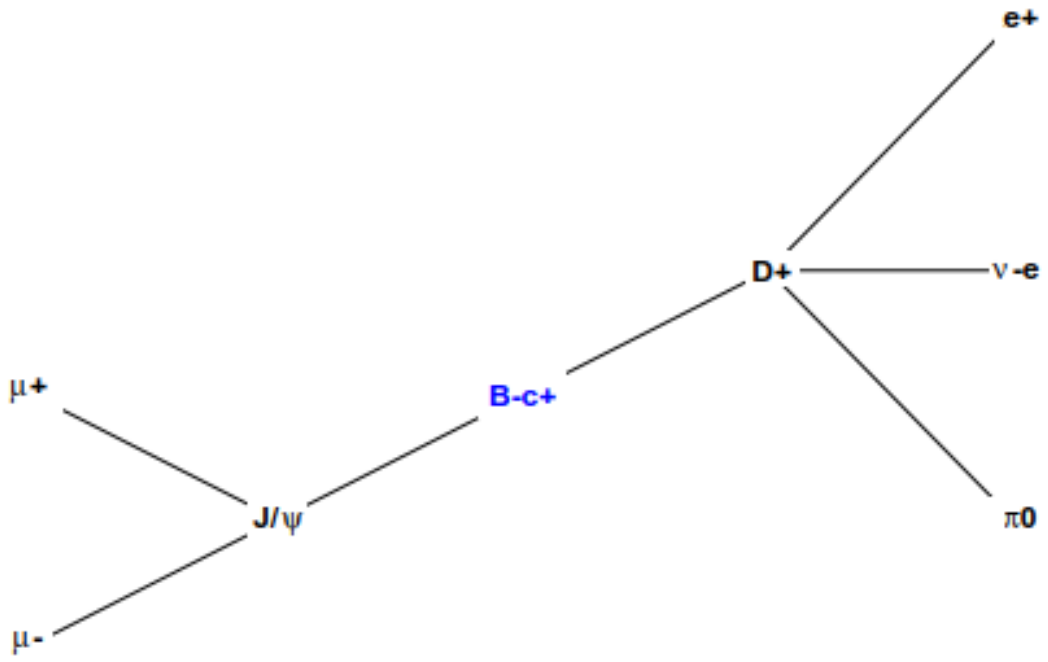


Figure 5.7: An example of a cascade resulting in $e\mu$. Here, a B_c^+ decays into a J/Ψ and a D^+. Thereafter, the J/Ψ produces the μ, and the D^+ the e.

where the first e^+ is not detected. As with the previous decay, the neutrino cannot produce an e^+. However, the pion has a decay channel to e^-, e^+ and γ. the resulting cascade is shown in 5.8. Note that the default number of maximum undetected particles is 3. As this cascade has 5 undetected particles, it will not be considered a significant background. This will be explained in more detail in section 5.6.

5.5 Writing and reading cascades

The function to both write cascades to a file, and read these from the same file, is given in appendix A.29

5.5.1 Writing cascade file

As mentioned several times in section 5.4, a cascade that has been found will be written to a file. This allows one single cascade object to be used throughout the function. This object will be build up, written to a file and broken down again. This file is meant to be the intermediate step between the program finding all cascades resulting in the desired daughter particles and the determination

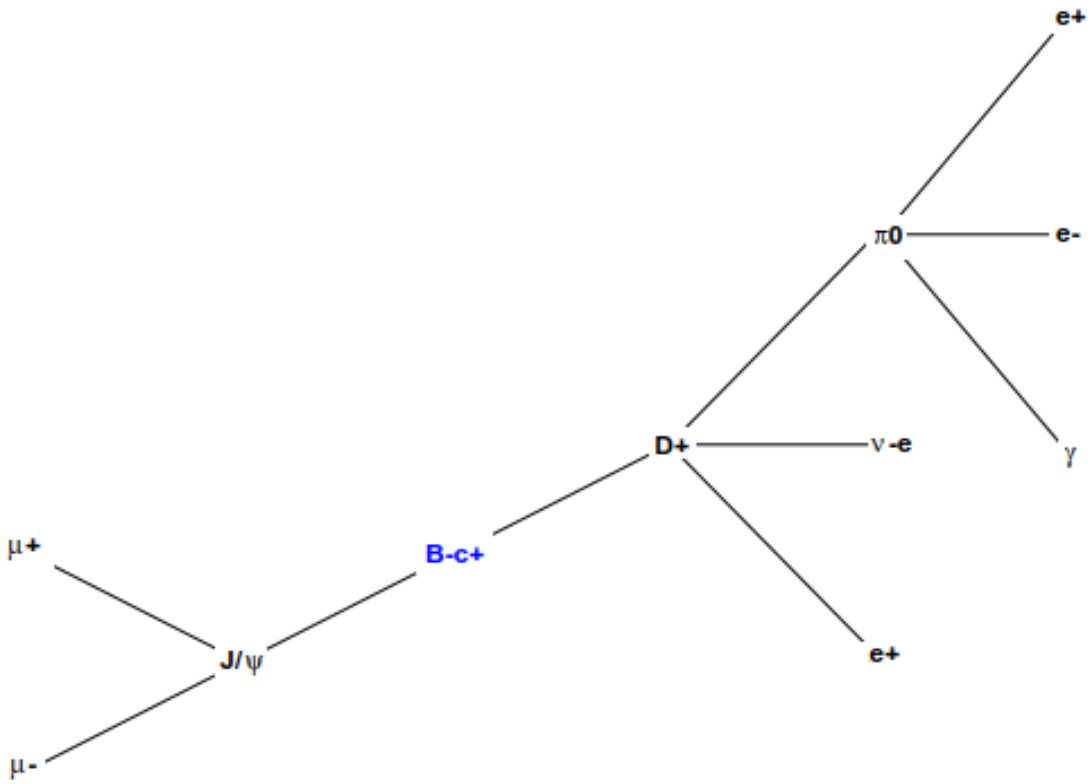


Figure 5.8: An example of a cascade resulting in $e\mu$. Here, a B_c^+ decays into a J/Ψ and a D^+. Thereafter, the J/Ψ produces the μ, and the D^+ produces a π^0 decaying into the e. Note that the e^+ created in the decay of D^+ to π^0 is assumed undetected in this case.

```
<Cascade id="6628" Pmiss="3" W="4.980511" BR="2.16148e-08" p0="511" p1="13" p2="-11" DBpath="ParticleDataSets/Rewrite/RewrittenDatabase.txt" >
<Channel myid="1" ch_id="4" mother="411" pdgdaughters=" -11 12 111 " dec_daughter="-11" />
<Channel myid="0" ch_id="57" mother="541" pdgdaughters=" 411 443 " dec_daughter="411" />
<Channel myid="0" ch_id="57" mother="541" pdgdaughters=" 411 443 " dec_daughter="443" />
<Channel myid="-1" ch_id="1" mother="443" pdgdaughters=" -13 13 " dec_daughter="13" />
</Cascade>
```

Figure 5.9: The cascade B_c^+ to $J/\Psi(\mu\mu)$ and $D^+(e^+\nu_e\pi^0)$ saved in the cascade file. This is the cascade seen in figure 5.7 and table 5.2

of the selection of cascades which are significant to the user. The name of the file can be user defined, but by default it is cascade_tmp. Note that only using the default name will overwrite the file each time the cascade function is run.

The process of finding cascades, as described in sections 5.3 and 5.4, gives a cascade object in the following form. First the channel producing the first daughter particle is saved, then the channel which produces the mother particle of this daughter particle. Up to the eventual mother particle of the cascade. Then the exact same decay channel of the mother particle is used to check if another daughter particle might produce the second signal daughter particle.

Cascade object from $e\mu$ case study An example of the structure is given here using the cascade in figure 5.7. This is a cascade to $e\mu$ Here, B_c^+ decays into a J/Ψ and a D^+. Thereafter, the J/Ψ produces the μ, and the D^+ produces a π^0 decaying into the e. This cascade will have the structure given in table 5.2. Each channel is accompanied by the particle that is either detected or decays in the next step. This particles is the decaying daughter.

Table 5.2: Structure of the cascade object B_c^+ to $J/\Psi(\mu\mu)$ and $D^+(e^+\nu_e\pi^0)$

Decay channel	Decaying daughter
$D^+ \to e^+\nu_e\pi^0$	e^+
$B_c^+ \to J/\Psi D+$	D^+
$B_c^+ \to J/\Psi D+$	J/Ψ
$J/\Psi \to \mu^+\mu^-$	μ^-

The cascade file is written in a XML format, like the Pythia database (chapter 3.2.2). Here each cascade object consists of a cascade node with several channel (cascade step) nodes. Each cascade node holds the properties to recreate the cascade object. See table 5.1 for the properties of this object. A node contains the number of missing particles, the branching ratio, a predicted invariant mass (see chapter 5.7), the path to the used database and the pdg code of the signal decay particles. In addition every cascade has an ID. This is the number of cascades in the file after that cascade was added. This allows to identify individual cascades in the file. Each channel node contains the ID ROOT had assigned to the channel, the pdg codes of the mother and daughter particles and which of the daughter decays in the following cascade step. In addition, every channel has an ID corresponding to the number of decay steps to get to that channel. For the first decay, this is 0 for the first channel of the cascade. 1, 2, 3 etc. gives decays towards the first daughter particle, whereas negative numbers indicate the path to the second daughter particle. Figure 5.9 shows how the cascade from figure 5.7 and table 5.2 would look like in the file.

5.5.2 Reading cascade file

It is essential to be able to recreate the exact cascade object by reading the file again. For most of the properties, this is a simple matter of added them again to a new cascade object, this creating a list of cascade objects. The signal particles can be found again by their pdg codes and the path to the database. For the channel nodes, two loops are needed. The first loop builds a step object for each channel node. With the pdg codes of the mother and decaying daughter, in combination with the channel ID in the ROOT database, each step can be created. The cascade is now build in reverse order compared to how it was saved to the file. Therefore, the second loop reverses the entire cascade, getting the original back.

5.6 Stopping criteria

As mentioned in section 5.2, one of the properties of each cascade is the maximum number of undetected particles. By default this was set to 3 undetected particles. Section 5.4 already showed a cascade with more (figure 5.8). Figure 5.10 shows how the reconstructed mass spectrum of a decay to $e\mu$ changes when there are undetected particles. As observed, already with 2 missing particles, the function starts to become much smoother. With 3 missing particles there are already virtually no events in the search region. With 4 missing particles this would be much less. Therefore, 3 is set as the maximum number of undetected particles. Even if the shape of the invariant mass spectrum would change, for example due to polarization of particles, it is unlikely that the spectrum of 4 undetected particles would change enough to create peak like events in the search region.

The parameters reducing the number of partially reconstructed background candidates can be chosen by the user in the program in appendix A.15. These are the number of missing particles, the maximum lifetime of a particle in the cascade, the minimum branching ratio and the search region. The number of particles and the lifetime constraints are applied while building the cascade. Increasing these value will increase the time needed for all the cascades to be found.

The other constraints, branching ratio and search region, are applied after all the cascades have been found. When reading the cascade from the file to visualize the results (section 5.8), a selection is made based on these parameters. Note that the minimum branching ratio is the same parameter that was used in the peaking backgrounds (chapter 4). This means, by default, this value is set to -1. As all branching ratios are positive values, every cascade will be included. Also note that there exists decays in the database with branching ratio equal to 0. Therefore, every positive limit will exclude some cascades. It is not known what these 0 branching decays represent. The last constraint is the reconstructed mass search window. This is defined by the signal mother particle and the ΔW defined in the parameters file. The reconstructed mass looks at the maximum invariant mass of the partially reconstructed event. This is discussed in more detail in section 5.7.

5.7 Invariant mass spectrum

Already, in figure 5.1 the shape of the reconstructed mass spectrum of a partially reconstructed even is shown. There are three distinct areas in such a picture. These areas are illustrated in figure 5.11. These three regions will be referred to as the begin (5.11a), middle (5.11b) and end (5.11c)

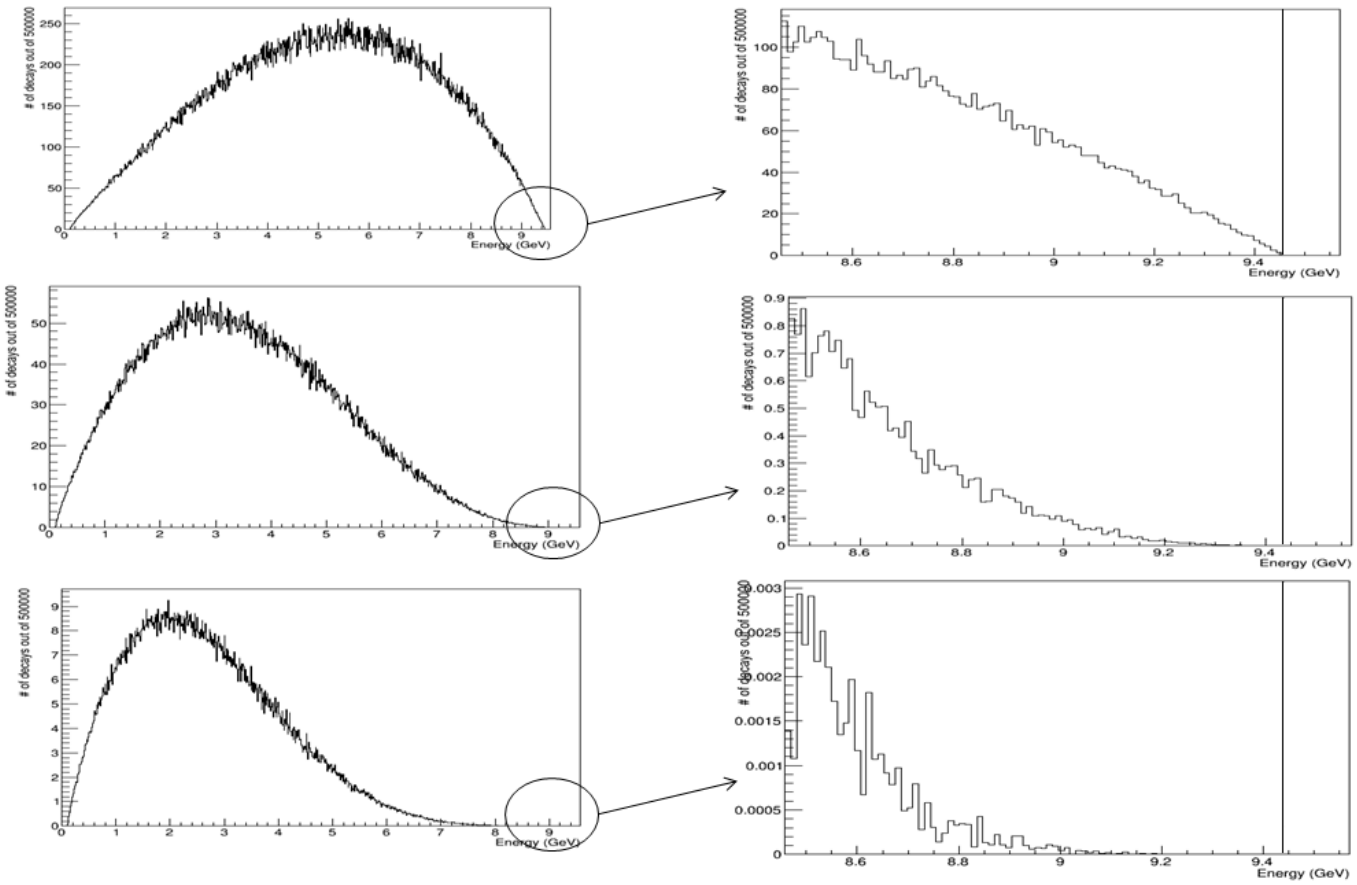


Figure 5.10: The change of the invariant mass of $e\mu$ as a function of the number of undetected particles. On the right the reconstructed mass spectrum from 0 to an arbitrary search mass is shown. On the right the search region around that mass is shown. The top two pictures show the reconstructed mass spectrum when one massless particle is undetected from the $e\mu$ decay. The middle two pictures shown the $e\mu$ reconstructed mass when two massless particles are undetected. The bottom case is 3 undetected particles. As can be observed, already at 3 undetected massless particles, there are virtually no events in the search region.

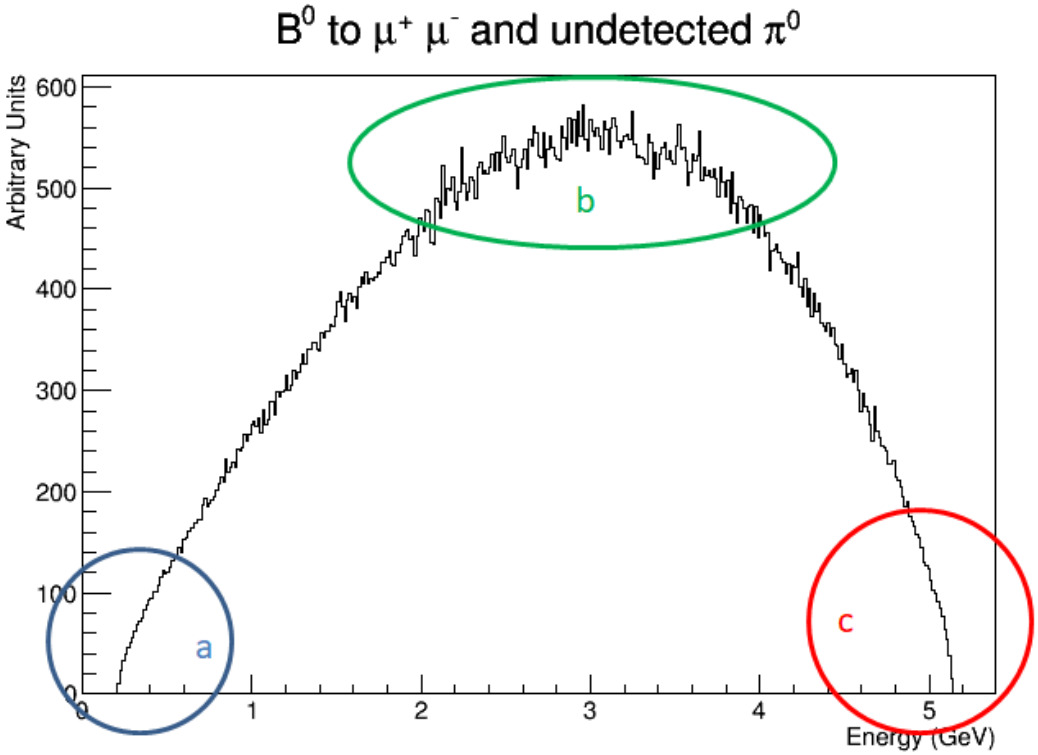
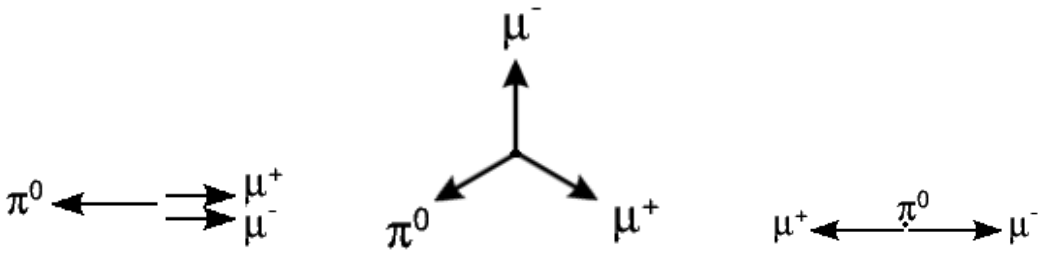


Figure 5.11: The invariant mass of B^0 to $\mu^+\mu^-\pi^0$, where only the two muons are detected. Three regions of a partially reconstructed mass spectrum are identified: the begin (a), the middle (b) and the end (c)

of the spectrum.

This figure shows the example of B^0 to $\mu\mu$ with a undetected π^0. When comparing to the spectrum of B^0 to $\mu\mu$ that was shown before (figure 4.8), it can be seen that the end of the cascade spectrum can appear in the invariant mass spectrum. In figure 5.12c the three point vertex of this area of the cascade reconstructed mass spectrum is shown. Here, the undetected pion is virtually at rest, in the rest frame of the B^0. Therefore, all available energy is in the dimuon energy. The reconstructed mass will then be given by the mass of the B^0 minus the mass of the π^0

If the search region is defined to be at the middle of the cascade spectrum, the background, due to the partially reconstructed events, will appear smooth. Therefore, when parameterizing the combinatorial background (chapter 2.3) this will also take into account this smooth background of the cascade middle. The three point vertex of the middle part (figure 5.12b) is given by a approximately equal distribution of the momentum of the three particles. With each having about a third of the (invariant) mass of the B^0, and the pion undetected, the reconstructed mass is at about $\frac{2}{3}$ of the B^0 mass (5.28 GeV).



(a) begin (b) middle (c) end

Figure 5.12: Illustration of the three point vertices of the beginning (a), middle (b) and end (c) of a partially reconstructed mass spectrum. These are the regions identified in figure 5.11

The final region is the begin of the spectrum. When looking at the mass spectrum, this can look the reverse of the end. Therefore, if observed, it would also be a semi peaking background. However, as shown in figure 5.12a, this background appears if the two muons have the same momentum, both in magnitude and direction. When reconstructing these muons, in the center of mass frame, it appear they have been created at rest. As this is a unphysical situation, this region is not treated as a possible background.

5.7.1 Calculating cascade endpoint

In section 5.2.1, it was mentioned how every cascade has an invariant mass value. This value determined which backgrounds fall in the user defined search region. As explained in this section, the only notable point of the cascade is the endpoint of the reconstructed mass spectrum. Therefore, the invariant mass of a cascade is given by the reconstructed mass of the endpoint of that cascade. The invariant mass can only be calculated when the entire cascade has been build. When writing a found cascade to the file (section 5.5), the invariant mass is calculated and added to the file.

The function calculating this value is given in appendix A.30. This function works by reducing the cascade to a 1 dimensional series of two body decays. Each two body decay will have a mother particle, two daughter particles and a boost. From this input, a Lorentzvector for the first daughter particle is created. This first daughter particle will be the particle named decaying daughter in section 5.2.1. The two body decay is applied for each step of the cascade. The process of calculating the momentum and energy of a two particle decay has been described in chapter 4. Equation 4.3 is again used for the momentum, and equation 5.1 is used to calculate the center of mass energy for p_1.

$$E_{CM,p_1} = \frac{mass^2_{mother} + mass^2_{p_1} - mass^2_{p_2}}{2 \cdot mass_{mother}} \tag{5.1}$$

The momentum and energy of the particle are then boosted by the given factor β, from which the Lorentzfactor γ can be calculated. The boosted energy and momentum are given by equations 5.2 and 5.3.

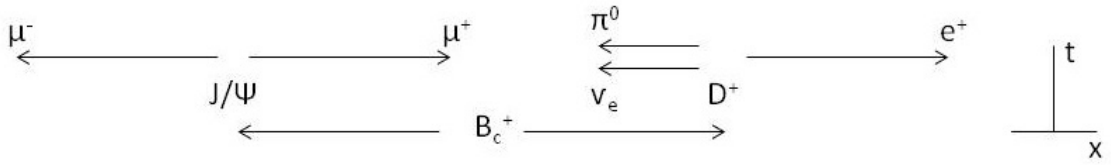


Figure 5.13: The one dimensional approximation to the cascade B_c^+ to $J/\Psi(\mu\mu)$ and $D^+(e^+\nu_e\pi^0)$. Here μ^- is produced with its momentum in the $-x$ direction. e^+ is produced in the $+x$ direction. The vertical direction shows the order of the decays. As can be observed, the direction of each detected daughter particle is in the direction of flight of its mother particle.

$$E_{p_1} = \gamma \cdot (p_{CM,p_1} + \beta \cdot E_{CM,p_1}) \tag{5.2}$$

$$p_{p_1} = \gamma \cdot (E_{CM,p_1} + \beta \cdot p_{CM,p_1}) \tag{5.3}$$

This two body decay can then be used to examine the maximum reconstructed mass of a cascade. Next is a description of how it is used in the special case of the mother particle decay, and the generic case of a decaying daughter.

When maximizing the reconstructed mass, the mother particle of the cascade will decay as shown in figure 5.12c. This can be approximated by a two body decay. In this approximation, the mother particle mass is reduced. The new mass is given by subtracting the mass of the undetected particles from the mass of the mother. By letting a particle with this new mass decay, the Lorentzvector of the two detected daughter particles can be found. This is done separately for each of the two daughter particles. These vectors are then used to go through the cascade steps towards p_1 and p_2. The endpoint of a cascade is given by the point where the energy of the two daughter particles in maximized. In the calculation, each decay is given in the center of mass frame, and subsequently boosted. The boost of the decay is given by the boost of the mother particle. To calculate the maximum energy in each step, the decaying daughter particle has a momentum in the opposite direction of all the undetected particles. As an illustration, think of rocket propulsion, which works by emitting particles in the direction opposite to the direction of travel. The decay of every step is given by a two body decay of the mother particle of that step. The first daughter particles is given by the particle whose Lorentzvector is to be determined. The other daughter particle consists of the sum of all undetected particles in the decay. Figure 5.13 shows an illustration of the 1D decay approximation of the cascade B_c^+ to $J/\Psi(\mu\mu)$ and $D^+(e^+\nu_e\pi^0)$, which was shown in figure 5.7.

5.8 Visualizing cascades

When all cascades have been found and written to the file (section 5.5), the next step is to select which backgrounds are of interest. This is done by the search region and the minimum branching ratio. Both parameters are defined by the user. Every cascade which is in the search region and above the minimum branching ratio is given as a result. There are two ways to visualize this result. The first is the illustration which has featured in this chapter (figures 5.3-5.8). The function for this illustration is given in appendix A.31 The second method is to create a invariant mass spectrum of

the cascade. Although this projects includes a function for that simulation (appendix A.32), it is advised to use external programs for this.

5.8.1 Illustrating cascades

When visualizing a collection of cascades, the name of the new file has to be given as input. In addition, a choice whether to illustrate or simulate the cascade has to be made. With simulation = 0, there will be no simulation, with 1 there will be a simulation. The last input is the filename of the file containing the cascades. By default this was named "cascade_tmp". If a new filename was chosen when making the cascade file, it has to be an input value here.

All cascades which are in the user defined boundaries will be illustrated using the function `printfullcascade` in appendix A.31. For this visualization, a grid, with step size 100px is constructed on a canvas. The size of this canvas is given by the number of missing particles and the number of steps in the cascade (the size of the cascade). Following the cascade structure shown in table 5.2, the function starts with the first value in the list of cascade steps. This value will be the step added last. This will contain the final decay to the second daughter particle. The first daughter particle is placed one grid point, both vertically and horizontally, from the upper right corner. the other particles from the channel each a grid point below this. For the next decay, the decaying daughter particle will be the particle decaying into the second daughter particle. Therefore, this particle has to be placed in the same way as the daughter particle. It will be placed one grid point away, in the horizontal direction. The height is given by the number of daughter particles in its decay. For the subsequent decays this can be repeated, each moving one grid point horizontally, thus setting up the series of decays.

The mother particle of the cascade is recognized due to the fact that it appears twice in the cascade (see table 5.2). When encountering this particle the illustration function goes to the "tipping point". At this point, the bottom up approach of the second daughter particle has been drawn. For the first daughter particle, a top down approach is needed. Note that this is the reverse of the building of the cascade (sections 5.3 and 5.4). With the mother particle, also the number of missing particles, invariant mass and branching ratio are given. The mother particle also contains a daughter particle which will eventually decay into the first daughter particle. This has to be drawn one grid step to the left. The path to the first daughter particle is drawn similar to the path to the second. The difference is that with the first daughter particle the decaying daughter is drawn first. Then the products of its decay are drawn one grid point away. The last step of the cascade is the decay to the first daughter particle.

5.8.2 Outlook: cascade simulation

The function to simulate a cascade decay (appendix A.32) was used to test prediction on the maximum invariant mass of a cascade. Although given in the program, the simulation of event is not within the scope of this project. When having located all backgrounds with this program, it is advised to use specialized programs to simulate the decays. However, for simulation of a single decay to visualize the reconstructed mass spectrum, the function can be used.

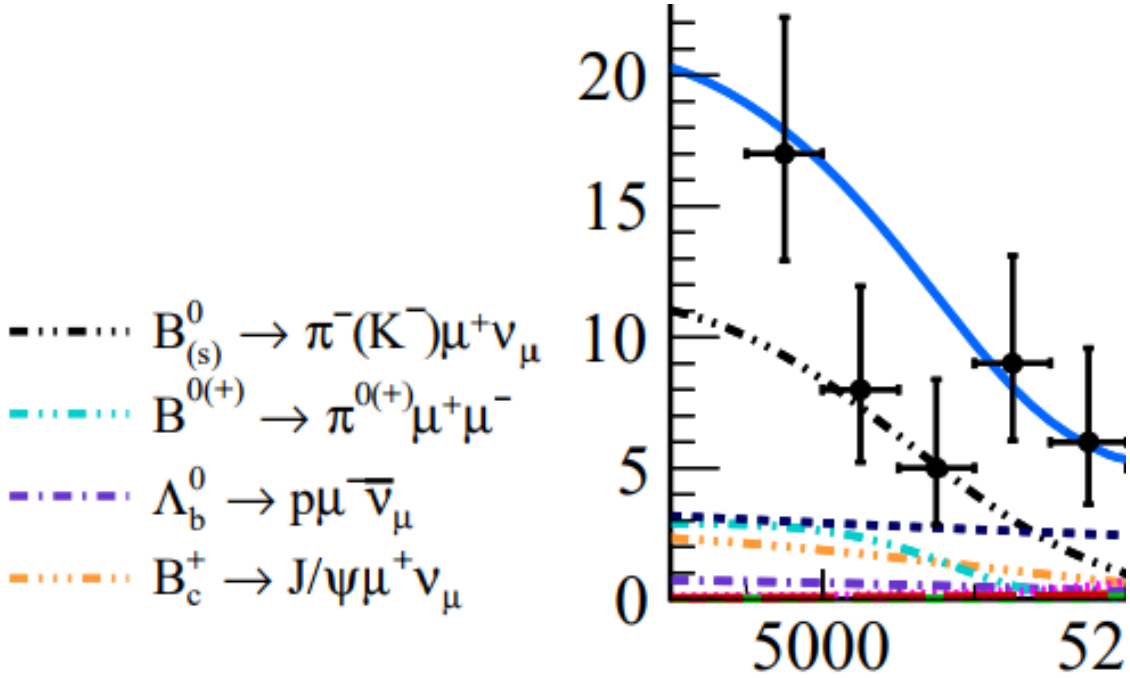


Figure 5.14: A zoom of figure 4.8, focusing on the partially reconstructed backgrounds of that figure.

5.9 Validation

To verify the backgrounds found with the function seeking semi peaking background, the decays $B^0_{(s)}$ to $\mu\mu$ is again used. This is the same decay used to test the peaking backgrounds (chapter 4.5 and figure 4.8). In this decay, several partially reconstructed events are identified. Figure 5.14 shows a zoom on the partially reconstructed events of figure 4.8.

To find these backgrounds, the functions `find_cascade("B0", "mu+", "mu-")` and `visualizeCascade("mumu", 0)` are run. These will find all backgrounds to $\mu\mu$ and make an illustration of every background within the user defined constraints. To match the LHCb analysis, the minimum branching ratio was set to 10^{-8}. The search region was given by $\Delta W = 0.2$ GeV. The other parameters were at their default values. To find all cascades to $\mu\mu$ takes about 10min and 37s. To illustrate all backgrounds in the search region takes about 5 seconds. The resulting backgrounds are given in table 5.3. Here, the branching ratio is given by the Branching column. Rec. Mass gives the maximum reconstructed mass value of the cascade.

By far the most significant background found in this analysis is the decay $B^+_c \to J/\Psi(\mu^-\mu^+) + \mu^+ + \nu_\mu$. This decay has a branching ratio of the order of 10^{-3}, whereas all other backgrounds are of the order 10^{-8}. When looking at the reconstructed mass spectrum (figure 5.14), this background can indeed be observed. However, two more backgrounds are not found by the program. These

Table 5.3: Cascade backgrounds resulting in $\mu^+\mu^-$, as identified by the program

Cascade decay	Branching (10^{-8})	Rec. Mass
$B^- \to \eta(\mu^+\mu^-\gamma) + \mu^- + \bar{\nu_\mu}$	1.14	5.15 GeV
$B^- \to \omega(\mu^+\mu^-) + \mu^- + \bar{\nu_\mu}$	1.04	5.17 GeV
$B^- \to \eta'(\mu^+\mu^-\gamma) + \mu^- + \bar{\nu_\mu}$	2.94	5.16 GeV
$B_c^- \to J/\Psi(\mu^+\mu^-) + \mu^- + \bar{\nu_\mu}$	$1.13{\cdot}10^5$	5.46 GeV
$B^+ \to \eta(\mu^-\mu^+\gamma) + \mu^+ + \nu_\mu$	1.14	5.15 GeV
$B^+ \to \omega(\mu^-\mu^+) + \mu^+ + \nu_\mu$	1.04	5.17 GeV
$B^+ \to \eta'(\mu^-\mu^+\gamma) + \mu^+ + \nu_\mu$	2.94	5.16 GeV
$B_c^+ \to J/\Psi(\mu^-\mu^+) + \mu^+ + \nu_\mu$	$1.13{\cdot}10^5$	5.46 GeV

are the decays $B^{0/+} \to \mu^+\mu^-\pi^{0/+}$. When checking the databases, it was found these decays do not exist in either the ROOT, Pythia, or EvtGen databases. Checking the PDG for and B^+ [39], gives a branching fraction of $1.79 \cdot 10^{-8}$ for $B^+ \to \mu^+\mu^-\pi^+$. Why this decay is not in any of the databases is not known. Checking for B^0 [40], gives no branching fraction. For $B^0 \to \mu^+\mu^-\pi^0$ only a limit is defined, at $8.4 \cdot 10^{-8}$. It is assumed that the lack of a definitive branching ratio is the reason this decay is not in any of the databases.

Two other backgrounds are identified in figure 5.14. They are $B^0_{(s)} \to \pi^-(K^-)\mu^+\nu_\mu$ and $\Lambda_b^0 \to p\mu\bar{\nu_\mu}$. These backgrounds are a combination of misidentification (as $\mu\mu$) and partial reconstruction. Due to time limitations, these kind of backgrounds are not included in the function finding semi peaking backgrounds. Knowing the particles which can be misidentified as muon (i.e. K, π, p), it is possible to construct these event. For example, the background $B^0 \to \pi^-\mu^+\nu_\mu$ can be found by searching for semi peaking backgrounds resulting in $\pi^-\mu^+$. When these kinds of backgrounds are properly included, it will be possible to locate all backgrounds of a given two body decay.

Chapter 6

Conclusion

The aim of this projects was to construct a tool to make fast predictions on backgrounds in two body decays. To achieve this, a complete particle database was created (chapter 3) to locate peaking (chapter 4) and semi peaking (chapter 5) backgrounds.

Comparison with several existing studies shows good agreement in the peaking backgrounds (see 4.5). With several seconds needed for these predictions, the tool is considered fast. Semi peaking backgrounds originating in both direct and cascade decays were included in the search for semi peaking backgrounds. With computation times of the order of 10 minutes, the tool is not as fast in finding semi-peaking backgrounds. However, this is still much faster than the existing tools. Repeating the analysis on the same signal decay can be done in a matter of seconds. Analysis of the semi peaking backgrounds (see 5.9) revealed missing backgrounds, due to these decays channels not being present in any database, or being the result of a combination of partial reconstruction and misidentification. The semi peaking backgrounds which were found by the program did agree with backgrounds found in the comparison studies.

With this tool, almost all expected backgrounds of a given two body decay can be located. It is considered a fast tool, with a computation time ranging from seconds to tens of minutes.

Bibliography

[1] A. Pich, "Precision tests of the standard model," in *Fundamental physics: Selected topics on high-energy and astroparticle physics. Proceedings, 25th International Winter Meeting, Formigal, Spain, March 3-7, 1997*, pp. 1–30, 1997.

[2] M. W. Grunewald, "Precision tests of the standard model," *PoS*, vol. HEP2005, p. 306, 2006.

[3] S. Chatrchyan, V. Khachatryan, A. M. Sirunyan, A. Tumasyan, W. Adam, E. Aguilo, T. Bergauer, M. Dragicevic, J. Erö, C. Fabjan, and et al., "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC," *Physics Letters B*, vol. 716, pp. 30–61, Sept. 2012.

[4] "Proton beams are back in the LHC — CERN." `http://home.cern/about/updates/2015/04/proton-beams-are-back-lhc`. Accessed: 19/07/2017.

[5] "LHCb Starter kit, First Steps in LHCb." `https://lhcb.github.io/first-analysis-steps/`. Accessed: 19/07/2017.

[6] S. A. et al, "Geant4a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250 – 303, 2003.

[7] M. Mulder, "The branching fraction and effective lifetime of $B^0_{(s)} \to \mu^+\mu^-$ at LHCb with Run 1 and Run 2 data," 2017.

[8] R. Aaij *et al.*, "Search for the lepton-flavour violating decay $D^0 \to e^\pm\mu^\mp$," *Phys. Lett.*, vol. B754, pp. 167–175, 2016.

[9] G. Burdman, E. Golowich, J. L. Hewett, and S. Pakvasa, "Rare charm decays in the standard model and beyond," *Phys. Rev.*, vol. D66, p. 014009, 2002.

[10] C. Langenbruch, "Rare decays at LHCb," in *Proceedings, 51st Rencontres de Moriond on QCD and High Energy Interactions: La Thuile, Italy, March 19-26, 2016*, pp. 175–178, ARISF, ARISF, 2016.

[11] P. Koppenburg, Z. Dolezal, and M. Smizanska, "Rare decays of b hadrons," *Scholarpedia*, vol. 11, no. 6, p. 32643, 2016. revision #154125.

[12] The LHCb Collaboration, "The LHCb Detector at the LHC," *Journal of Instrumentation*, vol. 3, no. 08, p. S08005, 2008.

[13] "LHCb Calorimetes." https://lhcb-public.web.cern.ch/lhcb-public/en/Detector/Calorimeters-en.html. Accessed: 20/07/2017.

[14] V. Khachatryan *et al.*, "Observation of the rare $B_s^0 \to \mu^+\mu^-$ decay from the combined analysis of CMS and LHCb data," *Nature*, vol. 522, pp. 68–72, 2015.

[15] R. Aaij *et al.*, "Production of J/psi and Upsilon mesons in pp collisions at sqrt(s) = 8 TeV," *JHEP*, vol. 06, p. 064, 2013.

[16] C. Patrignani et al. (Particle Data Group), "REVIEW OF PARTICLE PHYSICS," 2016.

[17] "ROOT a Data analysis Framework." https://root.cern.ch/. Accessed: 28/06/2017.

[18] "PYTHIA." http://home.thep.lu.se/Pythia/. Accessed: 28/06/2017.

[19] "EvtGen Project." http://evtgen.warwick.ac.uk/. Accessed: 28/06/2017.

[20] "ROOT:TDatabasePDG Class Reference." https://root.cern.ch/doc/master/classTDatabasePDG.html. Accessed: 28/06/2017.

[21] "ROOT:TParticlePDG Class Reference." https://root.cern.ch/doc/master/classTParticlePDG.html. Accessed: 28/06/2017.

[22] "TParticlePDG Lifetime() problem - ROOT Forum." https://root-forum.cern.ch/t/tparticlepdg-lifetime-problem/8548. Accessed: 28/06/2017.

[23] "GitHub: ParticleData.xml." https://github.com/bbannier/pythia8/blob/master/xmldoc/ParticleData.xml. Accessed: 28/06/2017.

[24] "The Particle Data Scheme." http://home.thep.lu.se/~torbjorn/pythia81html/ParticleDataScheme.html. Accessed: 28/06/2017.

[25] "TXMLEngine Class Reference." https://root.cern.ch/doc/v608/classTXMLEngine.html. Accessed: 28/06/2017.

[26] "Source Repository - EvtGen Project." http://evtgen.warwick.ac.uk/src/R01-06-00/. Accessed: 28/06/2017.

[27] "std::map cpppreference." http://en.cppreference.com/w/cpp/container/map. Accessed: 28/06/2017.

[28] K. D. Lane, "An Introduction to technicolor," in *Theoretical Advanced Study Institute (TASI 93) in Elementary Particle Physics: The Building Blocks of Creation - From Microfermius to Megaparsecs Boulder, Colorado, June 6-July 2, 1993*, pp. 381–408, 1993.

[29] "MONTE CARLO PARTICLE NUMBERING SCHEME." http://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf. Accessed: 28/06/2017.

[30] "IBM Knowlegde Center: Delimited Flat Files." https://www.ibm.com/support/knowledgecenter/en/SS6QYM_9.2.0/com.ibm.help.fnd.config.doc/r_DelimitedFlatFiles.html. Accessed: 28/06/2017.

[31] "std::mismatch cpppreference." http://en.cppreference.com/w/cpp/algorithm/mismatch. Accessed: 28/06/2017.

[32] "GitHub - bbannier/pythia8 at master." https://github.com/bbannier/pythia8/tree/master. Accessed: 28/06/2017.

[33] "Releases — ROOT a Data analysis Framework." https://root.cern.ch/releases. Accessed: 28/06/2017.

[34] "TRandom Class Reference." https://root.cern.ch/doc/master/classTRandom.html. Accessed: 11/07/2017.

[35] S. Kirwan, "Development of the Selection Procedure for a Measurement of the Ratio of Branching Fractions $R_{pK} = \mathcal{B}(\Lambda_b^0 \to pK\mu^+\mu^-)/\mathcal{B}(\Lambda_b^0 \to pKe^+e^-)$ as a Test of Lepton Universality," Master's thesis, University of Southampton, 2016-05-15.

[36] "fstate - final state search engine." http://fstate.org/. Accessed: 11/07/2017.

[37] "TGenPhaseSpace Class Reference." https://root.cern.ch/doc/v608/classTGenPhaseSpace.html. Accessed: 11/07/2017.

[38] D. d'Enterria, "Physics at the LHC: A Short overview," *J. Phys. Conf. Ser.*, vol. 270, p. 012001, 2011.

[39] "Particle Data Group: B+." http://pdg.lbl.gov/2017/listings/rpp2017-list-B-plus-minus.pdf. Accessed: 17/07/2017.

[40] "Particle Data Group: B0." http://pdg.lbl.gov/2017/listings/rpp2017-list-B-zero.pdf. Accessed: 17/07/2017.

[41] "Visual Studio." https://www.visualstudio.com/. Accessed: 19/07/2017.

Appendices

Appendix A

Program functions

This appendix contains the functions of the developed tool. The tool is programmed using ROOT, a c++ based software framework [17]. Visual Studio [41] was used to compile the code. A flowchart of the functions is shown in figure A.1. In addition there are several global functions. These functions do not belong specifically to the database, peaking or semi-peaking backgrounds. These are: `detect`(A14), `globals`(A15) `filename`(A17), `titleName`(A24), `beta_distribution`(A25) and `myparticlestructs`. The latter contains `FParticle`(A3), `FParticle_maps`(A4), `bType`(A20) and `FCascade`(A26).

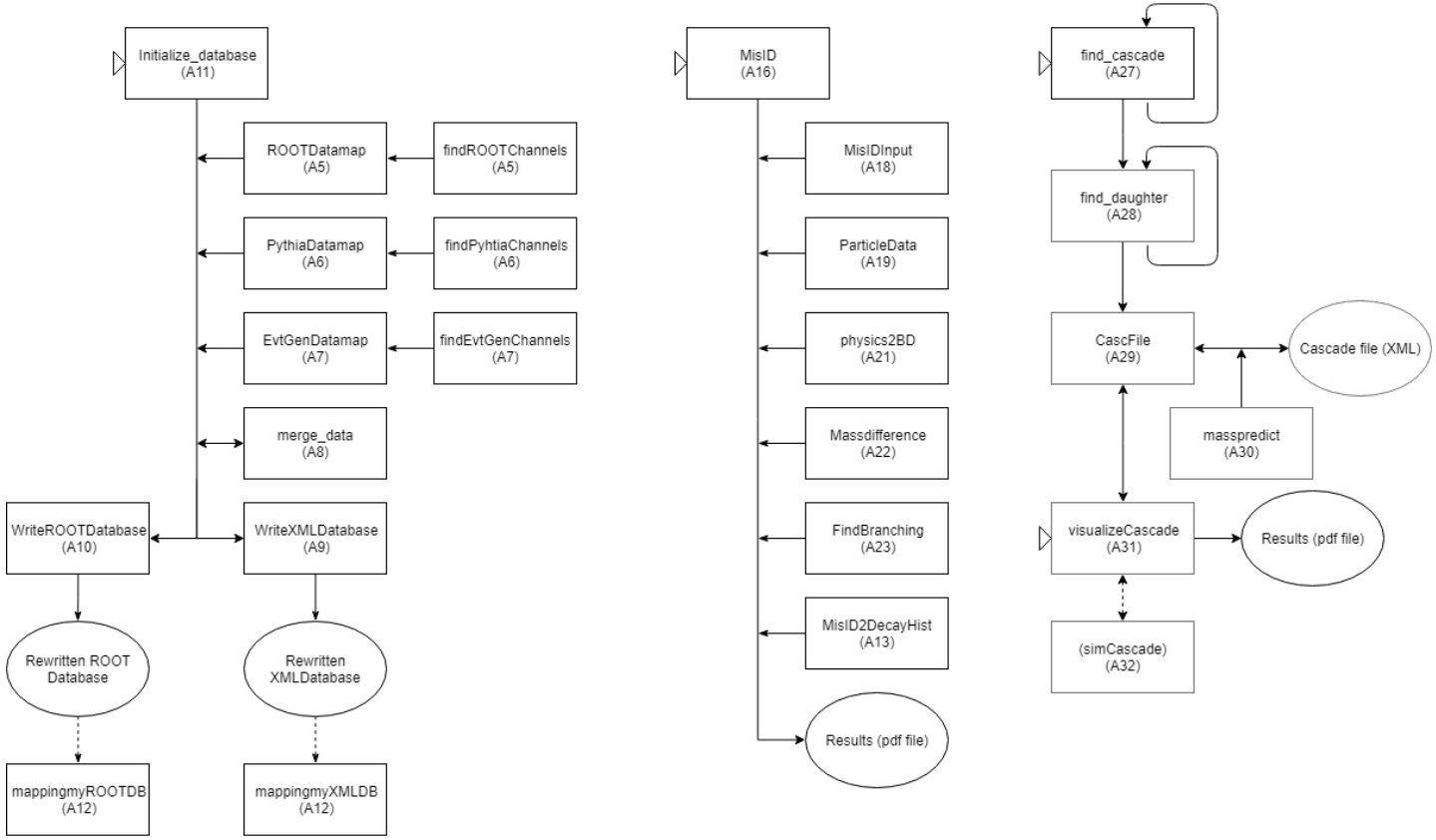


Figure A.1: A flowchart of the functions in this appendix. A triangle indicates user input. In addition to the functions in the flowchart, there are several global functions. These are used in all three sections shown.

A.1 ROOTcount

A program that is used to count the particles in the ROOT database given certain properties.

```
void ROOTcount () {
  TDatabasePDG* pdg = TDatabasePDG::Instance();
  TParticlePDG* particle = 0;
  TIter next(pdg->ParticleList());
  int npart = 0;
  while(particle=(TParticlePDG*)next())
  {
      /* Possible if statement here defining property that is being counted */
    npart++; /*counting all particles*/
  }
  cout << npart;
  delete pdg;
}
```

A.2 Pythiacount

A Program used to count particle without lifetimes in the Pythia database

```
void Pythiacount() {
  double tau0, width;
  int npart, notau, nowidth;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleData.xml");
  XMLNodePointer_t xmlnode = engine->DocGetRootElement(doc);
  xmlnode = engine->GetChild(xmlnode);
  while (xmlnode)
  {
    if (TString(engine->GetNodeName(xmlnode)).EqualTo("particle"))
    {
      npart++;
      tau0 = TString(engine->GetAttr(xmlnode, "tau0")).Atof();
      if (!tau0)
      {
        notau++;
        width = TString(engine->GetAttr(xmlnode, "mWidth")).Atof();
        if (!width)
        {
          nowidth++;
        }
      }
    }
    xmlnode = engine->GetNext(xmlnode);
  }
  cout << "Total number: " << npart << ". No lifetime: " << notau << ". Also, no width: "
       << nowidth;
  delete engine;
};
```

A.3 FParticle

The particle struct that is used throughout this project.

```
struct FChannel;

struct FParticle {
  int pdgCode;
  TString tName;
  double Mass;
  int Charge3e;
  double Spin;
  double Lifetime;
  std::vector<FChannel*> Channels;
  TString tType;
  int Colortype;
  bool _scopeFlag;
  FParticle() :_scopeFlag(0) {}
};

struct FChannel {
  double Branchingratio;
  FParticle* Motherparticle;
  std::vector<FParticle*> Daughterparticles;
  bool _chFlag;
  FChannel() : _chFlag(0) {}
};
```

A.4 Particle maps

The two maps made for each databse.

```
struct FParticle_maps {
  std::map<int, FParticle*> map_pdg;
  std::map<TString, int> map_names;
};
```

A.5 ROOTmap

Various functions used to map the ROOT database particles and channels

ROOTDatamap.cpp :

```
#ifndef ROOTDATAMAP_CPP
#define ROOTDATAMAP_CPP

#include <map>
#include <vector>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "THashList.h"

#include "globals.hpp"
#include "myparticlestructs.hpp"
#include "findROOTChannels.hpp"

#include "ROOTDatamap.hpp"

void fillChannels(std::map<int, FParticle*> ROOTmap, TDatabasePDG* pdg);
void PopulateParticleROOT(TParticlePDG* t_particle, FParticle* oParticle);
double FindROOTLifetime(TParticlePDG* t_particle);
void name_handler_ROOT(FParticle* oParticle, TString tName);

FParticle_maps ROOTDatamap() {
  FParticle_maps mapsROOT;
  std::map<TString, int> mapNames;
  std::map<int, FParticle*> mapParticles;
  TDatabasePDG* pdg = TDatabasePDG::Instance();
  pdg->ReadPDGTable();
  TParticlePDG* t_particle = 0;
  TIter next(pdg->ParticleList());
  while (t_particle = (TParticlePDG*)next())
  {
    FParticle* oParticle = new FParticle();
    PopulateParticleROOT(t_particle, oParticle);
    mapParticles.insert(std::pair<int, FParticle*>(oParticle->pdgCode, oParticle));
    mapNames.insert(std::pair<TString, int>(oParticle->tName, oParticle->pdgCode));
  }
  fillChannels(mapParticles, pdg);
  mapsROOT.map_names = mapNames;
  mapsROOT.map_pdg = mapParticles;
  delete pdg;
  return mapsROOT;
};

void fillChannels(std::map<int, FParticle*> ROOTmap, TDatabasePDG* pdg) {
  FParticle* oParticle;
  TParticlePDG* t_particle;
  for (auto const &ent1 : ROOTmap)
  {
    oParticle = ent1.second;
    t_particle = pdg->GetParticle(ent1.first);
    oParticle->Channels = findROOTChannels(oParticle, t_particle, ROOTmap);
  }
};

void PopulateParticleROOT(TParticlePDG* t_particle, FParticle* oParticle) {
```

```
  oParticle->pdgCode = t_particle->PdgCode();
  oParticle->Mass = t_particle->Mass();
  oParticle->Charge3e = (int) t_particle->Charge();
  oParticle->Spin = t_particle->Spin();
  oParticle->Lifetime = FindROOTLifetime(t_particle);
  oParticle->Colortype = 0;
  oParticle->tType = TString(t_particle->ParticleClass());
  name_handler_ROOT(oParticle, TString(t_particle->GetName()));
};

double FindROOTLifetime(TParticlePDG* t_particle) {
  double Lifetime = t_particle->Lifetime();
  if (!Lifetime)
  {
    double Width = t_particle->Width();
    if (!Width) { return 0.00; }
    Lifetime = constants::hbar / Width;
  }
  return Lifetime;
};

void name_handler_ROOT(FParticle* oParticle, TString tName) {
  if (tName.EqualTo("Z\"0"))
  {
    tName = TString("Z''0");
    oParticle->tName = tName;
    oParticle->tType = TString("Unscope");
    return;
  }
  Ssiz_t pos_tc = tName.Index("tech");
  if (pos_tc != kNPOS)
  {
    oParticle->tType = TString("Technicolor");
  }
  oParticle->tName = tName;
}

#endif
```

ROOTDatamap.hpp :

```
#ifndef ROOTDATAMAP_HPP
#define ROOTDATAMAP_HPP
FParticle_maps ROOTDatamap();
#endif
```

findROOTchannels.cpp :

```
#ifndef FINDROOTCHANNELS_CPP
#define FINDROOTCHANNELS_CPP

#include <vector>
#include <map>
#include <algorithm>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TParticlePDG.h"
#include "TDecayChannel.h"

#include "myparticlestructs.hpp"

#include "findROOTChannels.hpp"
```

```
void fillROOTDaughters(FChannel* oChannel, TDecayChannel* t_channel, std::map<int,
    FParticle*> ROOTmap);

std::vector <FChannel*> findROOTChannels(FParticle* oParticle, TParticlePDG* t_particle,
    std::map<int, FParticle*> ROOTmap) {
  std::vector <FChannel*> V_channels;
  TDecayChannel* t_channel;
  int i_channels, n_channels;
  n_channels = t_particle->NDecayChannels();
  for (i_channels = 0; i_channels < n_channels; i_channels++)
  {
    FChannel* oChannel = new FChannel();
    t_channel = t_particle->DecayChannel(i_channels);
    oChannel->Motherparticle = oParticle;
    oChannel->Branchingratio = t_channel->BranchingRatio();
    fillROOTDaughters(oChannel, t_channel, ROOTmap);
    V_channels.push_back(oChannel);
  }
  return V_channels;
};

bool sortROOTbypdg(FParticle* p1, FParticle* p2) {
  return p1->pdgCode < p2->pdgCode;
};

void fillROOTDaughters(FChannel* oChannel, TDecayChannel* t_channel, std::map<int,
    FParticle*> ROOTmap) {
  std::vector <FParticle*> V_daughters;
  FParticle* odaughter;
  int i_daughters, n_daughers, pdg_daughter;
  n_daughers = t_channel->NDaughters();
  for (i_daughters = 0; i_daughters < n_daughers; i_daughters++)
  {
    pdg_daughter = t_channel->DaughterPdgCode(i_daughters);
    odaughter = ROOTmap.find(pdg_daughter)->second;
    V_daughters.push_back(odaughter);
  }
  std::sort(V_daughters.begin(), V_daughters.end(), sortROOTbypdg);
  oChannel->Daughterparticles = V_daughters;
}
#endif
```

findROOTchannels.hpp :

```
#ifndef FINDROOTCHANNELS_HPP
#define FINDROOTCHANNELS_HPP
std::vector <FChannel*> findROOTChannels(FParticle* oParticle, TParticlePDG* t_particle,
    std::map<int, FParticle*> ROOTmap);
#endif
```

A.6 Pythiamap

The functions used to map the Pythia databse

Pythiadatamap.cpp :

```
#ifndef PYTHIADATAMAP_CPP
#define PYTHIADATAMAP_CPP

#include <map>
#include <vector>
#include <cmath>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"

#include "globals.hpp"
#include "myparticlestructs.hpp"
#include "findPythiaChannels.hpp"

#include "PythiaDatamap.hpp"

void fillChannels(std::map<int, FParticle*> PythiaDatamap, std::map<int, XMLNodePointer_t
    > xmlPythiaDatamap, TXMLEngine* engine);
FParticle* AntiParticlePythia(TXMLEngine* engine, XMLNodePointer_t xmlParticle);
bool PopulateParticlePythia(TXMLEngine* engine, XMLNodePointer_t xmlParticle, FParticle*
    oParticle);
void PythiaName(FParticle* oParticle, TString tName);
double PythiaLifetime(TXMLEngine* engine, XMLNodePointer_t xmlnode);
double PythiaSpin(int spintype, FParticle* oParticle);

FParticle_maps PythiaDatamap() {
  FParticle_maps mapsPythia;
  FParticle* aParticle;
  bool foundAnti;
  std::map <TString, int> mapPythiaNames;
  std::map <int, FParticle*> mapPythia;
  std::map <int, XMLNodePointer_t> xmlmap;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/ParticleData.xml");
  XMLNodePointer_t xmlnode = engine->DocGetRootElement(doc);
  xmlnode = engine->GetChild(xmlnode);
  while (xmlnode)
  {
    if (TString(engine->GetNodeName(xmlnode)).EqualTo("particle"))
    {
      FParticle* oParticle = new FParticle();
      foundAnti = PopulateParticlePythia(engine, xmlnode, oParticle);
      xmlmap.insert(std::pair<int, XMLNodePointer_t>(oParticle->pdgCode, xmlnode));
      mapPythiaNames.insert(std::pair<TString, int>(oParticle->tName, oParticle->pdgCode)
          );
      mapPythia.insert(std::pair<int, FParticle*>(oParticle->pdgCode, oParticle));
      if (foundAnti)
      {
        aParticle = AntiParticlePythia(engine, xmlnode);
        mapPythia.insert(std::pair<int, FParticle*>(aParticle->pdgCode, aParticle));
        mapPythiaNames.insert(std::pair<TString, int> (aParticle->tName, aParticle->
            pdgCode));
      }
    }
    xmlnode = engine->GetNext(xmlnode);
  }
```

```
  fillChannels(mapPythia, xmlmap, engine);
  mapsPythia.map_names = mapPythiaNames;
  mapsPythia.map_pdg = mapPythia;
  return mapsPythia;
};

void fillChannels(std::map<int, FParticle*> PythiaDatamap, std::map<int, XMLNodePointer_t
    > xmlPythiaDatamap, TXMLEngine* engine) {
  int p_pdgcode;
  FParticle* oParticle;
  XMLNodePointer_t xmlChannel;
  for (auto const &ent1 : PythiaDatamap)
  {
    oParticle = ent1.second;
    p_pdgcode = oParticle->pdgCode;
    xmlChannel = xmlPythiaDatamap.find(abs(p_pdgcode))->second;
    xmlChannel = engine->GetChild(xmlChannel);
    oParticle->Channels = findPythiaChannels(PythiaDatamap, engine, xmlChannel, oParticle
        );
  }
}

FParticle* AntiParticlePythia(TXMLEngine* engine, XMLNodePointer_t xmlParticle) {
  FParticle* aParticle = new FParticle();
  aParticle->tType = TString("PythiaAntiParticle");
  aParticle->pdgCode = -1 * TString(engine->GetAttr(xmlParticle, "id")).Atoi();
  aParticle->Mass = TString(engine->GetAttr(xmlParticle, "m0")).Atof();
  aParticle->Charge3e = -1 * TString(engine->GetAttr(xmlParticle, "chargeType")).Atoi();
  aParticle->Lifetime = PythiaLifetime(engine, xmlParticle);
  aParticle->Colortype = TString(engine->GetAttr(xmlParticle, "colType")).Atoi();
  if (aParticle->pdgCode < -999999) { aParticle->tType = TString("Unscope"); }
  aParticle->Spin = PythiaSpin(TString(engine->GetAttr(xmlParticle, "spinType")).Atoi(),
      aParticle);
  if (aParticle->Colortype > 0) { aParticle->tType = TString("ColorMultiplet"); }
  PythiaName(aParticle, TString(engine->GetAttr(xmlParticle, "antiName")));
  return aParticle;
}

bool PopulateParticlePythia(TXMLEngine* engine, XMLNodePointer_t xmlParticle, FParticle*
    oParticle) {
  oParticle->tType = TString("PythiaParticle");
  oParticle->pdgCode = TString(engine->GetAttr(xmlParticle, "id")).Atoi();
  bool anti = engine->GetAttr(xmlParticle, "antiName");
  oParticle->Mass = TString(engine->GetAttr(xmlParticle, "m0")).Atof();
  oParticle->Charge3e = TString(engine->GetAttr(xmlParticle, "chargeType")).Atoi();
  oParticle->Lifetime = PythiaLifetime(engine, xmlParticle);
  oParticle->Colortype = TString(engine->GetAttr(xmlParticle, "colType")).Atoi();
  if (oParticle->pdgCode > 999999) { oParticle->tType = TString("Unscope"); }
  oParticle->Spin = PythiaSpin(TString(engine->GetAttr(xmlParticle, "spinType")).Atoi(),
      oParticle);
  if (oParticle->Colortype > 0) { oParticle->tType = TString("ColorMultiplet"); }
  PythiaName(oParticle, TString(engine->GetAttr(xmlParticle, "name")));
  return anti;
}

void PythiaName(FParticle* oParticle, TString tName) {
  Ssiz_t pos_tc = tName.Index("tc");
  if (pos_tc != kNPOS)
  {
    oParticle->tType = TString("Technicolor");
    oParticle->_scopeFlag = 1;
  }
  Ssiz_t pos_susy = tName.Index("~");
  if (pos_susy != kNPOS)
  {
    oParticle->tType = TString("SParticle");
    oParticle->_scopeFlag = 1;
  }
  oParticle->tName = tName;
```

```
}

double PythiaLifetime(TXMLEngine* engine, XMLNodePointer_t xmlnode) {
  double tau0 = TString(engine->GetAttr(xmlnode, "tau0")).Atof();
  double Width = TString(engine->GetAttr(xmlnode, "mWidth")).Atof();
  double Lifetime = tau0*3.333e-12;
  if (tau0)
  {
    return Lifetime;
  }
  if (Width)
  {
    return (constants::hbar / Width);
  }
  return 0.00;
};

double PythiaSpin(int spintype, FParticle* oParticle) {
  if (spintype == 0)
  {
    oParticle->tType = TString("Unscope");
    return 0.00;
  }
  return (0.5 * (spintype - 1));
}

#endif
```

Pyhtiadatamap.hpp :

```
#ifndef PYTHIADATAMAP_HPP
#define PYTHIADATAMAP_HPP
FParticle_maps PythiaDatamap();
#endif
```

findPythiaChannels.cpp :

```
#ifndef FINDPYTHIACHANNELS_CPP
#define FINDPYTHIACHANNELS_CPP

#include <vector>
#include <map>
#include <string>
#include <sstream>
#include <algorithm>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"

#include "myparticlestructs.hpp"

std::vector <FParticle*> mapPythiaproducts(std::map<int, FParticle*> PythiaDatamap, std::
    string sproducts, int pdg);
std::vector <FParticle*> mapPythiaproducts(std::map<int, FParticle*> PythiaDatamap, std::
    string sproducts);
std::vector <FParticle*> mapPythiaproductsAnti(std::map<int, FParticle*> PythiaDatamap,
    std::string sproducts);
bool sortPythiabypdg(FParticle* p1, FParticle* p2);

std::vector <FChannel*> findPythiaChannels(std::map<int, FParticle*> PythiaDatamap,
    TXMLEngine* engine, XMLNodePointer_t xmlChannel, FParticle* mParticle) {
  std::vector <FChannel*> PythiaChannels;
```

```
  std::string sproducts;
  while (xmlChannel)
  {
    FChannel* oChannel = new FChannel();
    oChannel->Motherparticle = mParticle;
    oChannel->Branchingratio = TString(engine->GetAttr(xmlChannel, "bRatio")).Atof();
    sproducts = engine->GetAttr(xmlChannel, "products");
    oChannel->Daughterparticles = mapPythiaproducts(PythiaDatamap, sproducts, mParticle->
        pdgCode);
    PythiaChannels.push_back(oChannel);
    xmlChannel = engine->GetNext(xmlChannel);
  }
  return PythiaChannels;
}

std::vector<FParticle*> mapPythiaproducts(std::map<int, FParticle*> PythiaDatamap, std::
    string sproducts, int pdg) {
  if (pdg < 0) {
    return mapPythiaproductsAnti(PythiaDatamap, sproducts);
  }
  else {
    return mapPythiaproducts(PythiaDatamap, sproducts);
  }
}

std::vector<FParticle*> mapPythiaproducts(std::map<int, FParticle*> PythiaDatamap, std::
    string sproducts) {
  std::vector<FParticle*> VProducts;
  std::stringstream sStream;
  int n;
  sStream << sproducts;
  while (sStream >> n)
  {
    VProducts.push_back(PythiaDatamap.find(n)->second);
  }
  sStream.clear();
  std::sort(VProducts.begin(), VProducts.end(), sortPythiabypdg);
  return VProducts;
}

std::vector<FParticle*> mapPythiaproductsAnti(std::map<int, FParticle*> PythiaDatamap,
    std::string sproducts) {
  std::vector<FParticle*> VProducts;
  std::stringstream sStream;
  int n;
  sStream << sproducts;
  while (sStream >> n)
  {
    if (PythiaDatamap.count(-n) == 0) {
      VProducts.push_back(PythiaDatamap.find(n)->second);
    }
    else {
      VProducts.push_back(PythiaDatamap.find(-n)->second);
    }
  }
  sStream.clear();
  std::sort(VProducts.begin(), VProducts.end(), sortPythiabypdg);
  return VProducts;
}

bool sortPythiabypdg(FParticle* p1, FParticle* p2) {
  return p1->pdgCode < p2->pdgCode;
}
#endif
```

findPythiaChannels.hpp :

```
#ifndef FINDPYTHIACHANNELS_HPP
```

```
#define FINDPYTHIACHANNELS_HPP
std::vector <FChannel*> findPythiaChannels(std::map<int, FParticle*> PythiaDatamap,
    TXMLEngine* engine, XMLNodePointer_t xmlChannel, FParticle* mParticle);
#endif
```

A.7 EvtGenmap

The functions used to map the EvtGen database

EvtGenDatamap.cpp :

```
#ifndef EVTGENDATAMAP_CPP
#define EVTGENDATAMAP_CPP

#include <map>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"

#include "globals.hpp"
#include "myparticlestructs.hpp"
#include "findEvtGenChannels.hpp"

#include "EvtGenDatamap.hpp"

void PopulateParticleEvtGen(std::string line, FParticle* oParticle);
bool isEvtGencomment(std::string line);

FParticle_maps EvtGenDatamap() {
  FParticle_maps mapsEvtGen;
  std::map<int, FParticle*> mapEvtGen;
  std::map<TString, int> t_mapEvtGen;
  std::string line;
  std::ifstream evtfile("ParticleDataSets/evt.pdl");
  if (evtfile.is_open())
  {
    while (getline(evtfile, line))
    {
      if (isEvtGencomment(line)) { continue; }
      FParticle* oParticle = new FParticle();
      PopulateParticleEvtGen(line, oParticle);
      t_mapEvtGen.insert(std::pair<TString, int>(oParticle->tName, oParticle->pdgCode));
      mapEvtGen.insert(std::pair<int, FParticle*>(oParticle->pdgCode, oParticle));
    }
    evtfile.close();
  }
  else printf("Unable to open evt.pdl");
  t_mapEvtGen = findEvtGenChannels(mapEvtGen, t_mapEvtGen);
  mapsEvtGen.map_names = t_mapEvtGen;
  mapsEvtGen.map_pdg = mapEvtGen;
  return mapsEvtGen;
}

void PopulateParticleEvtGen(std::string line, FParticle* oParticle) {
  oParticle->Colortype = 0;
  int n_attr = 0;
  int evtID, pythiaID;
  double lifetime_mm, Width;
  std::string attr;
  TString t_attr;
  std::stringstream strstr;
  strstr << line;
  const int i_type = 2;
  const int i_name = 3;
```

```
  const int i_pdg = 4;
  const int i_mass = 5;
  const int i_width = 6;
  //const int i_maxDm = 7;
  const int i_charge = 8;
  const int i_spin = 9;
  const int i_lifetime = 10;
  const int i_pythiaid = 11;
  while (strstr >> attr)
  {
    t_attr = TString(attr.data());
    switch (n_attr)
    {
    case i_type: oParticle->tType = t_attr.Data(); break;
    case i_name: oParticle->tName = t_attr; break;
    case i_pdg: evtID = t_attr.Atoi(); break;
    case i_mass: oParticle->Mass = t_attr.Atof(); break;
    case i_width: Width = t_attr.Atof(); break;
    case i_charge: oParticle->Charge3e = t_attr.Atoi(); break;
    case i_spin: oParticle->Spin = 0.5 * t_attr.Atoi(); break;
    case i_lifetime: lifetime_mm = t_attr.Atof(); break;
    case i_pythiaid: pythiaID = t_attr.Atoi(); break;
    }
    n_attr++;
  }
  if (pythiaID == 0) { oParticle->pdgCode = evtID; }
  else { oParticle->pdgCode = pythiaID; }
  if (lifetime_mm == 0 && Width != 0)
  {
    oParticle->Lifetime = constants::hbar / Width;
  }
  else { oParticle->Lifetime = lifetime_mm *  3.333e-12; }
  strstr.clear();
}

bool isEvtGencomment(std::string line) {
  std::string s_partial, s_potential_end;
  s_partial = line.substr(0, 1);
  s_potential_end = line.substr(0, 3);
  if (s_partial == "*") { return true; }
  if (s_potential_end == "end") { return true; }
  else { return false; }
};
#endif
```

EvtGenDatamap.hpp :

```
#ifndef EVTGENDATAMAP_HPP
#define EVTGENDATAMAP_HPP
FParticle_maps EvtGenDatamap();
#endif
```

findEvtGenChannels.cpp :

```
#ifndef FINDEVTGENCHANNELS_CPP
#define FINDEVTGENCHANNELS_CPP

#include <map>
#include <vector>
#include <string>
#include <sstream>
#include <utility>
#include <algorithm>

#include "TROOT.h"
#include "TSystem.h"
```

```
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"

#include "myparticlestructs.hpp"

std::vector <FParticle*> mapEvtGenproducts(std::map<int, FParticle*> mapEvtGen, std::map<
    TString, int> t_mapEvtGen, std::string sproducts);
bool sortEvtGenbypdg(FParticle* p1, FParticle* p2);
std::map<TString, int> makealias(std::map<TString, int> t_mapEvtGen, TXMLEngine* engine,
    XMLNodePointer_t aliasnode);

std::map<TString, int> findEvtGenChannels(std::map<int, FParticle*> mapEvtGen, std::map<
    TString, int> t_mapEvtGen) {
  double Branchingratio;
  std::string sproducts;
  std::vector<FChannel*> VChannels;
  FParticle* mParticle;
  TString tname;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/DECAY_2010.xml");
  XMLNodePointer_t secondary_xmlnode;
  XMLNodePointer_t primary_xmlnode = engine->DocGetRootElement(doc);
  primary_xmlnode = engine->GetChild(primary_xmlnode);
  while (primary_xmlnode)
  {
    if (TString(engine->GetNodeName(primary_xmlnode)).EqualTo("alias"))
    {
      t_mapEvtGen = makealias(t_mapEvtGen, engine, primary_xmlnode);
    }
    if (TString(engine->GetNodeName(primary_xmlnode)).EqualTo("decay"))
    {
      tname = TString(engine->GetAttr(primary_xmlnode, "name"));
      mParticle = mapEvtGen.find(t_mapEvtGen.find(tname)->second)->second;
      secondary_xmlnode = engine->GetChild(primary_xmlnode);
      while (secondary_xmlnode)
      {
        FChannel* oChannel = new FChannel();
        oChannel->Motherparticle = mParticle;
        Branchingratio = TString(engine->GetAttr(secondary_xmlnode, "br")).Atof();
        oChannel->Branchingratio = Branchingratio;
        sproducts = engine->GetAttr(secondary_xmlnode, "daughters");
        oChannel->Daughterparticles = mapEvtGenproducts(mapEvtGen, t_mapEvtGen, sproducts
            );
        if (oChannel->Daughterparticles.size() > 0) { VChannels.push_back(oChannel); }
        secondary_xmlnode = engine->GetNext(secondary_xmlnode);
      }
      mParticle->Channels = VChannels;
      VChannels.clear();
    }
    primary_xmlnode = engine->GetNext(primary_xmlnode);
  }
  return t_mapEvtGen;
}
std::vector <FParticle*> mapEvtGenproducts(std::map<int, FParticle*> mapEvtGen, std::map<
    TString, int> t_mapEvtGen, std::string sproducts) {
  std::vector <FParticle*> VProducts;
  std::stringstream sStream;
  std::string s;
  TString ts;
  sStream << sproducts;
  while (sStream >> s)
  {
    ts = TString(s.data());
    VProducts.push_back(mapEvtGen.find(t_mapEvtGen.find(ts)->second)->second);
  }
  sStream.clear();
  std::sort(VProducts.begin(), VProducts.end(), sortEvtGenbypdg);
```

```
  return VProducts;
}

bool sortEvtGenbypdg(FParticle* p1, FParticle* p2) {
  return p1->pdgCode < p2->pdgCode;
}

std::map<TString, int> makealias(std::map<TString, int> t_mapEvtGen, TXMLEngine* engine,
    XMLNodePointer_t aliasnode) {
  TString tName, tAlias;
  int iParticle;
  tName = TString(engine->GetAttr(aliasnode, "particle"));
  tAlias = TString(engine->GetAttr(aliasnode, "name"));
  iParticle = t_mapEvtGen.find(tName)->second;
  t_mapEvtGen.insert(std::pair<TString, int> (tAlias, iParticle));
  return t_mapEvtGen;
}
#endif
```

findEvtGenChannels.hpp :

```
#ifndef FINDEVTGENCHANNELS_HPP
#define FINDEVTGENCHANNELS_HPP
std::map<TString, int> findEvtGenChannels(std::map<int, FParticle*> mapEvtGen, std::map<
    TString, int> t_mapEvtGen);
#endif
```

A.8 Merge Databases

The function used to merge the three particle databases

merge_data.cpp :

```
#ifndef MERGE_DATA_CPP
#define MERGE_DATA_CPP

#include <map>
#include <vector>
#include <algorithm>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "myparticlestructs.hpp"

#include "merge_data.hpp"

void merge_EvtGen_Pythia_particles(FParticle* oPythiaParticle, FParticle* oEvtGenParticle
    );
void merge_EvtGen_ROOT_particles(FParticle* oPythiaParticle, FParticle* oROOTParticle);
void check_lifetimes(FParticle* oEvtGenParticle, FParticle* oTherparticle);
std::vector<FChannel*> mergechannels(std::vector<FChannel*> VChannels_extra, std::vector<
    FChannel*> VChannels_primary);
bool compare_products(std::vector<FParticle*> V1, std::vector<FParticle*> V2);
std::map<TString, int> combineNames(std::map<TString, int> mapPrimaryNames, std::map<
    TString, int> mapExtraNames);
FParticle_maps manual_additions(FParticle_maps mymaps);

FParticle_maps merge_data(FParticle_maps mapsPythia, FParticle_maps mapsEvtGen,
    FParticle_maps mapsROOT) {
  FParticle_maps merged_maps;
  std::map<int, FParticle*> mapPythiaParticles, mapROOTParticles, mapEvtGenParticles;
  std::map<TString, int> mynamesmap;
  int pdgROOT, pdgPythia;
  FParticle* oPythiaParticle, *oEvtGenParticle, *oROOTParticle;
  mapPythiaParticles = mapsPythia.map_pdg;
  mapEvtGenParticles = mapsEvtGen.map_pdg;
  mapROOTParticles = mapsROOT.map_pdg;
  for (auto const &ent1 : mapPythiaParticles)
  {
    oPythiaParticle = ent1.second;
    pdgPythia = oPythiaParticle->pdgCode;
    if (mapEvtGenParticles.count(pdgPythia) == 0)
    {
      mapEvtGenParticles.insert(std::pair<int, FParticle*>(pdgPythia, oPythiaParticle));
    }
    else
    {
      oEvtGenParticle = mapEvtGenParticles.find(pdgPythia)->second;
      merge_EvtGen_Pythia_particles(oEvtGenParticle, oPythiaParticle);
    }
  }
  for (auto const &ent2 : mapROOTParticles)
  {
    oROOTParticle = ent2.second;
    pdgROOT = oROOTParticle->pdgCode;
    if (mapEvtGenParticles.count(pdgROOT) == 0)
    {
      mapEvtGenParticles.insert(std::pair<int, FParticle*>(pdgROOT, oROOTParticle));
    }
    else
    {
```

```
      oEvtGenParticle = mapEvtGenParticles.find(pdgROOT)->second;
      merge_EvtGen_ROOT_particles(oEvtGenParticle, oROOTParticle);
    }
  }
  merged_maps.map_pdg = mapEvtGenParticles;
  mynamesmap = combineNames(mapsEvtGen.map_names, mapsPythia.map_names);
  mynamesmap = combineNames(mynamesmap, mapsROOT.map_names);
  merged_maps.map_names = mynamesmap;
  merged_maps = manual_additions(merged_maps);
  return merged_maps;
}

void merge_EvtGen_Pythia_particles(FParticle* oEvtGenParticle, FParticle* oPythiaParticle
    ) {
  std::vector<FChannel*> VChannelsPythia, VChannelsEvtGen, VChannels;
  check_lifetimes(oEvtGenParticle, oPythiaParticle);
  oEvtGenParticle->Colortype = oPythiaParticle->Colortype;
  oEvtGenParticle->tType = oPythiaParticle->tType;
  VChannelsPythia = oPythiaParticle->Channels;
  VChannelsEvtGen = oEvtGenParticle->Channels;
  VChannels = mergechannels(VChannelsEvtGen, VChannelsPythia);
  oEvtGenParticle->Channels = VChannels;
}

void merge_EvtGen_ROOT_particles(FParticle* oEvtGenParticle, FParticle* oROOTParticle) {
  std::vector<FChannel*> VChannelsEvtGen, VChannelsROOT, VChannels;
  check_lifetimes(oEvtGenParticle, oROOTParticle);
  VChannelsEvtGen = oEvtGenParticle->Channels;
  VChannelsROOT = oROOTParticle->Channels;
  VChannels = mergechannels(VChannelsEvtGen, VChannelsROOT);
  oEvtGenParticle->Channels = VChannels;
  if (oEvtGenParticle->tType.Index("Particle") != kNPOS)
  {
    oEvtGenParticle->tType = oROOTParticle->tType;
  }
}

void check_lifetimes(FParticle* oEvtGenParticle, FParticle* oTherparticle) {
  double lifetime_EvtGen = oEvtGenParticle->Lifetime;
  if (!lifetime_EvtGen || lifetime_EvtGen == 0)
  {
    oEvtGenParticle->Lifetime = oTherparticle->Lifetime;
    if (!oTherparticle->Lifetime)
    {
      oEvtGenParticle->Lifetime = 0;
    }
  }
}

std::vector<FChannel*> mergechannels(std::vector<FChannel*> VChannels_primary, std::
    vector<FChannel*> VChannels_extra) {
  FChannel* pCh, *eCh;
  bool foundchannel;
  int i_channels_extra, n_channels_extra, i_channels_primary, n_channels_primary;
  n_channels_extra = VChannels_extra.size();
  n_channels_primary = VChannels_primary.size();
  for (i_channels_extra = 0; i_channels_extra < n_channels_extra; i_channels_extra++)
  {
    eCh = VChannels_extra[i_channels_extra];
    foundchannel = false;
    for (i_channels_primary = 0; i_channels_primary < n_channels_primary;
        i_channels_primary++)
    {
      pCh = VChannels_primary[i_channels_primary];
      if (compare_products(pCh->Daughterparticles, eCh->Daughterparticles))
      {
        foundchannel = true;
        break;
      }
```

```
    }
    if (!foundchannel)
    {
      VChannels_primary.push_back(eCh);
    }
  }
  return VChannels_primary;
}

bool compare_particles(FParticle* p1, FParticle* p2) {
  return (p1->pdgCode == p2->pdgCode);
}

bool compare_products(std::vector<FParticle*> V1, std::vector<FParticle*> V2) {
  if (V1.size() != V2.size())
  {
    return false;
  }
  std::pair<std::vector<FParticle*>::iterator, std::vector<FParticle*>::iterator> mypair;
  mypair = std::mismatch(V1.begin(), V1.end(), V2.begin(), compare_particles);
  return (mypair.first == V1.end());
}

std::map<TString, int> combineNames(std::map<TString, int> mapPrimaryNames, std::map<
    TString, int> mapExtraNames) {
  TString ExtraName;
  for (auto const &ent1 : mapExtraNames)
  {
    ExtraName = ent1.first;
    if (mapPrimaryNames.count(ExtraName) == 0)
    {
      mapPrimaryNames.insert(ent1);
    }
  }
  return mapPrimaryNames;
}

FParticle_maps manual_additions(FParticle_maps mymaps) {
  std::map<int, FParticle*> mypdgmap = mymaps.map_pdg;
  std::map<TString, int> mynamesmap = mymaps.map_names;
  mynamesmap.insert(std::pair<TString, int>(TString("Upsilon(1S)"), 553));
  FParticle* oParticle;
  int uns_first = 31, uns_last = 100;
  for (int i = uns_first; i <= uns_last; i++)
  {
    if (mypdgmap.count(i) != 0)
    {
      oParticle = mypdgmap.find(i)->second;
      oParticle->tType = TString("Unscope");
    }
    if (mypdgmap.count(-i) != 0)
    {
      oParticle = mypdgmap.find(-i)->second;
      oParticle->tType = TString("Unscope");
    }
  }
  // 4th gen lepton
  int id_L = 17, id_nu_L = 18;
  oParticle = mypdgmap.find(id_L)->second;
  oParticle->tType = TString("Unscope");
  oParticle = mypdgmap.find(id_nu_L)->second;
  oParticle->tType = TString("Unscope");
  // Stable particles
  int pdg_e = 11;
  int pdg_nu_e = 12;
  int pdg_nu_mu = 14;
  int pdg_nu_tau = 16;
  int pdg_gamma = 22;
  int pdg_proton = 2212;
```

```
  int stable_part[11] = {pdg_e, pdg_nu_e, pdg_nu_mu, pdg_nu_tau, pdg_gamma, pdg_proton, -
      pdg_e, -pdg_nu_e, -pdg_nu_mu, -pdg_nu_tau, -pdg_proton };
  double infinite_lifetime = 3.14e7;
  for (int i = 0; i < 11; i++)
  {
    oParticle = mypdgmap.find(stable_part[i])->second;
    oParticle->Lifetime = infinite_lifetime;
  }
  return{mypdgmap, mynamesmap};
}

#endif
```

merge_data.hpp :

```
#ifndef MERGE_DATA_HPP
#define MERGE_DATA_HPP
FParticle_maps merge_data(FParticle_maps mapsPythia, FParticle_maps mapsEvtGen,
    FParticle_maps mapsROOT);
#endif
```

A.9 Writing XML database

The function used to write the combined particle map into a XML database.

WriteXMLDatabase.cpp :

```
#ifndef WRITEXMLDATABASE_CPP
#define WRITEXMLDATABASE_CPP

#include <map>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "myparticlestructs.hpp"

void printXMLChannels(FParticle* oParticle, std::map<int, FParticle*> mapParticles, FILE*
     fp);

void WriteXMLDatabase(FParticle_maps mymap, char* filename) {
  FILE *fp;
  fp = fopen(filename, "w+");
  fprintf(fp, "<ParticleDataFile>\n Rewritten Particle database combining Pythia, EvtGen
       and ROOT databases\n");
  std::map<TString, int> mapNames = mymap.map_names;
  std::map<int, FParticle*> mapParticles = mymap.map_pdg;
  TString alias_candidate;
  TString pName;
  FParticle* pParticle;
  int p_pdg;
  for (auto const &name_and_id : mapNames)
  {
    alias_candidate = name_and_id.first;
    p_pdg = name_and_id.second;
    pParticle = mapParticles.find(p_pdg)->second;
    pName = pParticle->tName;
    if (!alias_candidate.EqualTo(pName))
    {
      fprintf(fp, "<alias name=\"%s\" alias=\"%s\" id=\"%d\"/>\n", pName.Data(),
          alias_candidate.Data(), name_and_id.second);
    }
  }
  for (auto const &ent1 : mapParticles)
  {
    FParticle *oParticle = ent1.second;
    fprintf(fp, "<particle ");
    fprintf(fp, "id=\"%d\" ", oParticle->pdgCode);
    fprintf(fp, "name=\"%s\" ", oParticle->tName.Data());
    fprintf(fp, "spin=\"%g\" ", oParticle->Spin);
    fprintf(fp, "charge3e=\"%d\" ", oParticle->Charge3e);
    fprintf(fp, "mass_GeV=\"%g\" ", oParticle->Mass);
    fprintf(fp, "lifetime_s=\"%g\" ", oParticle->Lifetime);
    fprintf(fp, "colortype=\"%d\" ", oParticle->Colortype);
    fprintf(fp, "type=\"%s\" ", oParticle->tType.Data());
    fprintf(fp, ">\n");
    printXMLChannels(oParticle, mapParticles, fp);
    fprintf(fp, "</particle>\n");
  }
  fprintf(fp, "</ParticleDataFile>");
  fclose(fp);
};

void printXMLChannels(FParticle* oParticle, std::map<int, FParticle*> mapParticles, FILE*
     fp) {
  double branching;
  std::vector <FChannel*> VChannels;
```

```
  std::vector <FParticle*> Vproducts;
  FParticle* oProduct;
  int n_channels, n_products, i_channels, i_products;
  VChannels = oParticle->Channels;
  n_channels = VChannels.size();
  for (i_channels = 0; i_channels < n_channels; i_channels++)
  {
    branching = VChannels[i_channels]->Branchingratio;
    fprintf(fp, "<channel branchingratio=\"%g\" ->", branching);
    Vproducts = VChannels[i_channels]->Daughterparticles;
    n_products = Vproducts.size();
    for (i_products = 0; i_products < n_products; i_products++) {
      oProduct = mapParticles.find(Vproducts[i_products]->pdgCode)->second;
      fprintf(fp, "<product id=\"%d\" name=\"%s\"/>", oProduct->pdgCode, oProduct->tName.
          Data());
    }
    fprintf(fp, "</channel>\n");
  }
}

#endif
```

WriteXMLDatabase.hpp :

```
#ifndef WRITEXMLDATABASE_HPP
#define WRITEXMLDATABASE_HPP
void WriteXMLDatabase(FParticle_maps mymap, char* filename);
#endif
```

A.10 Writing ROOT database

The function used to write the combined particle map into a ROOT database.

WriteROOTDatabase.cpp :

```
#ifndef WRITEROOTDATABASE_CPP
#define WRITEROOTDATABASE_CPP

#include <map>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"

#include "myparticlestructs.hpp"
#include "globals.hpp"

#include "WriteROOTDatabase.hpp"

void fillROOTChannels(FParticle* oParticle, TParticlePDG* tParticle);

void WriteROOTDatabase(FParticle_maps mymap, char* filename) {
  FILE* fp;
  fp = fopen(filename, "w+");
  fclose(fp);

  FILE* xmlp;
  xmlp = fopen("ParticleDataSets/Rewrite/particleAlias.xml", "w+");
  fprintf(xmlp, "<ParticleAliasFile>\n");

  int p_pdgcode;
  FParticle* oParticle, *aParticle;
  TParticlePDG* tParticle;
  TString pName, alias_candidate;

  std::map<int, FParticle*> mapParticles = mymap.map_pdg;
  std::map<TString, int> mapNames = mymap.map_names;
  TDatabasePDG *pdgdata = new TDatabasePDG();
  pdgdata->ReadPDGTable(filename);
  for (auto const &name_and_id : mapNames)
  {
    alias_candidate = name_and_id.first;
    pName = (mapParticles.find(name_and_id.second)->second)->tName;
    if (!alias_candidate.EqualTo(pName))
    {
      fprintf(xmlp, "<alias name=\"%s\" alias=\"%s\" id=\"%d\"/>\n", pName.Data(),
          alias_candidate.Data(), name_and_id.second);
    }
  }
  for (auto const &ent1 : mapParticles)
  {
    p_pdgcode = ent1.first;
    if (p_pdgcode <= 0) { continue; }
    oParticle = ent1.second;
    if (mapParticles.count(-1 * p_pdgcode) > 0) {
      pdgdata->AddParticle(oParticle->tName.Data(), oParticle->tName.Data(), oParticle->
          Mass, 0, oParticle->Lifetime, oParticle->Charge3e, oParticle->tType.Data(),
          oParticle->pdgCode, -1);
      aParticle = mapParticles.find(-1 * p_pdgcode)->second;
      pdgdata->AddAntiParticle(aParticle->tName.Data(), aParticle->pdgCode);
    }
    else
```

```
    {
      pdgdata->AddParticle(oParticle->tName.Data(), oParticle->tName.Data(), oParticle->
          Mass, 0, oParticle->Lifetime, oParticle->Charge3e, oParticle->tType.Data(),
          oParticle->pdgCode, 0);
    }
    tParticle = pdgdata->GetParticle(oParticle->tName.Data());
    fillROOTChannels(oParticle, tParticle);
  }

  pdgdata->WritePDGTable(filename);
  delete pdgdata;
  fprintf(xmlp, "</ParticleAliasFile>\n");
  fclose(xmlp);
}

void fillROOTChannels(FParticle* oParticle, TParticlePDG* tParticle) {
  double branching;
  std::vector <FChannel*> VChannels;
  std::vector <FParticle*> Vproducts;
  FParticle* oProduct;
  int n_channels, n_products, i_channels, i_products;
  int* daughterpdgcodes;
  VChannels = oParticle->Channels;
  n_channels = VChannels.size();
  for (i_channels = 0; i_channels < n_channels; i_channels++)
  {
    branching = VChannels[i_channels]->Branchingratio;
    Vproducts = VChannels[i_channels]->Daughterparticles;
    n_products = Vproducts.size();
    daughterpdgcodes = (int*)malloc(sizeof(int) * n_products);
    for (i_products = 0; i_products < n_products; i_products++) {
      oProduct = Vproducts[i_products];
      daughterpdgcodes[i_products] = oProduct->pdgCode;
    }
    tParticle->AddDecayChannel(i_channels, branching, n_products, daughterpdgcodes);
  }
}

#endif

void fillROOTChannels(FParticle* oParticle, TParticlePDG* tParticle) {
  double branching;
  std::vector <FChannel*> VChannels;
  std::vector <FParticle*> Vproducts;
  FParticle* oProduct;
  int n_channels, n_products, i_channels, i_products;
  int* daughterpdgcodes;
  VChannels = oParticle->Channels;
  n_channels = VChannels.size();
  for (i_channels = 0; i_channels < n_channels; i_channels++)
  {
    branching = VChannels[i_channels]->Branchingratio;
    Vproducts = VChannels[i_channels]->Daughterparticles;
    n_products = Vproducts.size();
    daughterpdgcodes = (int*)malloc(sizeof(int) * n_products);
    for (i_products = 0; i_products < n_products; i_products++) {
      oProduct = Vproducts[i_products];
      daughterpdgcodes[i_products] = oProduct->pdgCode;
    }
    tParticle->AddDecayChannel(i_channels, branching, n_products, daughterpdgcodes);
  }
}

#endif
```

WriteROOTDatabase.hpp :

```
#ifndef WRITEROOTDATABASE_HPP
```

```
#define WRITEROOTDATABASE_HPP
void WriteROOTDatabase(FParticle_maps mymap, char* filename);
#endif
```

A.11 Initializing the databases

A function to initialize a rewritten, combined database from the ROOT, Pythia and EvtGen databases. This can also be used to update the combined database.

initialize_databse.cpp :

```
#ifndef INITIALISE_DATABASE_CPP
#define INITIALISE_DATABASE_CPP

#include <map>
#include <vector>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "myparticlestructs.hpp"
#include "PythiaDatamap.hpp"
#include "EvtGenDatamap.hpp"
#include "ROOTDatamap.hpp"
#include "merge_data.hpp"
#include "WriteXMLDatabase.hpp"
#include "WriteROOTDatabase.hpp"

#include "initialise_database.hpp"

void initialise_database() {
  FParticle_maps mapROOTParticles = ROOTDatamap();
  FParticle_maps mapPythiaParticles = PythiaDatamap();
  FParticle_maps mapEvtGenParticles = EvtGenDatamap();
  FParticle_maps mapMyParticles;
  mapMyParticles = merge_data(mapPythiaParticles, mapEvtGenParticles, mapROOTParticles);
  WriteROOTDatabase(mapMyParticles, "ParticleDataSets/Rewrite/RewrittenDatabase.txt");
  WriteXMLDatabase(mapMyParticles, "ParticleDataSets/Rewrite/RewrittenDatabase.xml");
};

#endif
```

initialize_databse.hpp :

```
#ifndef INITIALISE_DATABASE_HPP
#define INITIALISE_DATABASE_HPP
void initialise_database();
#endif
```

A.12 Remapping the database

These functions can be used to regain the cpp map from the rewritten databases. One function for the XML database, and one for the ROOT databse.

mappingmyXMLDB.cpp :

```
#ifndef MAPPINGMYXMLDB_CPP
#define MAPPINGMYXMLDB_CPP

#include <map>
#include <vector>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TXMLEngine.h"
#include "TString.h"

#include "myparticlestructs.hpp"

#include "mappingmyXMLDB.hpp"

bool populateParticle(TXMLEngine* engine, XMLNodePointer_t particle_xmlnode, FParticle*
    oParticle);
void fillChannels(TXMLEngine* engine, std::map<int, FParticle*> myparticlemap, std::map <
    int, XMLNodePointer_t> xmlmap);
std::vector <FChannel*> findChannels(TXMLEngine* engine, XMLNodePointer_t xmlChannel, std
    ::map<int, FParticle*> myparticlemap, FParticle* mParticle);
void findDaughters(TXMLEngine* engine, XMLNodePointer_t xmlproducts, std::map<int,
    FParticle*> myparticlemap, int m_pdg, FChannel* oChannel);

FParticle_maps mappingmyXMLDB() {
  std::map <int, XMLNodePointer_t> xmlmap;
  std::map<int, FParticle*> myparticlemap;
  std::map<TString, int> mynamesmap;
  int pdg_alias;
  TString alias;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/Rewrite/RewrittenDatabase.xml
      ");
  XMLNodePointer_t particle_xmlnode = engine->DocGetRootElement(doc);
  particle_xmlnode = engine->GetChild(particle_xmlnode);
  while (particle_xmlnode)
  {
    if (TString(engine->GetNodeName(particle_xmlnode)).EqualTo("alias"))
    {
      pdg_alias = TString(engine->GetAttr(particle_xmlnode, "id")).Atoi();
      alias = TString(engine->GetAttr(particle_xmlnode, "alias"));
      mynamesmap.insert(std::pair<TString, int>(alias, pdg_alias));
    }
    if (TString(engine->GetNodeName(particle_xmlnode)).EqualTo("particle"))
    {
      FParticle* oParticle = new FParticle();
      populateParticle(engine, particle_xmlnode, oParticle);
      xmlmap.insert(std::pair<int, XMLNodePointer_t>(oParticle->pdgCode, particle_xmlnode
          ));
      myparticlemap.insert(std::pair<int, FParticle*>(oParticle->pdgCode, oParticle));
      mynamesmap.insert(std::pair<TString, int>(oParticle->tName, oParticle->pdgCode));
    }
    particle_xmlnode = engine->GetNext(particle_xmlnode);
  }
  fillChannels(engine, myparticlemap, xmlmap);
  return{ myparticlemap, mynamesmap};
};
```

```
bool populateParticle(TXMLEngine* engine, XMLNodePointer_t particle_xmlnode, FParticle*
    oParticle) {
  bool foundAnti = false;
  oParticle->pdgCode = TString(engine->GetAttr(particle_xmlnode, "id")).Atoi();
  oParticle->tName = TString(engine->GetAttr(particle_xmlnode, "name"));
  oParticle->Spin = TString(engine->GetAttr(particle_xmlnode, "spin")).Atof();
  oParticle->Charge3e = TString(engine->GetAttr(particle_xmlnode, "charge3e")).Atoi();
  oParticle->Mass = TString(engine->GetAttr(particle_xmlnode, "mass_GeV")).Atof();
  oParticle->Lifetime = TString(engine->GetAttr(particle_xmlnode, "lifetime_s")).Atof();
  oParticle->Colortype = TString(engine->GetAttr(particle_xmlnode, "colortype")).Atoi();
  oParticle->tType = TString(engine->GetAttr(particle_xmlnode, "type"));
  oParticle->_scopeFlag = bTypeFlag(oParticle->tType);
  return foundAnti;
};

void fillChannels(TXMLEngine* engine, std::map<int, FParticle*> myparticlemap, std::map <
    int, XMLNodePointer_t> xmlmap) {
  int p_pdgcode;
  FParticle* oParticle;
  XMLNodePointer_t xmlChannel;
  for (auto const &ent1 : myparticlemap)
  {
    oParticle = ent1.second;
    p_pdgcode = oParticle->pdgCode;
    xmlChannel = xmlmap.find(p_pdgcode)->second;
    xmlChannel = engine->GetChild(xmlChannel);
    oParticle->Channels = findChannels(engine, xmlChannel, myparticlemap, oParticle);
  }
};

std::vector <FChannel*> findChannels(TXMLEngine* engine, XMLNodePointer_t xmlChannel, std
    ::map<int, FParticle*> myparticlemap, FParticle* mParticle) {
  std::vector <FChannel*> V_channels;
  XMLNodePointer_t xmlproducts;
  while (xmlChannel)
  {
    FChannel* oChannel = new FChannel();
    oChannel->Motherparticle = mParticle;
    oChannel->Branchingratio = TString(engine->GetAttr(xmlChannel, "bRatio")).Atof();
    xmlproducts = engine->GetChild(xmlChannel);
    findDaughters(engine, xmlproducts, myparticlemap, mParticle->pdgCode, oChannel);
    V_channels.push_back(oChannel);
    xmlChannel = engine->GetNext(xmlChannel);
  }
  return V_channels;
};

void findDaughters(TXMLEngine* engine, XMLNodePointer_t xmlproducts, std::map<int,
    FParticle*> myparticlemap, int m_pdg, FChannel* oChannel) {
  int product_pdg;
  std::vector <FParticle*> V_products;
  FParticle* oProduct;
  bool p_flag = 0;
  bool E_violation = 0;
  while (xmlproducts)
  {
    product_pdg = TString(engine->GetAttr(xmlproducts, "id")).Atoi();
    oProduct = myparticlemap.find(product_pdg)->second;
    V_products.push_back(oProduct);
    p_flag |= oProduct->_scopeFlag;
    E_violation |= (oProduct->Mass > oChannel->Motherparticle->Mass);
    xmlproducts = engine->GetNext(xmlproducts);
  }
  oChannel->Daughterparticles = V_products;
  oChannel->_chFlag = (p_flag || E_violation);
}

#endif
```

mappingmyXMLDB.hpp :

```
#ifndef MAPPINGMYXMLDB_HPP
#define MAPPINGMYXMLDB_HPP
FParticle_maps mappingmyXMLDB();
#endif
```

mappingmyROOTDB :

```
#ifndef MAPPINGMYROOTDB_CPP
#define MAPPINGMYROOTDB_CPP

#include <map>
#include <vector>
#include <utility>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TXMLEngine.h"
#include "TString.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "THashList.h"

#include "myparticlestructs.hpp"
#include "findROOTChannels.hpp"
#include "globals.hpp"

#include "mappingmyROOTDB.hpp"

std::map<TString, int> mapROOTNames(std::map<TString, int> namemap);
void fillmyROOTChannels(std::map<int, FParticle*> ROOTmap, TDatabasePDG* pdg);
void PopulateMyParticleROOT(TParticlePDG* t_particle, FParticle* oParticle);

FParticle_maps mappingmyROOTDB(char * filename) {
  FParticle_maps mapsROOT;
  std::map<TString, int> mapNames;
  std::map<int, FParticle*> mapParticles;
  TDatabasePDG* pdg = TDatabasePDG::Instance();
  pdg->ReadPDGTable(filename);
  TParticlePDG* t_particle = 0;
  TIter next(pdg->ParticleList());
  while (t_particle = (TParticlePDG*)next())
  {
    FParticle* oParticle = new FParticle();
    PopulateMyParticleROOT(t_particle, oParticle);
    mapParticles.insert(std::pair<int, FParticle*>(oParticle->pdgCode, oParticle));
    mapNames.insert(std::pair<TString, int>(oParticle->tName, oParticle->pdgCode));
  }
  fillmyROOTChannels(mapParticles, pdg);
  mapNames = mapROOTNames(mapNames);
  mapsROOT.map_names = mapNames;
  mapsROOT.map_pdg = mapParticles;
  delete pdg;
  return mapsROOT;
};

void fillmyROOTChannels(std::map<int, FParticle*> ROOTmap, TDatabasePDG* pdg) {
  FParticle* oParticle;
  TParticlePDG* t_particle;
  for (auto const &ent1 : ROOTmap)
  {
    oParticle = ent1.second;
    t_particle = pdg->GetParticle(ent1.first);
    oParticle->Channels = findROOTChannels(oParticle, t_particle, ROOTmap);
  }
```

```
};

void PopulateMyParticleROOT(TParticlePDG* t_particle, FParticle* oParticle) {
  oParticle->pdgCode = t_particle->PdgCode();
  TString pName = TString(t_particle->GetName());
  oParticle->tName = pName;
  oParticle->Mass = t_particle->Mass();
  oParticle->Charge3e = (int)t_particle->Charge();
  oParticle->Spin = t_particle->Spin(); //Can only be 0
  oParticle->Lifetime = (constants::hbar / t_particle->Width());
  oParticle->Colortype = 0;
  oParticle->tType = TString(engine->GetAttr(particle_xmlnode, "type"));
  oParticle->_scopeFlag = bTypeFlag(oParticle->tType);
};

std::map<TString, int> mapROOTNames(std::map<TString, int> namemap) {
  int pdg_alias;
  TString alias;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/Rewrite/particleAlias.xml");
  XMLNodePointer_t alias_xmlnode = engine->DocGetRootElement(doc);
  alias_xmlnode = engine->GetChild(alias_xmlnode);
  while (alias_xmlnode)
  {
    pdg_alias = TString(engine->GetAttr(alias_xmlnode, "id")).Atoi();
    alias = TString(engine->GetAttr(alias_xmlnode, "alias"));
    if (namemap.count(alias) == 0)
    {
      namemap.insert(std::pair<TString, int>(alias, pdg_alias));
    }
    alias_xmlnode = engine->GetNext(alias_xmlnode);
  }
  return namemap;
}

FParticle_maps mappingmyROOTDB() {
  char* default_filename = "ParticleDataSets/Rewrite/RewrittenDatabase.txt";
  return mappingmyROOTDB(default_filename);
}

#endif
```

mappingmyROOTDB.hpp :

```
#ifndef MAPPINGMYXMLDB_HPP
#define MAPPINGMYXMLDB_HPP
FParticle_maps mappingmyXMLDB();
#endif
```

A.13 Making a histogram of a misidentification

This function simulates a misidentification and creates a histogram of the reconstructed mass spectrum.

MisID2DecayHist.cpp :

```
#ifndef MISID2DECAYHIST_CPP
#define MISID2DECAYHIST_CPP

#include <cmath>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TH1F.h"
#include "TLorentzVector.h"
#include "TGenPhaseSpace.h"
#include "TRandom2.h"

#include "globals.hpp"
#include "beta_distribution.hpp"
#include "detect.hpp"

#include "MisID2DecayHist.hpp"

TH1F* MisID2DecayHist(double MassMeson, double Massp1, double Massp2, double Massp1_MisID
    , double Massp2_MisID) {
  double weight;
  double x2, y2, z2, x1, y1, z1;
  double E1Measure, E2Measure, EMeasure, P_Recon_pow2, Mass_Recon;
  double m[2] = { Massp1, Massp2 };
  double betaz = beta_distribution();
  int i, n = 2;
  TLorentzVector Meson(0, 0, 0, MassMeson);
  TLorentzVector* LV_p1Rest;
  TLorentzVector* LV_p2Rest;
  TLorentzVector LV_p1, LV_p2;
  TGenPhaseSpace Event;
  TH1F* h_2BD = new TH1F("h_2BD", "h_2BD", 1000, 0, (MassMeson + 2));
  h_2BD->SetLineColor(1);
  Event.SetDecay(Meson, n, m);
  for (i = 0; i < globals::Nevents; i++) {
    weight = Event.Generate();
    LV_p1Rest = Event.GetDecay(0);
    LV_p2Rest = Event.GetDecay(1);
    //Boosting
    LV_p1 = *LV_p1Rest;
    LV_p2 = *LV_p2Rest;
    LV_p1.Boost(0, 0, betaz);
    LV_p2.Boost(0, 0, betaz);
    //Momenta
    x1 = LV_p1[0];
    y1 = LV_p1[1];
    z1 = LV_p1[2];
    x2 = LV_p2[0];
    y2 = LV_p2[1];
    z2 = LV_p2[2];
    //MisID
    LV_p1[3] = sqrt((pow(Massp1_MisID, 2) + pow(x1, 2) + pow(y1, 2) + pow(z1, 2)));
    LV_p2[3] = sqrt((pow(Massp2_MisID, 2) + pow(x2, 2) + pow(y2, 2) + pow(z2, 2)));
    //Detection
    E1Measure = detect(LV_p1, Massp1);
    E2Measure = detect(LV_p2, Massp2);
    // Reconstruction:
    EMeasure = E1Measure + E2Measure;
```

```
    P_Recon_pow2 = pow((x1 + x2), 2) + pow((y1 + y2), 2) + pow((z1 + z2), 2);
    Mass_Recon = sqrt(abs(pow(EMeasure, 2) - (P_Recon_pow2)));
    //Fill
    h_2BD->Fill(Mass_Recon, weight);
  }
  return h_2BD;
};
#endif
```

MisID2DecayHist.hpp :

```
#ifndef MISID2DECAYHIST_HPP
#define MISID2DECAYHIST_HPP
TH1F* MisID2DecayHist(double MassMeson, double Massp1, double Massp2, double Massp1_MisID
    , double Massp2_MisID);
#endif
```

A.14 Detector effects

This function applies detector uncertainty and Bremsstrahlung.

detect.cpp :

```
#ifndef INVARIANT_MASS_DETECTOR_PROPERTIES_CPP
#define INVARIANT_MASS_DETECTOR_PROPERTIES_CPP

#include <cmath>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TLorentzVector.h"
#include "TRandom2.h"

#include "detect.hpp"
#include "globals.hpp"

double detect(TLorentzVector LV, double rest_mass) {
  //Detector input: uncertainty in measuremaent (width gauss) & parameter bremstrahlung (
      exp power)
  double factor_brem = 1e-15;
  // parameters
  double E_perfect, E_Measure, rExp;
  TRandom2 *random = new TRandom2();
  random->SetSeed();
  // input specific:
  E_perfect = LV[3];
  //detector uncairtainty
  E_Measure = random->Gaus(E_perfect, globals::sigma);
  //Bremstrahlung
  rExp = random->Exp(globals::alpha_Brem);
  E_Measure = E_Measure - rExp * factor_brem * pow(E_Measure, 2) / pow(rest_mass, 4);
  delete random;
  return E_Measure;
};

#endif
```

detect.hpp :

```
#ifndef INVARIANT_MASS_DETECTOR_PROPERTIES_HPP
#define INVARIANT_MASS_DETECTOR_PROPERTIES_HPP
double detect(TLorentzVector LV, double rest_mass);
#endif
```

A.15 Global parameters

A function determining various variable used throughout the program

globlas.cpp :

```
#ifndef GLOBALS_CPP
#define GLOBALS_CPP

#include "globals.hpp"

namespace constants
{
  extern const double hbar = 6.582e-25; //Units of GeV*s
}

namespace globals
{
  extern double detector_tau(1.0e-9); //1ns of lifetime corresponds to almost 1m flight, reaching the 1st detector
  extern double sigma(0.001); //The width of the Gaussian function descibing the detector uncertainty
  extern double alpha_Brem(0.005); //defines how often an electron emmits bremstrahlung
  extern double deltaW(0.1); //defines the invariant mass search window in GeV
  extern double minBR(-1.00); //The minimum branching ratio for inclusion of a background, for no limit enter -1.00
  extern int Nevents(5000); //The number of events simulated in a single background
}

namespace cascade
{
  extern double max_tau(1.0e-9); //maximum lifetime for cascade particle to be considered background
  extern int max_P_missing(3); //The maximum number of particles that can be undetected in a significant cascade background
}

#endif
```

```
#ifndef GLOBALS_HPP
#define GLOBALS_HPP

namespace constants
{
  extern const double hbar;
}

namespace globals
{
  extern double detector_tau;
  extern double sigma;
  extern double alpha_Brem;
  extern double deltaW;
  extern double minBR;
  extern int Nevents;
}

namespace cascade
{
  extern double max_tau;
  extern int max_P_missing;
}

#endif
```

A.16 Locating peaking backgrounds

MisID.cpp :

```
#ifndef MISID_CPP
#define MISID_CPP

#include <vector>
#include <sstream>
#include <string>
#include <stdio.h>
#include <cmath>
#include <algorithm>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TError.h"
#include "TStyle.h"
#include "TLine.h"
#include "TLegend.h"
#include "TLegendEntry.h"
#include "TCanvas.h"
#include "TH1F.h"
#include "TDatabasePDG.h"

#include "globals.hpp"
#include "myparticlestructs.hpp"
#include "MisIDInput.hpp"
#include "titleName.hpp"
#include "particledata.hpp"
#include "MisID2DecayHist.hpp"
#include "physics2BD.hpp"
#include "Massdifference.hpp"
#include "FindBranching.hpp"
#include "filenames.hpp"

#include "MisID.hpp"

void testQ(int NCombi) {
  char ans;
  printf("Number of combinations = %d \n", NCombi);
  printf("Do you want to continue the operation? (Y/N): ");
  scanf("%c", &ans);
  if (ans == 'y' || ans == 'Y') { ; }
  else { exit(-1); }
}

std::vector<FParticle**> PCombine(std::vector<std::vector<FParticle*>> VParticles,
    TParticlePDG** oParticleInput) {
  int NPDG_P0MisID, NPDG_P1MisID, NPDG_P2MisID, NPDGP0, NPDGP1, NPDGP2, i_p0, i_p1, i_p2;
  int n_p0, n_p1, n_p2;
  double Massp0, Massp1, Massp2, Massdiff, SpinP0, SpinP1, SpinP2, MassP0MisID,
      MassP1MisID, MassP2MisID, chargetypep0, chargetypep1, chargetypep2;
  std::string Namep0, Namep1, Namep2;
  std::vector<FParticle*> Vecp0, Vecp1, Vecp2;
  std::vector<FParticle**> Combinations;
  TString Typep0, Typep1, Typep2;
  FParticle *p0, *p1, *p2;
  FParticle** combi;
  NPDG_P0MisID = oParticleInput[0]->PdgCode();
  NPDG_P1MisID = oParticleInput[1]->PdgCode();
  NPDG_P2MisID = oParticleInput[2]->PdgCode();
  MassP0MisID = oParticleInput[0]->Mass();
  MassP1MisID = oParticleInput[1]->Mass();
  MassP2MisID = oParticleInput[2]->Mass();
  chargetypep1 = oParticleInput[1]->Charge();
  chargetypep2 = oParticleInput[2]->Charge();
```

```
  chargetypep0 = chargetypep1 + chargetypep2;
  Vecp0 = VParticles[0];
  Vecp1 = VParticles[1];
  Vecp2 = VParticles[2];
  n_p0 = Vecp0.size();
  n_p1 = Vecp1.size();
  n_p2 = Vecp2.size();
  for (i_p0 = 0; i_p0 < n_p0; i_p0++) {
    p0 = Vecp0[i_p0];
    Massp0 = p0->Mass;
    Namep0 = p0->tName;
    NPDGP0 = p0->pdgCode;
    SpinP0 = p0->Spin;
    Typep0 = p0->tType;
    for (i_p1 = 0; i_p1 < n_p1; i_p1++) {
      p1 = Vecp1[i_p1];
      Massp1 = p1->Mass;
      Namep1 = p1->tName;
      NPDGP1 = p1->pdgCode;
      SpinP1 = p1->Spin;
      Typep1 = p1->tType;
      if (p1->Lifetime < globals::detector_tau) { continue; }
      for (i_p2 = 0; i_p2 < n_p2; i_p2++) {
        p2 = Vecp2[i_p2];
        Massp2 = p2->Mass;
        Namep2 = p2->tName;
        NPDGP2 = p2->pdgCode;
        SpinP2 = p2->Spin;
        Typep2 = p2->tType;
        //Conditions
        if (p2->Lifetime < globals::detector_tau) { continue; }
        if (NPDGP0 == NPDG_P0MisID && NPDGP1 == NPDG_P1MisID && NPDGP2 == NPDG_P2MisID) {
            continue; }
        if (!physics2BD(p0, p1, p2)) { continue; }
        Massdiff = Massdifference(Massp0, Massp1, Massp2, MassP0MisID, MassP1MisID,
            MassP2MisID);
        if (Massdiff > globals::deltaW) { continue; }
        combi = (FParticle**)malloc(sizeof(FParticle) * 3);
        combi[0] = p0;
        combi[1] = p1;
        combi[2] = p2;
        Combinations.push_back(combi);
      }
    }
  }
  //Purely for testing purpuses
  //testQ(Combinations.size());
  //
  return Combinations;
}

void MisID(TParticlePDG** iParticles, const char* nfile, TDatabasePDG* mypdg) {
  // environment
  gStyle->SetOptStat(kFALSE);
  gErrorIgnoreLevel = kWarning;
  // variables used
  double BranchingRatio, MassMeson, MassHeavyMeson, MassP1MisID, MassP2MisID, Massp1,
      Massp2, ymax;
  unsigned int ii;
  int NPDGHeavyMeson, NPDGP1, NPDGP2;
  const char* CBranchingRatio;
  const char** allfilenames;
  std::string String_Ratio;
  std::string iMesonTitle, ip1Title, ip2Title;
  std::vector<FParticle**> Combinations;
  std::vector<std::vector<FParticle*>> VParticles;
  TParticlePDG* ipMeson, *ip1, *ip2;
  TString NameHeavyMeson, Namep1, Namep2;
  TH1F* hist_signal, *hist_background;
```

```
TLegendEntry* LEhb;
TCanvas *Canvas1;
FParticle** combi;
// filename
allfilenames = filenames(nfile);
// particledata production
ipMeson = iParticles[0];
ip1 = iParticles[1];
ip2 = iParticles[2];
MassMeson = ipMeson->Mass();
MassP1MisID = ip1->Mass();
MassP2MisID = ip2->Mass();
iMesonTitle = titleName(ipMeson->GetName());
ip1Title = titleName(ip1->GetName());
ip2Title = titleName(ip2->GetName());
VParticles = particledata(ip1->Charge(), ip2->Charge());
Combinations = PCombine(VParticles, iParticles);
hist_signal = MisID2DecayHist(MassMeson, MassP1MisID, MassP2MisID, MassP1MisID,
    MassP2MisID);
hist_signal->SetName("signal");
Canvas1 = new TCanvas("Canvas1", "Canvas1");
Canvas1->Print(allfilenames[0], "pdf");
for (ii = 0; ii < Combinations.size(); ii++)
{
  combi = Combinations[ii];
  NPDGHeavyMeson = combi[0]->pdgCode;
  NameHeavyMeson = combi[0]->tName;
  MassHeavyMeson = combi[0]->Mass;
  NPDGP1 = combi[1]->pdgCode;
  Namep1 = combi[1]->tName;
  Massp1 = combi[1]->Mass;
  NPDGP2 = combi[2]->pdgCode;
  Namep2 = combi[2]->tName;
  Massp2 = combi[2]->Mass;
  //Branching Ratio
  BranchingRatio = FindBranching(NPDGHeavyMeson, NPDGP1, NPDGP2, mypdg);
  if (BranchingRatio < globals::minBR) { continue; }
  std::ostringstream Stream_Ratio;
  Stream_Ratio << "Branching Fraction = " << BranchingRatio;
  String_Ratio = Stream_Ratio.str();
  CBranchingRatio = String_Ratio.c_str();
  // Peaking Background
  hist_background = MisID2DecayHist(MassHeavyMeson, Massp1, Massp2, MassP1MisID,
      MassP2MisID);
  //Title
  std::string title;
  title.append("Reconstruced Invariant Mass Spectrum of ");
  title.append(titleName(NameHeavyMeson));
  title.append(" to ");
  title.append(titleName(Namep1));
  title.append(" and ");
  title.append(titleName(Namep2));
  title.append(" misidentified as ");
  title.append(iMesonTitle);
  title.append(" to ");
  title.append(ip1Title);
  title.append(" and ");
  title.append(ip2Title);
  hist_background->SetTitle(title.c_str());
  //Axis
  ymax = 1.1 * std::max(hist_background->GetMaximum(), hist_signal->GetMaximum());
  hist_background->GetXaxis()->SetRangeUser((MassMeson - globals::deltaW), (MassMeson +
      globals::deltaW));
  hist_background->GetXaxis()->SetTitle("Energy (GeV)");
  hist_background->GetYaxis()->SetRangeUser(0, ymax);
  hist_background->GetYaxis()->SetLabelSize(0);
  hist_background->GetYaxis()->SetTitle("Arbitrary Units");
  //Draw Hists
  hist_background->SetLineColor(2);
```

```
    hist_background->Draw();
    hist_signal->Draw("same");
    //Draw Line at Mesonmass
    TLine massline(MassMeson, 0, MassMeson, ymax);
    massline.Draw("same");
    //Draw Legend
    TLegend legend(0.075, 0.725, 0.275, 0.925);
    legend.AddEntry("hist_signal", "Signal", "l");
    LEhb = legend.AddEntry("hist_background", "Peaking Background", "l");
    LEhb->SetLineColor(2);
    legend.AddEntry((TObject*)0, CBranchingRatio, "");
    legend.Draw("same");
    // Add to file
    Canvas1->Print(allfilenames[1], "pdf");
    delete hist_background;
  }
  Canvas1->Print(allfilenames[2], "pdf");
  delete hist_signal;
  delete Canvas1;
};

void MisID(const char* iMesonName, const char* ip1Name, const char* ip2Name, const char*
    filename) {
  TDatabasePDG *mypdg = new TDatabasePDG();
  TParticlePDG** iParticles;
  mypdg->ReadPDGTable("ParticleDataSets/Rewrite/RewrittenDatabase.txt");
  iParticles = MisIDInput(iMesonName, ip1Name, ip2Name, mypdg);
  MisID(iParticles, filename, mypdg);
  delete mypdg;
}

void MisID(double search_mass_GeV, const char* ip1Name, const char* ip2Name, const char*
    filename) {
  TDatabasePDG *mypdg = new TDatabasePDG();
  TParticlePDG** iParticles;
  const char* iMesonName;
  std::ostringstream strstr;
  std::string str;
  strstr << "{" << search_mass_GeV << " GeV}";
  str = strstr.str();
  iMesonName = str.data();
  mypdg->ReadPDGTable("ParticleDataSets/Rewrite/RewrittenDatabase.txt");
  mypdg->AddParticle(iMesonName, iMesonName, search_mass_GeV, 0, 0.0, 0.0, "manual",
      10000001);
  iParticles = MisIDInput(iMesonName, ip1Name, ip2Name, mypdg);
  MisID(iParticles, filename, mypdg);
  delete mypdg;
}

void MisID(int ip0, int ip1, int ip2, const char* filename) {
  TDatabasePDG *mypdg = new TDatabasePDG();
  mypdg->ReadPDGTable("ParticleDataSets/Rewrite/RewrittenDatabase.txt");
  TParticlePDG** iParticles;
  iParticles = MisIDInput(ip0, ip1, ip2, mypdg);
  MisID(iParticles, filename, mypdg);
  delete mypdg;
}

#endif
```

MisID.hpp :

```
#ifndef MISID_HPP
#define MISID_HPP
void MisID(int ip0, int ip1, int ip2, const char* filename);
void MisID(double search_mass_GeV, const char* ip1Name, const char* ip2Name, const char*
    filename);
```

```
void MisID(const char* Input_NameMeson, const char* Input_Name_P1MisID, const char*
    Input_Name_P2MisID, const char* filename);
#endif
```

A.17 Filename generation

filenames.cpp :

```
#ifndef FILENAMES_CPP
#define FILENAMES_CPP

#include <string>

const char** filenames(const char* fn) {
  const char** filenames;
  const char* fn0;
  const char* fn1;
  const char* fn2;
  static std::string str0, str1, str2;
  str0.clear();
  str1.clear();
  str2.clear();
  str0.append("Results/");
  str0.append(fn);
  str0.append(".pdf[");
  str1.append("Results/");
  str1.append(fn);
  str1.append(".pdf");
  str2.append("Results/");
  str2.append(fn);
  str2.append(".pdf]");
  filenames = (const char**)malloc(sizeof(const char*) * 3);
  fn0 = str0.data();
  fn1 = str1.data();
  fn2 = str2.data();
  filenames[0] = fn0;
  filenames[1] = fn1;
  filenames[2] = fn2;
  return filenames;
}

#endif
```

filenames.hpp :

```
#ifndef FILENAMES_HPP
#define FILENAMES_HPP
const char** filenames(const char* fn);
#endif
```

A.18 Input of misidentification function

MisIDInput.cpp :

```
#ifndef MISIDINPUT_CPP
#define MISIDINPUT_CPP

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "TDatabasePDG.h"

#include "MisIDInput.hpp"

TParticlePDG** MisIDInput(const char* p0name, const char* p1name, const char* p2name,
    TDatabasePDG *mypdg) {
  TParticlePDG** allParticles;
  bool nptr = 0;
  double Charge1MisID, Charge2MisID;
  TParticlePDG* p0 = mypdg->GetParticle(p0name);
  TParticlePDG* p1 = mypdg->GetParticle(p1name);
  TParticlePDG* p2 = mypdg->GetParticle(p2name);
  TString alias, pName;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/Rewrite/particleAlias.xml");
  XMLNodePointer_t alias_xmlnode = engine->DocGetRootElement(doc);
  alias_xmlnode = engine->GetChild(alias_xmlnode);
  while (alias_xmlnode)
  {
    pName = TString(engine->GetAttr(alias_xmlnode, "name"));
    alias = TString(engine->GetAttr(alias_xmlnode, "alias"));
    if (alias.EqualTo(p0name))
    {
      p0 = mypdg->GetParticle(pName);
    }
    if (alias.EqualTo(p1name))
    {
      p1 = mypdg->GetParticle(pName);
    }
    if (alias.EqualTo(p2name))
    {
      p2 = mypdg->GetParticle(pName);
    }
    alias_xmlnode = engine->GetNext(alias_xmlnode);
  }
  if (p0 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p0name);
  }
  if (p1 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p1name);
  }
  if (p2 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p2name);
  }
  if (nptr)
  {
    exit(10);
  }
```

```
  allParticles = (TParticlePDG**)malloc(sizeof(TParticlePDG*) * 3);
  allParticles[0] = p0;
  allParticles[1] = p1;
  allParticles[2] = p2;
  return allParticles;
}

TParticlePDG** MisIDInput(int ip0, int ip1, int ip2, TDatabasePDG *mypdg) {
  TParticlePDG** allParticles;
  bool nptr = 0;
  TParticlePDG* p0 = mypdg->GetParticle(ip0);
  TParticlePDG* p1 = mypdg->GetParticle(ip1);
  TParticlePDG* p2 = mypdg->GetParticle(ip2);
  if (p0 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip0);
  }
  if (p1 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip1);
  }
  if (p2 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip2);
  }
  if (nptr)
  {
    exit(10);
  }
  allParticles = (TParticlePDG**)malloc(sizeof(TParticlePDG*) * 3);
  allParticles[0] = p0;
  allParticles[1] = p1;
  allParticles[2] = p2;
  return allParticles;
}
#endif
```

MisID.hpp :

```
#ifndef MISIDINPUT_HPP
#define MISIDINPUT_HPP
TParticlePDG** MisIDInput(int ip0, int ip1, int ip2, TDatabasePDG *mypdg);
TParticlePDG** MisIDInput(const char *NameMeson, const char *Name_P1MisID, const char *
    Name_P2MisID, TDatabasePDG* pdgptr);
#endif
```

A.19 Misidentification particles

particledata.cpp :

```
#ifndef PARTICLEDATA_CPP
#define PARTICLEDATA_CPP

#include <vector>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"

#include "myparticlestructs.hpp"
#include "initialise_database.hpp"

#include "particledata.hpp"

void PopulateParticleData(FParticle* oParticle, TXMLEngine* engine, XMLNodePointer_t
    particle_xmlnode);

std::vector<std::vector<FParticle*>> particledata(double in_chargetypep1, double
    in_chargetypep2) {
  int chargetype, chargetypep0, chargetypep1, chargetypep2;
  chargetypep1 = std::round(in_chargetypep1);
  chargetypep2 = std::round(in_chargetypep2);
  chargetypep0 = chargetypep1 + chargetypep2;
  std::vector<FParticle*> Particles0;
  std::vector<FParticle*> Particles1;
  std::vector<FParticle*> Particles2;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/Rewrite/RewrittenDatabase.xml
      ");
  XMLNodePointer_t xmlnode = engine->DocGetRootElement(doc);
  xmlnode = engine->GetChild(xmlnode);
  while (xmlnode)
  {
    if (TString(engine->GetNodeName(xmlnode)).EqualTo("particle"))
    {
      if (bTypeFlag(TString(engine->GetAttr(xmlnode, "type"))))
      {
        xmlnode = engine->GetNext(xmlnode);
        continue;
      }
      FParticle* oParticle = new FParticle();
      PopulateParticleData(oParticle, engine, xmlnode);
      chargetype = oParticle->Charge3e;
      if (chargetype == chargetypep0) {
        Particles0.push_back(oParticle);
      }
      if (chargetype == chargetypep1) {
        Particles1.push_back(oParticle);
      }
      if (chargetype == chargetypep2) {
        Particles2.push_back(oParticle);
      }
    }
    xmlnode = engine->GetNext(xmlnode);
  }
  std::vector<std::vector<FParticle*>> allParticles(3);
  allParticles[0] = Particles0;
  allParticles[1] = Particles1;
  allParticles[2] = Particles2;
  delete engine;
  return allParticles;
}
```

```
void PopulateParticleData(FParticle* oParticle, TXMLEngine* engine, XMLNodePointer_t
    particle_xmlnode) {
  oParticle->pdgCode = TString(engine->GetAttr(particle_xmlnode, "id")).Atoi();
  oParticle->tName = TString(engine->GetAttr(particle_xmlnode, "name"));
  oParticle->Spin = TString(engine->GetAttr(particle_xmlnode, "spin")).Atof();
  oParticle->Charge3e = TString(engine->GetAttr(particle_xmlnode, "charge3e")).Atoi();
  oParticle->Mass = TString(engine->GetAttr(particle_xmlnode, "mass_GeV")).Atof();
  oParticle->Lifetime = TString(engine->GetAttr(particle_xmlnode, "lifetime_s")).Atof();
  oParticle->Colortype = TString(engine->GetAttr(particle_xmlnode, "colortype")).Atoi();
  oParticle->tType = TString(engine->GetAttr(particle_xmlnode, "type"));
}

#endif
```

particledata.hpp :

```
#ifndef PARTICLEDATA_HPP
#define PARTICLEDATA_HPP
std::vector<std::vector<FParticle*>> particledata(double chargetype1, double chargetype2)
    ;
#endif
```

A.20 BSM particle types

These particle types are considered BSM. NB: Quarks themselves are SM, but only appear in color singlet bound states.

```
bool bTypeFlag(TString type) {
  return type.EqualTo("Technicolor") ||
    type.EqualTo("Unscope") ||
    type.EqualTo("Sparticle") ||
    type.EqualTo("ColorMultiplet") ||
    type.EqualTo("Quark") ||
    type.EqualTo("Generator");
};
```

A.21 2 Body decay conserved properties

physics2BD.cpp :

```
#ifndef PHYSICS2BD_CPP
#define PHYSICS2BD_CPP

#include <cmath>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"

#include "myparticlestructs.hpp"

#include "physics2BD.hpp"

bool E_concervation(double m0, double m1, double m2) {
  return (m0 > (m1 + m2));
}

bool iCheck(double candidate) {
  return (candidate == std::floor(candidate));
}

bool spinCheck(double spinMeson, double spinP1, double spinP2) {
  return ((iCheck(spinP1) == iCheck(spinP2)) == iCheck(spinMeson));
}

int LeptonNr(int PDGCode, TString Type) {
  int pdgsign = PDGCode / abs(PDGCode);
  return (pdgsign * Type.EqualTo("Lepton"));
}

bool LeptonCheck2Body(FParticle* p0, FParticle* p1, FParticle* p2) {
  return LeptonNr(p0->pdgCode, p0->tType) == LeptonNr(p1->pdgCode, p1->tType) + LeptonNr(
      p2->pdgCode, p2->tType);
}

bool physics2BD(FParticle* p0, FParticle* p1, FParticle* p2) {
  return (E_concervation(p0->Mass, p1->Mass, p2->Mass) &&
        spinCheck(p0->Spin, p1->Spin, p2->Spin) &&
        LeptonCheck2Body(p0, p1, p2));
}

#endif
```

physics2BD.hpp :

```
#ifndef PHYSICS2BD_HPP
#define PHYSICS2BD_HPP
bool physics2BD(FParticle* p0, FParticle* p1, FParticle* p2);
#endif
```

A.22 Mass difference signal and background

This function calculate the difference in invariant mass between a peaking background candidate and its signal.

Massdifference.cpp :

```
#ifndef MASSDIFFERENCE_CPP
#define MASSDIFFERENCE_CPP

#include <cmath>

#include "Massdifference.hpp"

double Massdifference(double MM_bg, double m1_bg, double m2_bg, double MM_signal, double
    m1_signal, double m2_signal) {
  //Gamma = 0 reconstruction
  // If MisID:
  double P1_mID = sqrt((pow(MM_bg, 2) - pow((m1_bg + m2_bg), 2))*(pow(MM_bg, 2) - pow((
      m1_bg - m2_bg), 2))) / (2 * MM_bg);
  double P2_mID = -P1_mID;
  double E_mID = sqrt(pow(m1_signal, 2) + pow(P1_mID, 2)) + sqrt(pow(m2_signal, 2) + pow(
      P2_mID, 2));
  // If no MisID
  double P1 = sqrt((pow(MM_signal, 2) - pow((m1_signal + m2_signal), 2))*(pow(MM_signal,
      2) - pow((m1_signal - m2_signal), 2))) / (2 * MM_signal);
  double P2 = -P1;
  double E_ID = sqrt(pow(m1_signal, 2) + pow(P1, 2)) + sqrt(pow(m2_signal, 2) + pow(P2,
      2));
  //difference
  double Massdiff = abs(E_ID - E_mID);
  return Massdiff;
}
#endif
```

Massdifference.hpp :

```
#ifndef MASSDIFFERENCE_HPP
#define MASSDIFFERENCE_HPP
double Massdifference(double MM_bg, double m1_bg, double m2_bg, double MM_signal, double
    m1_signal, double m2_signal);
#endif
```

A.23 Branching ratio of peaking background

FindBranching.cpp :

```
#ifndef FINDBRANCHING_CPP
#define FINDBRANCHING_CPP

#include <string>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TDatabasePDG.h"
#include "TDecayChannel.h"

#include "FindBranching.hpp"

double FindBranching(TParticlePDG* Meson, TParticlePDG* p1, TParticlePDG* p2) {
  int Npdg_p1 = p1->PdgCode();
  int Npdg_p2 = p2->PdgCode();
  TDecayChannel *channel;
  double ratio = 0.00; // starting value 0 for if not found
  int D1pdg, D2pdg, D1pdgAnti, D2pdgAnti, NDaughters, MaxChannels;
  MaxChannels = Meson->NDecayChannels();
  for (int i = 0; i < MaxChannels; i++) {
    channel = Meson->DecayChannel(i);
    D1pdg = channel->DaughterPdgCode(0);
    D2pdg = channel->DaughterPdgCode(1);
    NDaughters = channel->NDaughters();
    D1pdgAnti = -D1pdg;
    D2pdgAnti = -D2pdg;
    if (((D1pdg == Npdg_p1 && D2pdg == Npdg_p2) ||
      (D1pdg == Npdg_p2 && D2pdg == Npdg_p1) ||
      (D1pdgAnti == Npdg_p1 && D2pdgAnti == Npdg_p2) ||
      (D2pdgAnti == Npdg_p1 && D1pdgAnti == Npdg_p2))
     && NDaughters == 2) {
      ratio = channel->BranchingRatio();
    }
  } return ratio;
};
double FindBranching(const char* NMeson, const char* Name_p1, const char* Name_p2,
    TDatabasePDG* pdgdata) {
  TParticlePDG* Meson = pdgdata->GetParticle(NMeson);
  TParticlePDG* p1 = pdgdata->GetParticle(Name_p1);
  TParticlePDG* p2 = pdgdata->GetParticle(Name_p2);
  if (!Meson || !p1 || !p2) { return 0.00; }
  return FindBranching(Meson, p1, p2);
};
double FindBranching(std::string Input_NMeson, std::string Input_Name_p1, std::string
    Input_Name_p2, TDatabasePDG* pdg) {
  const char* NMeson;
  NMeson = Input_NMeson.c_str();
  const char* Name_p1 = Input_Name_p1.c_str();
  const char* Name_p2 = Input_Name_p2.c_str();
  return FindBranching(NMeson,Name_p1,Name_p2, pdg);
};

double FindBranching(int pdgCodeMeson, int pdgCodep1, int pdgCodep2, TDatabasePDG*
    pdgdata) {
  TParticlePDG* Meson = pdgdata->GetParticle(pdgCodeMeson);
  TParticlePDG* p1 = pdgdata->GetParticle(pdgCodep1);
  TParticlePDG* p2 = pdgdata->GetParticle(pdgCodep2);
  if (!Meson || !p1 || !p2) { return 0.00; }
  return FindBranching(Meson, p1, p2);
};

#endif
```

FindBranching.hpp :

```
#ifndef FINDBRANCHING_HPP
#define FINDBRANCHING_HPP
double FindBranching(const char* NMeson, const char* Name_p1, const char* Name_p2,
    TDatabasePDG* pdg);
double FindBranching(std::string Input_NMeson, std::string Input_Name_p1, std::string
    Input_Name_p2, TDatabasePDG* pdg);
double FindBranching(int pdgCodeMeson, int pdgCodep1, int pdgCodep2, TDatabasePDG* pdg);
#endif
```

A.24 Histogram title names

titleName.cpp :

```
#ifndef TITLENAME_CPP
#define TITLENAME_CPP

#include <cmath>
#include <string>

#include "titleName.hpp"

std::string titleName(std::string Name) {
  unsigned int i, j;
  size_t lengthName, lengthLetter;
  /* "Upsilon" contains "psi" */
  if (Name.find("Upsilon") != std::string::npos || Name.find("upsilon") != std::string::
      npos)
  {
    Name.insert(0, "#");
    return Name;
  }
  std::string letters[46] = { "Alpha","Beta","Gamma","Delta","Epsilon","Zeta","Eta","
      Theta","Iota", "Kappa","Lambda","Mu","Nu","Xi","Omicron","Pi","Rho","Sigma","Tau","
      Phi","Chi","Psi","Omega", "alpha","beta","gamma","delta","epsilon","zeta","eta","
      theta","iota", "kappa","lambda","mu","nu","xi","omicron","pi","rho","sigma","tau",
      "phi","chi","psi","omega" };
  lengthName = Name.length();
  size_t pos;
  for (i = 0; i < 46; i++) {
    lengthLetter = letters[i].length();
    if (lengthLetter>lengthName) { continue; }
    pos = Name.find(letters[i]);
    if (pos != std::string::npos)
    {
      Name.insert(pos, "#");
    }
    pos = Name.find("_");
    while (pos != std::string::npos && pos < lengthName)
    {
      Name.replace(pos, 1, "-");
      pos = Name.find("_");
    }
  }
  return Name;
};
std::string titleName(const char* input_name) {
  std::string sName;
  sName.append(input_name);
  return titleName(sName);
};
#endif
```

titleName.hpp :

```
#ifndef TITLENAME_HPP
#define TITLENAME_HPP
std::string titleName(std::string Name);
std::string titleName(const char* input_name);
#endif
```

A.25 Boost distribution

beta_distribution.cpp :

```
#ifndef BETA_DISTRIBUTION_CPP
#define BETA_DISTRIBUTION_CPP

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TF1.h"

#include "beta_distribution.hpp"

double beta_distribution() {
  TF1* gamma_distribution = new TF1("gamma_distribution", "exp(1/(x-10))", 10.01, 100.0 )
      ;
  double gamma = gamma_distribution->GetRandom();
  double betaz = sqrt(1 - 1 / (pow(gamma, 2)));
  delete gamma_distribution;
  return betaz;
};
#endif
```

beta_distribution.hpp :

```
#ifndef BETA_DISTRIBUTION_HPP
#define BETA_DISTRIBUTION_HPP
double beta_distribution();
#endif
```

A.26 cascade struct

```
struct Fcascade_tstep {
  TParticlePDG* mother;
  TParticlePDG* decaying_daughter;
  TDecayChannel* Channel;
  int ChannelID;
};

struct Fcascade_t {
  std::vector<Fcascade_tstep*> cascade;
  double predictmass;
  double combined_BR;
  int nr_P_missing;
  int _p1steps;
  int _p2steps;
  TDatabasePDG* pdgptr;
  const char* pdgpath;
  TParticlePDG** iParticles;
  Fcascade_t() :predictmass(0), combined_BR(1), nr_P_missing(0), _p1steps(0), _p2steps(0)
      , pdgptr(nullptr) {}
};
```

A.27 Finding first cascade particle

find_cascade.cpp :

```
#ifndef FIND_CASCADE_CPP
#define FIND_CASCADE_CPP

#include <vector>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TXMLEngine.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "THashList.h"
#include "TDecayChannel.h"
#include "TCanvas.h"

#include "myparticlestructs.hpp"
#include "globals.hpp"
#include "find_daughter.hpp"
#include "visualizeCascade.hpp"

#include "find_cascade.hpp"

void setupcascade(TParticlePDG** iParticles, TDatabasePDG* mypdg, const char* DBpath,
    const char* tmpfilename);
void find_mother(Fcascade* oCascade, TParticlePDG** funcParticles, const char*
    tmpfilename);
void find_channels(TParticlePDG* t_particle, Fcascade* oCascade, TParticlePDG**
    funcParticles, const char* tmpfilename);
void found_product(TDecayChannel* t_channel, TParticlePDG* mParticle, int ch_id, int
    product_id, Fcascade* oCascade, TParticlePDG** funcParticles, const char* tmpfilename
    );

void find_cascade(const char* p0name, const char* p1name, const char* p2name, const char*
     tmpfilename) {
  const char* DBpath = "ParticleDataSets/Rewrite/RewrittenDatabase.txt";
  bool nptr = 0;
  TDatabasePDG *mypdg = new TDatabasePDG();
  mypdg->ReadPDGTable(DBpath);
  TParticlePDG* p0 = mypdg->GetParticle(p0name);
  TParticlePDG* p1 = mypdg->GetParticle(p1name);
  TParticlePDG* p2 = mypdg->GetParticle(p2name);
  TString alias, pName;
  TParticlePDG** iParticles;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile("ParticleDataSets/Rewrite/particleAlias.xml");
  XMLNodePointer_t alias_xmlnode = engine->DocGetRootElement(doc);
  alias_xmlnode = engine->GetChild(alias_xmlnode);
  while (alias_xmlnode)
  {
    pName = TString(engine->GetAttr(alias_xmlnode, "name"));
    alias = TString(engine->GetAttr(alias_xmlnode, "alias"));
    if (alias.EqualTo(p0name))
    {
      p0 = mypdg->GetParticle(pName);
    }
    if (alias.EqualTo(p1name))
    {
      p1 = mypdg->GetParticle(pName);
    }
    if (alias.EqualTo(p2name))
    {
      p2 = mypdg->GetParticle(pName);
    }
```

```
    alias_xmlnode = engine->GetNext(alias_xmlnode);
  }
  if (p0 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p0name);
  }
  if (p1 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p1name);
  }
  if (p2 == nullptr)
  {
    nptr = 1;
    printf("%s not found in database\n", p2name);
  }
  if (nptr)
  {
    exit(10);
  }
  iParticles = (TParticlePDG**)malloc(sizeof(TParticlePDG) * 3);
  iParticles[0] = p0;
  if (p1->Mass() < p2->Mass())
  {
    iParticles[2] = p1;
    iParticles[1] = p2;
  }
  else
  {
    iParticles[1] = p1;
    iParticles[2] = p2;
  }
  setupcascade(iParticles, mypdg, DBpath, tmpfilename);
};

void find_cascade(int ip0, int ip1, int ip2, const char* tmpfilename) {
  const char* DBpath = "ParticleDataSets/Rewrite/RewrittenDatabase.txt";
  bool nptr = 0;
  TDatabasePDG *mypdg = new TDatabasePDG();
  mypdg->ReadPDGTable(DBpath);
  TParticlePDG* p0 = mypdg->GetParticle(ip0);
  TParticlePDG* p1 = mypdg->GetParticle(ip1);
  TParticlePDG* p2 = mypdg->GetParticle(ip2);
  if (p0 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip0);
  }
  if (p1 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip1);
  }
  if (p2 == nullptr)
  {
    nptr = 1;
    printf("PDGCode: %d not found in database\n", ip2);
  }
  if (nptr)
  {
    exit(10);
  }
  TParticlePDG** iParticles;
  if (p1->Mass() < p2->Mass())
  {
    p1 = mypdg->GetParticle(ip2);
    p2 = mypdg->GetParticle(ip1);
  }
```

```
  iParticles = (TParticlePDG**)malloc(sizeof(TParticlePDG) * 3);
  iParticles[0] = p0;
  iParticles[1] = p1;
  iParticles[2] = p2;
  setupcascade(iParticles, mypdg, DBpath, tmpfilename);
};

void setupcascade(TParticlePDG** iParticles, TDatabasePDG* mypdg, const char* DBpath,
    const char* tmpfilename) {
  Fcascade* oCascade = new Fcascade();
  oCascade->pdgpath = DBpath;
  oCascade->pdgptr = mypdg;
  oCascade->iParticles = iParticles;
  find_mother(oCascade, iParticles, tmpfilename);
  delete mypdg;
}

void find_mother(Fcascade* oCascade, TParticlePDG** funcParticles, const char*
    tmpfilename) {
  TDatabasePDG* mypdg = oCascade->pdgptr;
  TParticlePDG* t_particle = 0;
  TIter next(mypdg->ParticleList());
  while (t_particle = (TParticlePDG*)next())
  {
    if (c_condition(t_particle, funcParticles[1])) { continue; }
    find_channels(t_particle, oCascade, funcParticles, tmpfilename);
  }
}

void find_channels(TParticlePDG* t_particle, Fcascade* oCascade, TParticlePDG**
    funcParticles, const char* tmpfilename) {
  TDatabasePDG* mypdg = oCascade->pdgptr;
  int i_ch, j, iproduct, n_channels, n_products, pdgProduct, pdgin1;
  double tot_mass;
  TDecayChannel* t_channel;
  bool flag;
  bool found = 0;
  TParticlePDG* oProduct;
  pdgin1 = funcParticles[1]->PdgCode();
  n_channels = t_particle->NDecayChannels();
  for (i_ch = 0; i_ch < n_channels; i_ch++)
  {
    flag = 0;
    found = 0;
    tot_mass = 0;
    t_channel = t_particle->DecayChannel(i_ch);
    n_products = t_channel->NDaughters();
    for (j = 0; j < n_products; j++)
    {
      pdgProduct = t_channel->DaughterPdgCode(j);
      oProduct = mypdg->GetParticle(pdgProduct);
      tot_mass += oProduct->Mass();
      if (pdgin1 == pdgProduct)
      {
        found = 1;
        iproduct = j;
      }
      flag |= bTypeFlag(oProduct->ParticleClass());
    }
    flag |= (tot_mass >= t_particle->Mass());
    if (found && !flag)
    {
      found_product(t_channel, t_particle, i_ch, iproduct, oCascade, funcParticles,
          tmpfilename);
      found = 0;
    }
  }
}
```

```
void found_product(TDecayChannel* t_channel, TParticlePDG* mParticle, int ch_id, int
    product_id, Fcascade* oCascade, TParticlePDG** funcParticles, const char* tmpfilename
    ) {
  TDatabasePDG* mypdg = oCascade->pdgptr;
  TParticlePDG** oPnew;
  int npart;
  double orig_BR;
  npart = t_channel->NDaughters() - 1;
  Fcascadestep* step = new Fcascadestep({ mParticle, funcParticles[1], t_channel, ch_id
      });
  oCascade->cascade.push_back(step);
  orig_BR = oCascade->combined_BR;
  oCascade->combined_BR *= t_channel->BranchingRatio();
  oCascade->nr_P_missing += npart;
  oCascade->_p1steps++;
  if (oCascade->nr_P_missing <= (cascade::max_P_missing + 1))
  {
    Fcascade orig_casc = *oCascade;
    find_daughter(oCascade, funcParticles, product_id, tmpfilename);
    new (oCascade) Fcascade(orig_casc);
  }

  if (oCascade->nr_P_missing <= cascade::max_P_missing)
  {
    oPnew = (TParticlePDG**)malloc(sizeof(TParticlePDG) * 3);
    oPnew[0] = funcParticles[0];
    oPnew[1] = mParticle;
    oPnew[2] = funcParticles[2];
    find_mother(oCascade, oPnew, tmpfilename);
  }
  oCascade->cascade.pop_back();
  oCascade->combined_BR = orig_BR;
  oCascade->nr_P_missing -= npart;
  oCascade->_p1steps--;
}

bool c_condition(TParticlePDG* oParticle, TParticlePDG* iParticle1) {
  return (oParticle->Width() > cascade::max_tau || bTypeFlag(TString(oParticle->
      ParticleClass())) || (oParticle->Mass() <= iParticle1->Mass()));
}

#endif
```

find_cascade.hpp :

```
#ifndef FIND_CASCADE_HPP
#define FIND_CASCADE_HPP
void find_cascade(const char* p0name, const char* p1name, const char* p2name, const char*
    tmpfilename = "cascade_tmp");
void find_cascade(int ip0, int ip1, int ip2, const char* tmpfilename = "cascade_tmp");
bool c_condition(TParticlePDG* oParticle, TParticlePDG* iParticle1);
#endif
```

A.28 Finding second cascade particle

find_daughter.cpp :

```
#ifndef FIND_DAUGHTER_CPP
#define FIND_DAUGHTER_CPP

#include <map>
#include <vector>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "TDecayChannel.h"

#include "globals.hpp"
#include "myparticlestructs.hpp"
#include "titleName.hpp"
#include "find_cascade.hpp"
#include "visualizeCascade.hpp"
#include "CascFile.hpp"

#include "find_daughter.hpp"

void find_daughter_stepwise(Fcascade* oCascade, TParticlePDG** funcParticles, const char*
    tmpfilename) {
  TDatabasePDG* mypdg = oCascade->pdgptr;
  TParticlePDG* oddaughter = oCascade->cascade.back()->decaying_daughter;
  TParticlePDG* mydaughter = funcParticles[2];
  int vd_size, j, nch, i;
  int opdg, mydpdg = mydaughter->PdgCode();
  double orig_BR, tot_mass;
  bool found, flag;
  TParticlePDG* oParticle;
  TDecayChannel* oChannel;
  nch = oddaughter->NDecayChannels();
  for (i = 0; i < nch; i++)
  {
    oChannel = oddaughter->DecayChannel(i);
    vd_size = oChannel->NDaughters();
    if ((oCascade->nr_P_missing + (vd_size - 1)) > cascade::max_P_missing) { continue; }
    tot_mass = 0;
    flag = false;
    found = false;
    for (j = 0; j < vd_size ; j++)
    {
      opdg = oChannel->DaughterPdgCode(j);
      oParticle = mypdg->GetParticle(opdg);
      found |= (opdg == mydpdg);
      tot_mass += oParticle->Mass();
      flag |= bTypeFlag(oParticle->ParticleClass());
    }
    if ((tot_mass >= oddaughter->Mass()) || flag) { continue; }
    oCascade->nr_P_missing += (vd_size - 1);
    orig_BR = oCascade->combined_BR;
    oCascade->combined_BR *= oChannel->BranchingRatio();
    for (j = 0; j < vd_size; j++)
    {
      opdg = oChannel->DaughterPdgCode(j);
      oParticle = mypdg->GetParticle(opdg);
      if (c_condition(oParticle, mydaughter)) { continue; }
      Fcascadestep* step = new Fcascadestep({ oddaughter, oParticle, oChannel, i });
      oCascade->cascade.push_back(step);
      oCascade->_p2steps++;
      find_daughter_stepwise(oCascade, funcParticles, tmpfilename);
```

```
      oCascade->cascade.pop_back();
      oCascade->_p2steps--;
    }
    if (found)
    {
      Fcascadestep* step = new Fcascadestep({ oddaughter, mydaughter, oChannel, i });
      oCascade->cascade.push_back(step);
      oCascade->_p2steps++;
      printcascadetofile(oCascade, tmpfilename);
      oCascade->cascade.pop_back();
      oCascade->_p2steps--;
    }
    oCascade->nr_P_missing -= (vd_size - 1);
    oCascade->combined_BR = orig_BR;
  }
};

void find_daughter(Fcascade* oCascade, TParticlePDG** funcParticles, int i_product1,
    const char* tmpfilename) {
  TDatabasePDG* mypdg = oCascade->pdgptr;
  Fcascadestep* mystep = oCascade->cascade.back();
  TParticlePDG* odaughter, *mydaughter = funcParticles[2];
  int i, od_id, dd_id, my_id, nd, ch_id;
  TParticlePDG* decaying_daughter = mystep->decaying_daughter;
  TDecayChannel* oChannel = mystep->Channel;
  TParticlePDG* mParticle = mystep->mother;
  ch_id = mystep->ChannelID;
  bool found = false;
  my_id = mydaughter->PdgCode();
  dd_id = decaying_daughter->PdgCode();
  oCascade->nr_P_missing--;
  nd = oChannel->NDaughters();
  for (i = 0; i < nd; i++)
  {
    if (i == i_product1) { continue; }
    od_id = oChannel->DaughterPdgCode(i);
    odaughter = mypdg->GetParticle(od_id);
    if (od_id == my_id && !found)
    {
      found = true;
      Fcascadestep* step = new Fcascadestep({ mParticle, mydaughter, oChannel, ch_id });
      oCascade->cascade.push_back(step);
      oCascade->_p2steps++;
      printcascadetofile(oCascade, tmpfilename);
      oCascade->cascade.pop_back();
      oCascade->_p2steps--;
      continue;
    }
    if (c_condition(odaughter, mydaughter)) { continue; }
    if (oCascade->nr_P_missing < cascade::max_P_missing)
    {
      Fcascadestep* step = new Fcascadestep({ mParticle, odaughter, oChannel, ch_id });
      oCascade->cascade.push_back(step);
      oCascade->_p2steps++;
      find_daughter_stepwise(oCascade, funcParticles, tmpfilename);
      oCascade->cascade.pop_back();
      oCascade->_p2steps--;
    }
  }
  oCascade->nr_P_missing++;
};

#endif
```

find_daughter.hpp :

```
#ifndef FIND_DAUGHTER_HPP
```

```
#define FIND_DAUGHTER_HPP
void find_daughter(Fcascade* oCascade, TParticlePDG** iParticles, int i_product1, const
    char* tmpfilename);
#endif
```

A.29 Cascade file

CascFile.cpp :

```
#ifndef CascFile_CPP
#define CascFile_CPP

#include <vector>
#include <string>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TString.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "TDecayChannel.h"
#include "TXMLEngine.h"

#include "myparticlestructs.hpp"
#include "masspredict.hpp"

#include "CascFile.hpp"

void getCascCh(Fcascade* oCasc, TXMLEngine* engine, XMLNodePointer_t xmlnode,
    TDatabasePDG* pdgdata);

std::vector<Fcascade*> readcascadesfromfile(TDatabasePDG* pdgdata, const char* filename)
    {
  TString fn_t;
  fn_t.Form("Results/%s.xml", filename);
  filename = fn_t.Data();
  std::vector<Fcascade*> Vcasc;
  Fcascade* oCasc;
  TParticlePDG** iParticles;
  int ip0, ip1, ip2;
  const char* path;
  TXMLEngine* engine = new TXMLEngine();
  XMLDocPointer_t doc = engine->ParseFile(filename);
  if (!doc)
  {
    printf("%s not found", filename);
    exit(30);
  }
  XMLNodePointer_t xmlnode = engine->DocGetRootElement(doc);
  path = engine->GetAttr(xmlnode, "DBpath");
  pdgdata->ReadPDGTable(path);
  ip0 = TString(engine->GetAttr(xmlnode, "p0")).Atoi();
  ip1 = TString(engine->GetAttr(xmlnode, "p1")).Atoi();
  ip2 = TString(engine->GetAttr(xmlnode, "p2")).Atoi();
  TParticlePDG* p0 = pdgdata->GetParticle(ip0);
  TParticlePDG* p1 = pdgdata->GetParticle(ip1);
  TParticlePDG* p2 = pdgdata->GetParticle(ip2);
  iParticles = (TParticlePDG**)malloc(sizeof(TParticlePDG) * 3);
  iParticles[0] = p0;
  iParticles[1] = p1;
  iParticles[2] = p2;
  while (xmlnode)
  {
    oCasc = new Fcascade();
    oCasc->pdgpath = path;
    oCasc->pdgptr = pdgdata;
    oCasc->iParticles = iParticles;
    oCasc->combined_BR = TString(engine->GetAttr(xmlnode, "BR")).Atof();
    oCasc->predictmass = TString(engine->GetAttr(xmlnode, "W")).Atof();
    oCasc->nr_P_missing = TString(engine->GetAttr(xmlnode, "Pmiss")).Atoi();
    getCascCh(oCasc, engine, xmlnode, pdgdata);
    Vcasc.push_back(oCasc);
```

```
    xmlnode = engine->GetNext(xmlnode);
  }
  return Vcasc;
}

void getCascCh(Fcascade* oCasc, TXMLEngine* engine, XMLNodePointer_t xmlnode,
    TDatabasePDG* pdgdata) {
  int ch_id;
  std::vector<Fcascadestep*> sCasc, revCasc;
  Fcascadestep* step;
  TParticlePDG* t_mother;
  xmlnode = engine->GetChild(xmlnode);
  oCasc->_p2steps = TString(engine->GetAttr(xmlnode, "myid")).Atoi() + 1;
  while (xmlnode)
  {
    t_mother = pdgdata->GetParticle(TString(engine->GetAttr(xmlnode, "mother")).Atoi());
    step = new Fcascadestep();
    ch_id = TString(engine->GetAttr(xmlnode, "ch_id")).Atoi();
    step->mother = t_mother;
    step->ChannelID = ch_id;
    step->Channel = t_mother->DecayChannel(ch_id);
    step->decaying_daughter = pdgdata->GetParticle(TString(engine->GetAttr(xmlnode, "
        dec_daughter")).Atoi());
    revCasc.push_back(step);
    xmlnode = engine->GetNext(xmlnode);
  }
  oCasc->_p1steps = revCasc.size() - oCasc->_p2steps;
  while (!revCasc.empty())
  {
    sCasc.push_back(revCasc.back());
    revCasc.pop_back();
  }
  oCasc->cascade = sCasc;
}

void printcascadetofile(Fcascade* oCascade, const char* filename) {
  TString fn_t;
  fn_t.Form("Results/%s.xml", filename);
  filename = fn_t.Data();
  FILE* fptr;
  std::vector<Fcascadestep*> sCascade;
  Fcascadestep* step;
  TDecayChannel* t_ch;
  int id, pdgmother, itr;
  static int casc_count = 0;
  if (casc_count == 0) { fptr = fopen(filename, "w"); }
  else { fptr = fopen(filename, "a"); }
  casc_count++;
  oCascade->predictmass = masspredict(oCascade);
  fprintf(fptr, "<Cascade ");
  fprintf(fptr, "id=\"%d\" ", casc_count);
  fprintf(fptr, "Pmiss=\"%d\" ", oCascade->nr_P_missing);
  fprintf(fptr, "W=\"%f\" ", oCascade->predictmass);
  fprintf(fptr, "BR=\"%g\" ", oCascade->combined_BR);
  fprintf(fptr, "p0=\"%d\" ", oCascade->iParticles[0]->PdgCode());
  fprintf(fptr, "p1=\"%d\" ", oCascade->iParticles[1]->PdgCode());
  fprintf(fptr, "p2=\"%d\" ", oCascade->iParticles[2]->PdgCode());
  fprintf(fptr, "DBpath=\"%s\" ", oCascade->pdgpath);
  fprintf(fptr, ">\n");
  id = oCascade->_p2steps;
  sCascade = oCascade->cascade;
  pdgmother = 0;
  while (!sCascade.empty())
  {
    step = sCascade.back();
    id -= (pdgmother != sCascade.back()->mother->PdgCode());
    fprintf(fptr, "<Channel ");
    fprintf(fptr, "myid=\"%d\" ", id);
    fprintf(fptr, "ch_id=\"%d\" ", step->ChannelID);
```

```
    fprintf(fptr, "mother=\"%d\" ", step->mother->PdgCode());
    fprintf(fptr, "pdgdaughters=\" ");
    t_ch = step->Channel;
    for (itr = 0; itr < t_ch->NDaughters(); itr++)
    {
      fprintf(fptr, "%d ", t_ch->DaughterPdgCode(itr));
    }
    fprintf(fptr, "\" ");
    fprintf(fptr, "dec_daughter=\"%d\" ", step->decaying_daughter->PdgCode());
    fprintf(fptr, "/>\n");
    pdgmother = step->mother->PdgCode();
    sCascade.pop_back();
  }
  fprintf(fptr, "</Cascade>\n\n");
  fclose(fptr);
}

#endif
```

CascFile.hpp :

```
#ifndef CascFile_HPP
#define CascFile_HPP
std::vector<Fcascade*> readcascadesfromfile(TDatabasePDG* pdgdata, const char* filename);
void printcascadetofile(Fcascade* oCascade, const char* filename);
#endif
```

A.30 Invariant mass of cascade

masspredict.cpp :

```
#ifndef MASSPREDICT_CPP
#define MASSPREDICT_CPP

#include <vector>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "TDecayChannel.h"

#include "myparticlestructs.hpp"

#include "masspredict.hpp"

double* LVdecay(double M, double m1, double m2, double beta) {
  double* LV;
  double E1, p1, magp1, Mk, m1k, m2k, gamma;
  Mk = M * M;
  m1k = m1 * m1;
  m2k = m2 * m2;
  LV = (double*)malloc(sizeof(double) * 2);
  // 2 body decay
  E1 = (Mk - m2k + m1k) / (2.00 * M);
  magp1 = sqrt((Mk - pow((m1 + m2), 2)) * (Mk - pow((m1 - m2), 2))) / (2.00 * M);
  // boost
  gamma = 1 / sqrt(1 - beta * beta);
  p1 = gamma * (magp1 + beta * E1);
  E1 = gamma * (E1 + beta * magp1);
  // make LV
  magp1 = p1;
  LV[0] = magp1;
  LV[1] = E1;
  return LV;
}

double* takestep(Fcascadestep* step, double* LV, TDatabasePDG* pdgdata) {
  int nd, itr;
  double massDD, massM, missmass;
  TDecayChannel* ch;
  massM = step->mother->Mass();
  massDD = step->decaying_daughter->Mass();
  ch = step->Channel;
  nd = ch->NDaughters();
  missmass = -1 * massDD;
  for (itr = 0; itr < nd; itr++)
  {
    missmass += pdgdata->GetParticle(ch->DaughterPdgCode(itr))->Mass();
  }
  LV = LVdecay(massM, massDD, missmass, LV[0]/LV[1]);
  return LV;
}


double masspredict(Fcascade* oCascade) {
  int itr, pdgcode, p1steps, p2steps, steps, npart;
  double p1mass, p2mass, missmass, p0mass, W;
  double* LVp1, *LVp2;
  TDatabasePDG* pdgdata;
  TParticlePDG* tMother, *tDD1, *tDD2;
  TDecayChannel* ch0;
  Fcascadestep* step;
  pdgdata = oCascade->pdgptr;
```

```
  steps = oCascade->cascade.size();
  p1steps = oCascade->_p1steps;
  p2steps = oCascade->_p2steps;
  step = oCascade->cascade[p1steps - 1];
  tMother = step->mother;
  tDD1 = step->decaying_daughter;
  tDD2 = oCascade->cascade[steps - p2steps]->decaying_daughter;
  p0mass = tMother->Mass();
  p1mass = tDD1->Mass();
  p2mass = tDD2->Mass();
  ch0 = step->Channel;
  npart = ch0->NDaughters();
  missmass = -1 * (p1mass + p2mass);
  for (itr = 0; itr < npart; itr++)
  {
    pdgcode = ch0->DaughterPdgCode(itr);
    missmass += pdgdata->GetParticle(pdgcode)->Mass();
  }
  LVp1 = LVdecay(p0mass - missmass, p1mass, p2mass, 0);
  LVp2 = LVdecay(p0mass - missmass, p2mass, p1mass, 0);
  for (itr = p1steps - 2; itr >= 0; itr--)
  {
    step = oCascade->cascade[itr];
    LVp1 = takestep(step, LVp1, pdgdata);
  }
  for (itr = steps - p2steps + 1; itr < steps; itr++)
  {
    step = oCascade->cascade[itr];
    LVp2 = takestep(step, LVp2, pdgdata);
  }
  W = sqrt(pow((LVp1[1] + LVp2[1]), 2) - pow((LVp1[0] - LVp2[0]), 2));
  return W;
}

#endif
```

masspredict.hpp :

```
#ifndef MASSPREDICT_HPP
#define MASSPREDICT_HPP
double masspredict(Fcascade* oCascade);
#endif
```

A.31 Visualizing cascades

visualiseCascade.cpp :

```
#ifndef PRINTFULLCASCADE_CPP
#define PRINTFULLCASCADE_CPP

#include <map>
#include <vector>
#include <sstream>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TError.h"
#include "TString.h"
#include "TCanvas.h"
#include "TLatex.h"
#include "TLine.h"
#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "THashList.h"
#include "TDecayChannel.h"

#include "myparticlestructs.hpp"
#include "globals.hpp"
#include "titleName.hpp"
#include "masspredict.hpp"
#include "filenames.hpp"
#include "CascFile.hpp"
#include "simCascade.hpp"

#include "visualizeCascade.hpp"

void visualizeCascade(std::vector<Fcascade*> Vcasc, const char* newfilename, bool
    simulation, double massMother, const char* sourcefilename);
void printfullcascade(Fcascade* oCascade, const char* filename);
void tipping_point(std::vector<Fcascadestep*> sCascade, int dd1pdg, int xmother, int
    ycoor, TDatabasePDG* mypdg, TCanvas* mycanvas);
void restcascade(std::vector<Fcascadestep*> sCascade, int xcoor, int ycoor, TDatabasePDG*
     mypdg, TCanvas* mycanvas);

void visualizeCascade(const char* newfilename, bool simulation, const char*
    sourcefilename) {
  TDatabasePDG* pdgdata = new TDatabasePDG();
  std::vector<Fcascade*> Vcasc = readcascadesfromfile(pdgdata, sourcefilename);
  visualizeCascade(Vcasc, newfilename, simulation, Vcasc[0]->iParticles[0]->Mass(),
      sourcefilename);
  delete pdgdata;
}

void visualizeCascade(const char* newfilename, bool simulation, double newmass, const
    char* sourcefilename) {
  TDatabasePDG* pdgdata = new TDatabasePDG();
  std::vector<Fcascade*> Vcasc = readcascadesfromfile(pdgdata, sourcefilename);
  visualizeCascade(Vcasc, newfilename, simulation, newmass, sourcefilename);
  delete pdgdata;
}

void visualizeCascade(std::vector<Fcascade*> Vcasc, const char* newfilename, bool
    simulation, double massMother, const char* sourcefilename) {
  gErrorIgnoreLevel = kWarning;
  TCanvas c1("c1");
  const char** fnames;
  fnames = filenames(newfilename);
  c1.Print(fnames[0], "pdf");
  Fcascade* oCascade;
  for (int i = 0; i < Vcasc.size(); i++)
```

```
  {
    oCascade = Vcasc[i];
    if ((oCascade->predictmass < massMother - globals::deltaW || oCascade->predictmass >
        massMother + globals::deltaW) ||
      (oCascade->combined_BR < globals::minBR))
    {
      continue;
    }
    printfullcascade(oCascade, fnames[1]);
    if (simulation)
    {
      simCascade(oCascade, fnames[1]);
    }
  }
  c1.Print(fnames[2], "pdf");
}

void printfullcascade(Fcascade* oCascade, const char* filename) {
  int j, jcount, h, wmax, n_products, ycoor, xcoor, ymother, yproduct, ddpdg, iproduct,
      ytxt;
  int grid = 100;
  bool found;
  TString pName;
  Fcascadestep* step;
  TParticlePDG* mother, *oProduct;
  TDatabasePDG* mypdg;
  std::vector<Fcascadestep*> scascade = oCascade->cascade;
  h = (oCascade->nr_P_missing + 4) * grid;
  wmax = (scascade.size() + 2) * grid;
  ycoor = h - grid;
  xcoor = wmax - grid;
  ddpdg = scascade.back()->decaying_daughter->PdgCode();
  TCanvas* mycanvas = new TCanvas();
  mycanvas->Range(0, 0, wmax, h);
  TLatex txt;
  txt.SetTextAlign(22);
  txt.SetTextSize(0.025);
  TLine* line;
  mypdg = oCascade->pdgptr;
  while (!scascade.empty())
  {
    step = scascade.back();
    scascade.pop_back();
    ddpdg = step->decaying_daughter->PdgCode();
    mother = step->mother;
    n_products = step->Channel->NDaughters();
    jcount = 1;
    found = 0;
    ymother = ycoor - (((n_products - 1)  * grid) / 2);
    if (mother->PdgCode() == scascade.back()->mother->PdgCode())
    {
      txt.SetTextSize(0.015);
      ytxt = ycoor - (n_products - 1) * grid;
      txt.DrawLatex(xcoor, ytxt, Form("Number of missing particles: %d", oCascade->
          nr_P_missing));
      txt.DrawLatex(xcoor, ytxt - 0.5 * grid, Form("Max inv mass value: %f GeV", oCascade
          ->predictmass));
      txt.DrawLatex(xcoor, ytxt - 1 * grid, Form("Combined branching ratio: %e", oCascade
          ->combined_BR));
      xcoor -= grid;
      tipping_point(scascade, ddpdg, xcoor, ycoor, mypdg, mycanvas);
      break;
    }
    for (j = 0; j < n_products; j++)
    {
      iproduct = step->Channel->DaughterPdgCode(j);
      oProduct = mypdg->GetParticle(iproduct);
      if (iproduct == ddpdg && !found)
      {
```

```
        yproduct = ycoor;
        found = 1;
      }
      else
      {
        yproduct = ycoor - jcount * grid;
        jcount++;
      }
      txt.DrawLatex(xcoor, yproduct, titleName(oProduct->GetName()).data());
      line = new TLine(xcoor - grid, ymother, xcoor, yproduct);
      line->Draw();
    }
    xcoor -= grid;
    ycoor = ymother;
  }
  mycanvas->Update();
  mycanvas->Print(filename, "pdf");
  delete mycanvas;
}

void tipping_point(std::vector<Fcascadestep*> sCascade, int dd1pdg, int xmother, int
    ycoor, TDatabasePDG* mypdg, TCanvas* mycanvas) {
  TLatex txt;
  int iproduct, dd2pdg, xproduct, ydd, yproduct, ymother, n_products, j, jcount = 1;
  int gridmedian, grid = 100;
  Fcascadestep* step;
  TParticlePDG* mother, *oProduct;
  std::vector<FParticle*> products;
  bool founddd1 = 0, founddd2 = 0;
  TLine* line;
  txt.SetTextAlign(22);
  txt.SetTextSize(0.025);
  step = sCascade.back();
  mother = step->mother;
  n_products = step->Channel->NDaughters();
  dd2pdg = step->decaying_daughter->PdgCode();
  gridmedian = ((n_products - 1)  * grid) / 2;
  ymother = ycoor - gridmedian;
  ydd = ymother - gridmedian;

  for (j = 0; j < n_products; j++)
  {
    iproduct = step->Channel->DaughterPdgCode(j);
    oProduct = mypdg->GetParticle(iproduct);
    if (iproduct == dd1pdg && !founddd1)
    {
      xproduct = xmother + grid;
      yproduct = ycoor;
      founddd1 = 1;
    }
    else if (iproduct == dd2pdg && !founddd2)
    {
      xproduct = xmother - grid;
      yproduct = ydd;
      founddd2 = 1;
    }
    else
    {
      xproduct = xmother + grid;
      yproduct = ycoor - jcount * grid;
      jcount++;
    }
    txt.DrawLatex(xproduct, yproduct, titleName(oProduct->GetName()).data());
    line = new TLine(xmother, ymother, xproduct, yproduct);
    line->Draw();
  }
  txt.SetTextColor(kBlue);
  txt.DrawLatex(xmother, ymother, titleName(mother->GetName()).data());
  xmother -= grid;
```

```
  sCascade.pop_back();
  if (!sCascade.empty())
  {
    restcascade(sCascade, xmother, ydd, mypdg, mycanvas);
  }
}


void restcascade(std::vector<Fcascadestep*> sCascade, int xcoor, int ycoor, TDatabasePDG*
     mypdg, TCanvas* mycanvas) {
  int iproduct, j, jcount, n_products, ydd, yproduct, mpdg, ddpdg;
  int gridmedian, grid = 100;
  bool found;
  TString pName;
  Fcascadestep* step;
  TParticlePDG* mother, *oProduct;
  ddpdg = sCascade.back()->decaying_daughter->PdgCode();
  mpdg = 0;

  TLatex txt;
  txt.SetTextAlign(22);
  txt.SetTextSize(0.025);
  TLine* line;

  while (!sCascade.empty())
  {
    xcoor -= grid;
    step = sCascade.back();
    ddpdg = step->decaying_daughter->PdgCode();
    mother = step->mother;
    n_products = step->Channel->NDaughters();
    gridmedian = ((n_products - 1)  * grid) / 2;
    jcount = 1;
    found = 0;
    for (j = 0; j < n_products; j++)
    {
      iproduct = step->Channel->DaughterPdgCode(j);
      oProduct = mypdg->GetParticle(iproduct);
      if (iproduct == ddpdg && !found)
      {
        ydd = ycoor - gridmedian;
        yproduct = ydd;
        found = 1;
      }
      else
      {
        yproduct = ycoor - gridmedian + jcount * grid;
        jcount++;
      }
      txt.DrawLatex(xcoor, yproduct, titleName(oProduct->GetName()).data());
      line = new TLine(xcoor + grid, ycoor, xcoor, yproduct);
      line->Draw();
    }
    ycoor -= gridmedian;
    mpdg = mother->PdgCode();
    sCascade.pop_back();
  }
}

#endif
```

visualiseCascade.hpp :

```
#ifndef PRINTFULLCASCADE_HPP
#define PRINTFULLCASCADE_HPP
void visualizeCascade(const char* newfilename, bool simulation, const char*
    sourcefilename = "cascade_tmp");
```

```
void visualizeCascade(const char* newfilename, bool simulation, double newmass, const
    char* sourcefilename = "cascade_tmp");
void printfullcascade(Fcascade* oCascade, const char* filename);
#endif
```

A.32 simulating cascades

simCascade.cpp :

```
#ifndef SIM_CASCADE_CPP
#define SIM_CASCADE_CPP

#include <vector>
#include <string>

#include "TROOT.h"
#include "TSystem.h"
#include "TApplication.h"

#include "TDatabasePDG.h"
#include "TParticlePDG.h"
#include "TString.h"
#include "TXMLEngine.h"
#include "TCanvas.h"
#include "TGenPhaseSpace.h"
#include "TLorentzVector.h"
#include "TDecayChannel.h"
#include "TH1F.h"
#include "TStyle.h"
#include "TError.h"
#include "TLine.h"

#include "partial_reconstruction.hpp"
#include "filenames.hpp"
#include "globals.hpp"
#include "beta_distribution.hpp"
#include "myparticlestructs.hpp"
#include "visualizeCascade.hpp"
#include "CascFile.hpp"

#include "simCascade.hpp"

struct Fsimcascstep {
  int ndaughters;
  double* masses;
  int id_dd1;
  int id_dd2;
  Fsimcascstep() : ndaughters (0) {}
};

void casc_hist(TLorentzVector iLV, Fsimcascstep** allsteps, TH1F* cascHist, int nsteps);
void Fill_casc_hist_p1(TLorentzVector* LVp1, TLorentzVector* LVp2, Fsimcascstep**
    cascsteps, int istep, TH1F* cascHist, int ndecay);
void Fill_casc_hist_p2(TLorentzVector* LVp1, TLorentzVector* LVp2, Fsimcascstep**
    cascsteps, int istep, TH1F* cascHist, int ndecay);
void Fill_casc_hist(TLorentzVector* LVp1, TLorentzVector* LVp2, TH1F* cascHist);
Fsimcascstep* makesimcascstep(Fcascadestep* step, TDatabasePDG* pdg, int pdgdd2);
Fsimcascstep* makesimcascstep(Fcascadestep* step, TDatabasePDG* pdg);

void simCascade(Fcascade* oCascade, const char* filename) {
  gStyle->SetOptStat(kFALSE);
  TCanvas* mycanvas = new TCanvas("mycanvas");
  TH1F* cascHist;
  TLorentzVector LVp1, LVp2, LVmother;
  TDatabasePDG* pdg;
  Fsimcascstep* common_step;
  Fsimcascstep* p1steps;
  Fsimcascstep* p2steps;
  Fsimcascstep** allsteps;
  Fcascadestep* step;
  std::vector<Fcascadestep*> sCasc;
  int count, p1count, p2count, nsteps, mempdg;
  double mmass, ymax;
  pdg = oCascade->pdgptr;
```

```
  mmass = oCascade->iParticles[0]->Mass();
  cascHist = new TH1F("CascadeBG", "CascadeBG", 50, 0, mmass + 2);
  p1count = oCascade->_p1steps - 1;
  p2count = oCascade->_p2steps - 1;
  count = oCascade->_p2steps;
  nsteps = p1count + p2count + 1;
  allsteps = (Fsimcascstep**)malloc(sizeof(Fsimcascstep*) * 3);
  p1steps = (Fsimcascstep*)malloc(sizeof(Fsimcascstep) * p1count);
  p2steps = (Fsimcascstep*)malloc(sizeof(Fsimcascstep) * p2count);
  sCasc = oCascade->cascade;
  while (!sCasc.empty())
  {
    count--;
    step = sCasc.back();
    if (count < 0)
    {
      p1steps[(-count) - 1] = *makesimcascstep(step, pdg);
    }
    if (count == 0)
    {
      mempdg = step->decaying_daughter->PdgCode();
      sCasc.pop_back();
      step = sCasc.back();
      LVmother = { 0 , 0, 0, step->mother->Mass() };
      common_step = makesimcascstep(step, pdg, mempdg);
    }
    if (count > 0)
    {
      p2steps[count - 1] = *makesimcascstep(step, pdg);
    }
    sCasc.pop_back();
  }
  allsteps[0] = common_step;
  allsteps[1] = p1steps;
  allsteps[2] = p2steps;
  casc_hist(LVmother, allsteps, cascHist, nsteps);
  cascHist->GetXaxis()->SetRangeUser((mmass - globals::deltaW), (mmass + globals::deltaW)
      );
  ymax = 1.1 * cascHist->GetMaximum();
  cascHist->GetXaxis()->SetTitle("Energy (GeV)");
  cascHist->GetYaxis()->SetRangeUser(0, ymax);
  cascHist->GetYaxis()->SetLabelSize(0);
  cascHist->GetYaxis()->SetTitle("Arbitrary Units");
  cascHist->Draw();
  TLine* line = new TLine(oCascade->predictmass, 0, oCascade->predictmass, ymax);
  line->Draw();
  mycanvas->Print(filename, "pdf");
  delete mycanvas;
  delete cascHist;
}

void casc_hist(TLorentzVector iLV, Fsimcascstep** allsteps, TH1F* cascHist, int nsteps) {
  TGenPhaseSpace t_event;
  int idecay, ndecay, nevents;
  TLorentzVector* LVp1, *LVp2;
  TLorentzVector LVp1b, LVp2b;
  nevents = globals::Nevents;
  ndecay = std::floor(pow(nevents, (1.00 / nsteps)));
  t_event.SetDecay(iLV, allsteps[0]->ndaughters, allsteps[0]->masses);
  for (idecay = 0; idecay < ndecay; idecay++)
  {
    t_event.Generate();
    LVp1 = t_event.GetDecay(allsteps[0]->id_dd1);
    LVp2 = t_event.GetDecay(allsteps[0]->id_dd2);
    Fill_casc_hist_p1(LVp1, LVp2, allsteps, 0, cascHist, ndecay);
  }
}
```

```
void Fill_casc_hist_p1(TLorentzVector* LVp1, TLorentzVector* LVp2, Fsimcascstep**
    cascsteps, int istep, TH1F* cascHist, int ndecay) {
  if (cascsteps[1][istep].ndaughters < 0)
  {
    Fill_casc_hist_p2(LVp1, LVp2, cascsteps, 0, cascHist, ndecay);
    return;
  }
  TGenPhaseSpace p1_event;
  int idecay;
  Fsimcascstep p1_step;
  TLorentzVector* upLVp1;
  p1_step = cascsteps[1][istep];
  p1_event.SetDecay(*LVp1, p1_step.ndaughters, p1_step.masses);
  for (idecay = 0; idecay < ndecay; idecay++)
  {
    p1_event.Generate();
    upLVp1 = p1_event.GetDecay(p1_step.id_dd1);
    Fill_casc_hist_p1(upLVp1, LVp2, cascsteps, istep + 1, cascHist, ndecay);
  }
}

void Fill_casc_hist_p2(TLorentzVector* LVp1, TLorentzVector* LVp2, Fsimcascstep**
    cascsteps, int istep, TH1F* cascHist, int ndecay) {
  if (cascsteps[2][istep].ndaughters < 0)
  {
    Fill_casc_hist(LVp1, LVp2, cascHist);
    return;
  }
  TGenPhaseSpace p2_event;
  int idecay;
  Fsimcascstep p2_step;
  TLorentzVector* upLVp2;
  p2_step = cascsteps[2][istep];
  p2_event.SetDecay(*LVp2, p2_step.ndaughters, p2_step.masses);
  for (idecay = 0; idecay < ndecay; idecay++)
  {
    p2_event.Generate();
    upLVp2 = p2_event.GetDecay(p2_step.id_dd1);
    Fill_casc_hist_p2(LVp1, upLVp2, cascsteps, istep + 1, cascHist, ndecay);
  }
}

void Fill_casc_hist(TLorentzVector* LVp1, TLorentzVector* LVp2, TH1F* cascHist) {
  double betaz, x1, x2, x, y1, y2, y, z1, z2, z, inv_mass_p1, inv_mass_p2, inv_mass,
      P_Recon_pow2, Mass_Recon;
  TLorentzVector LVp1b, LVp2b;
  betaz = beta_distribution();
  //Boosting
  LVp1b = *LVp1;
  LVp2b = *LVp2;
  LVp1b.Boost(0, 0, betaz);
  LVp2b.Boost(0, 0, betaz);
  //Prop LV
  x1 = LVp1b[0];
  y1 = LVp1b[1];
  z1 = LVp1b[2];
  inv_mass_p1 = LVp1b[3];
  x2 = LVp2b[0];
  y2 = LVp2b[1];
  z2 = LVp2b[2];
  inv_mass_p2 = LVp2b[3];
  // Detection
  //Detection
  //inv_mass_p1 = detect(LV_p1, Massp1);
  //inv_mass_p2 = detect(LV_p2, Massp2);
  //combine
  x = x1 + x2;
  y = y1 + y2;
  z = z1 + z2;
```

```
  inv_mass = inv_mass_p1 + inv_mass_p2;
  // Reconstruction:
  P_Recon_pow2 = pow(x, 2) + pow(y, 2) + pow(z, 2);
  Mass_Recon = sqrt(pow(inv_mass, 2) - (P_Recon_pow2));
  //Hist
  cascHist->Fill(Mass_Recon, 1);
}

Fsimcascstep* makesimcascstep(Fcascadestep* step, TDatabasePDG* pdg) {
  return makesimcascstep(step, pdg, 0);
}

Fsimcascstep* makesimcascstep(Fcascadestep* step, TDatabasePDG* pdg, int pdgdd2) {
  TDecayChannel* tch;
  TParticlePDG* t_particle;
  Fsimcascstep* simstep = new Fsimcascstep();
  double* Dmasses;
  int nd, j, pdgProduct;
  tch = step->Channel;
  nd = tch->NDaughters();
  simstep->ndaughters = nd;
  Dmasses = (double*)malloc(sizeof(double) * nd);
  for (j = 0; j < nd; j++)
  {
    pdgProduct = tch->DaughterPdgCode(j);
    if (pdgProduct == step->decaying_daughter->PdgCode())
    {
      simstep->id_dd1 = j;
    }
    if (pdgProduct == pdgdd2)
    {
      simstep->id_dd2 = j;
    }
    t_particle = pdg->GetParticle(pdgProduct);
    Dmasses[j] = t_particle->Mass();
  }
  simstep->masses = Dmasses;
  return simstep;
}

#endif
```

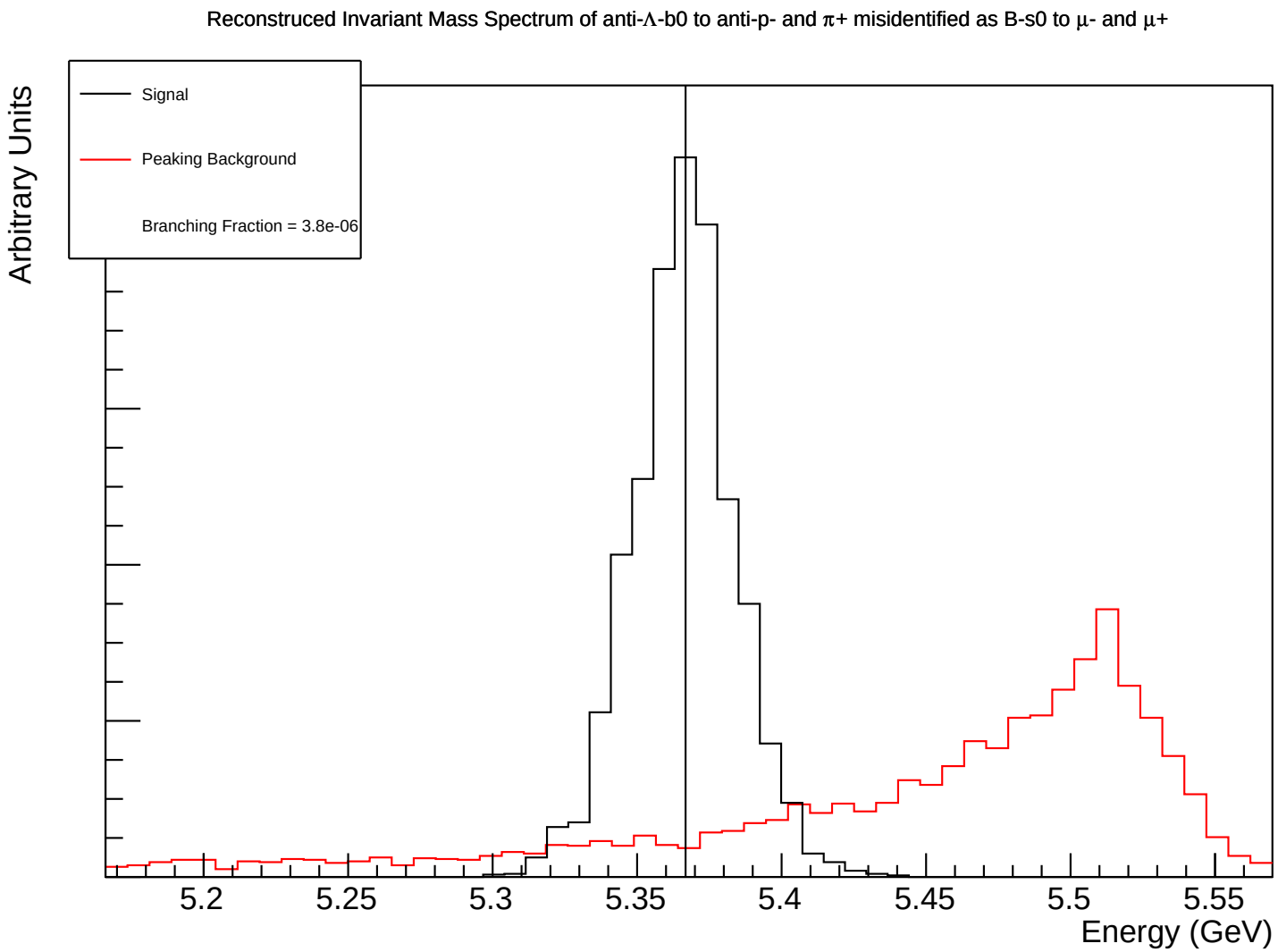
simCascade.hpp :

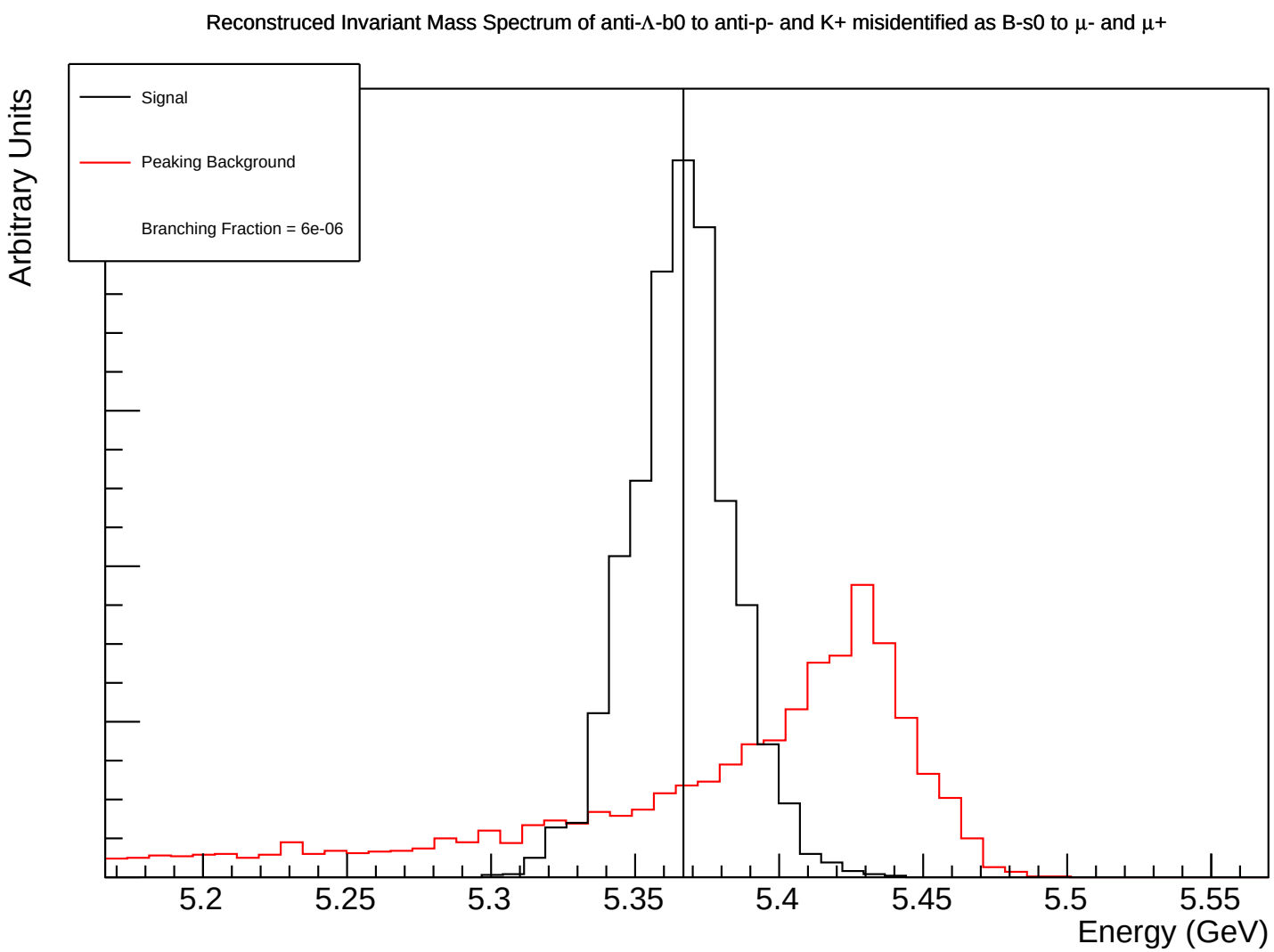
```
#ifndef SIM_CASCADE_HPP
#define SIM_CASCADE_HPP
void simCascade(Fcascade* oCascade, const char* filename);
#endif
```

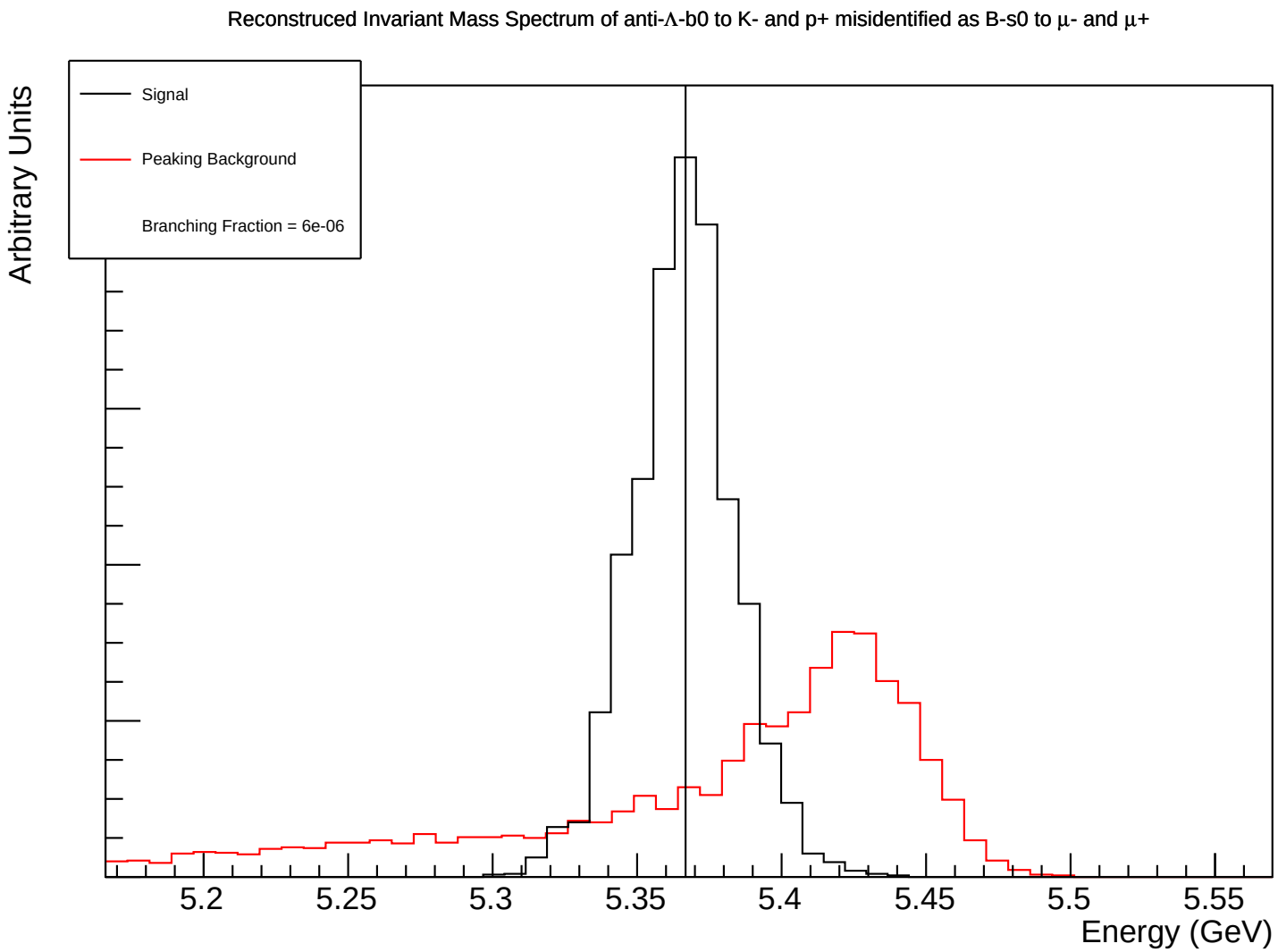
Appendix B

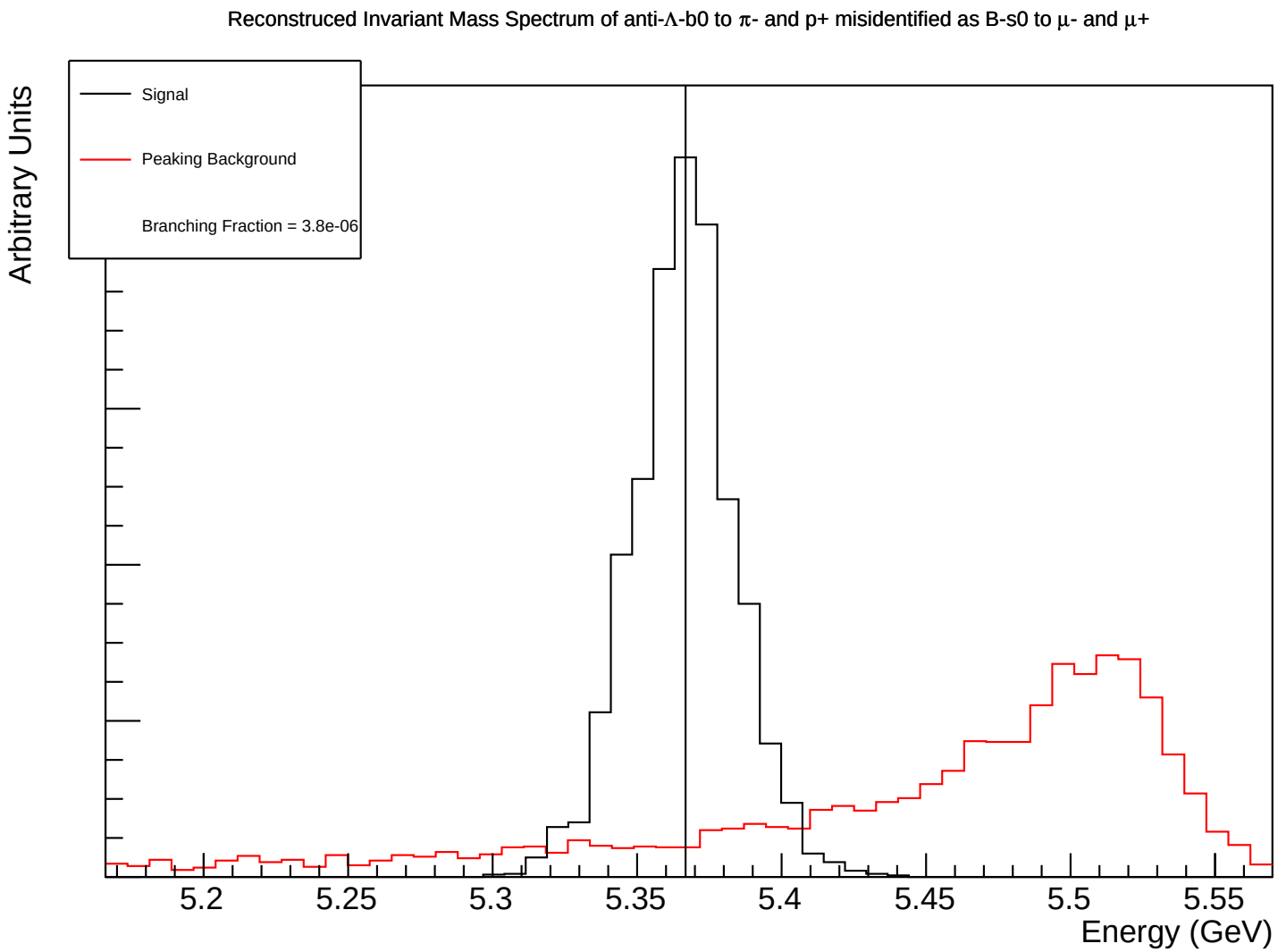
Peaking backgrounds for B_s^0 to $\mu\mu$

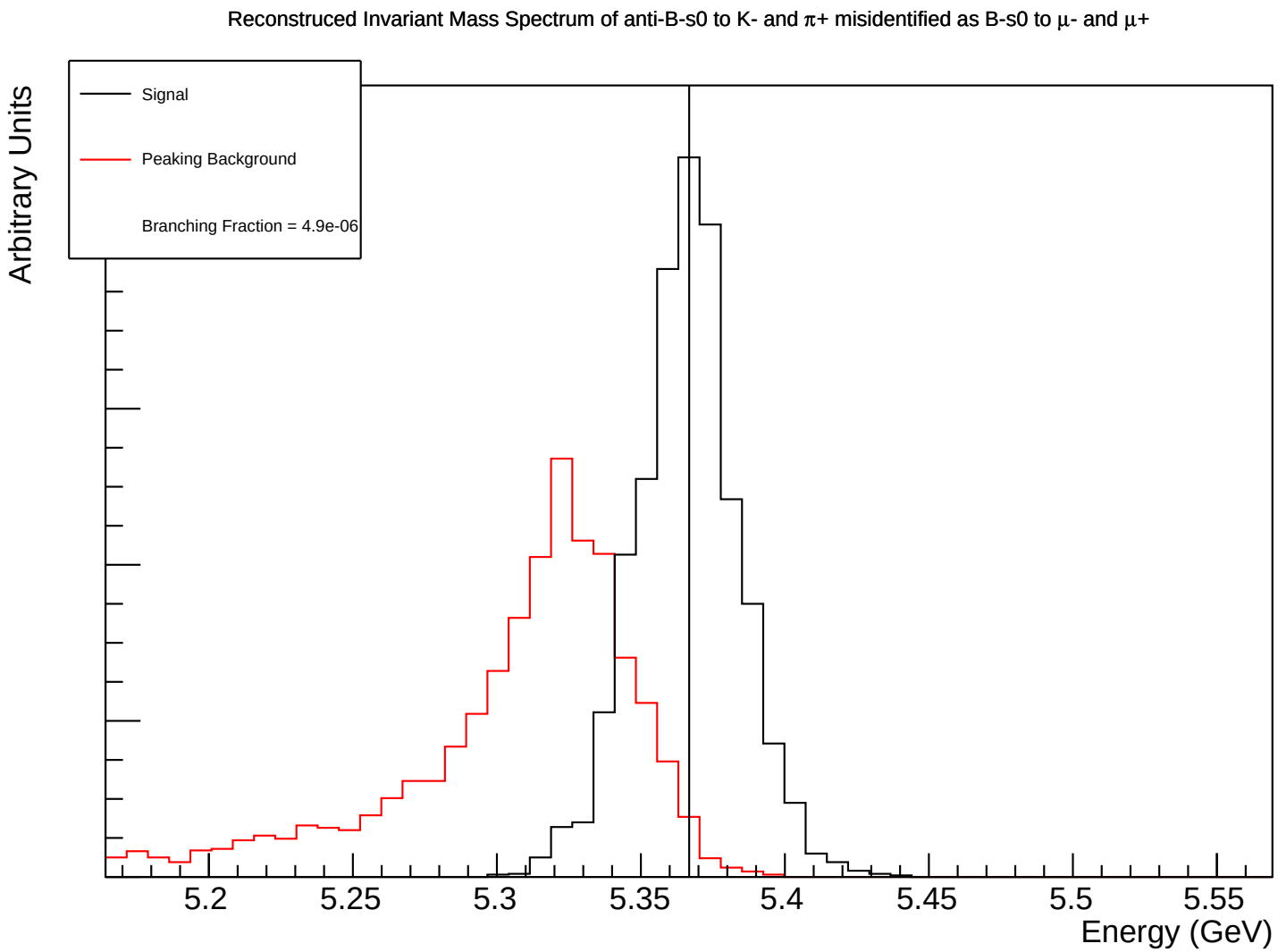
The result of running the function `MisID("B_s0", "mu+", "mu-", "Bmumu")` with the following global parameters:
detector tau = 10^{-9}
sigma = 0.001
alpha Brem = 0.005
deltaW = 0.2
minBR = 10^{-20}
Nevents = 5000

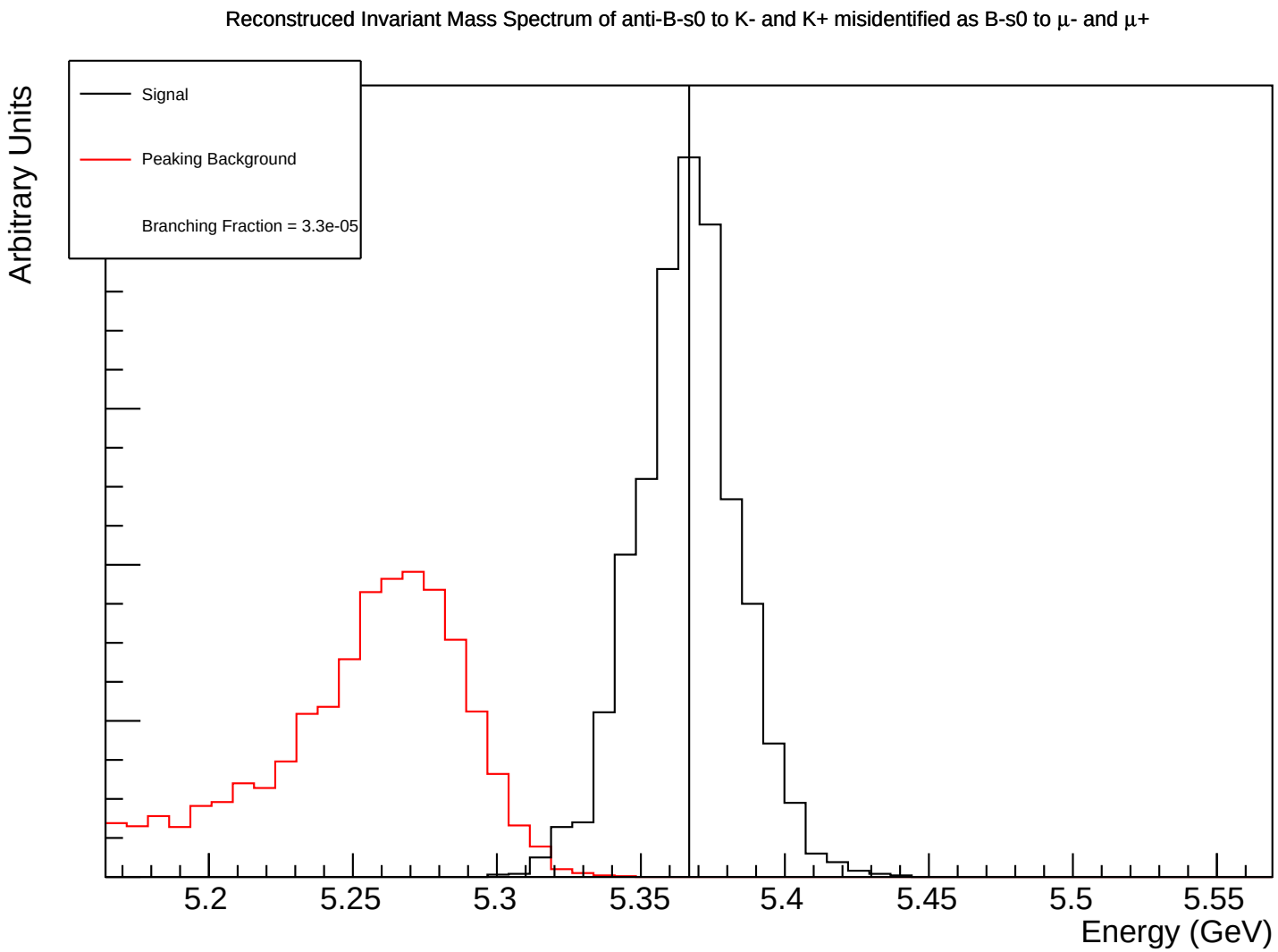

Reconstruced Invariant Mass Spectrum of anti-Λ-b0 to anti-p- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.8e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

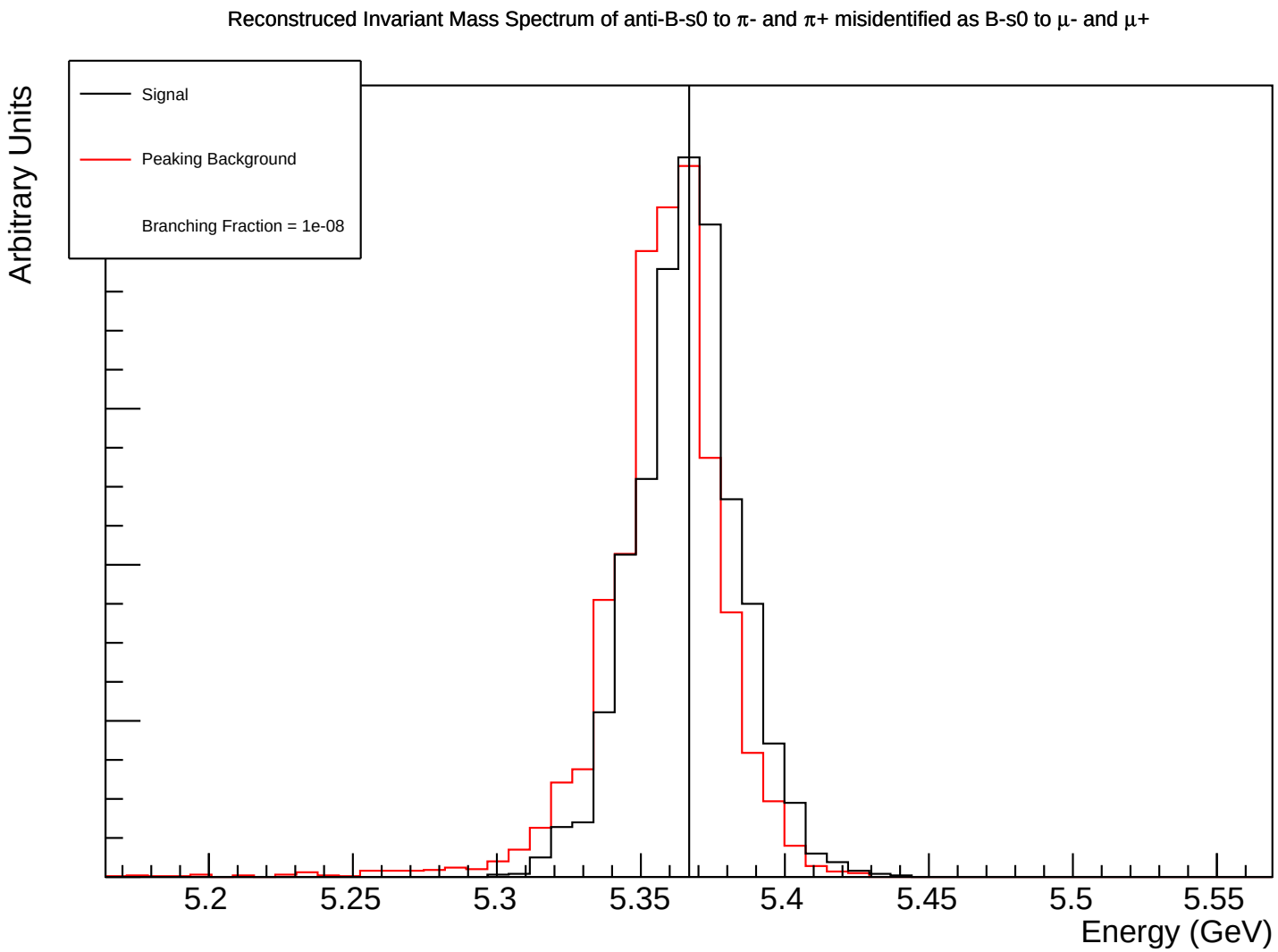

Reconstruced Invariant Mass Spectrum of anti-Λ-b0 to anti-p- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 6e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

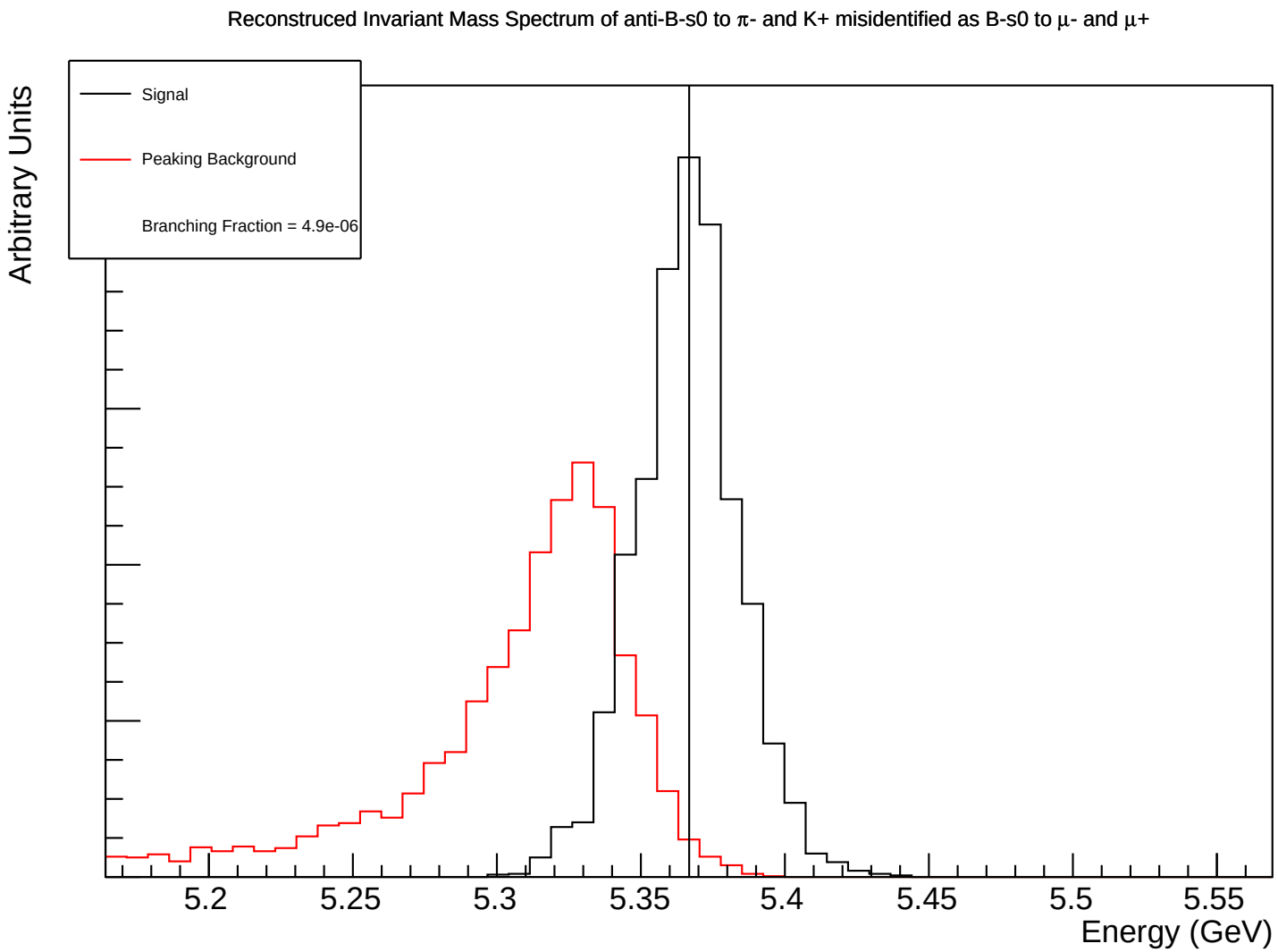

Reconstruced Invariant Mass Spectrum of anti-Λ-b0 to K- and p+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 6e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

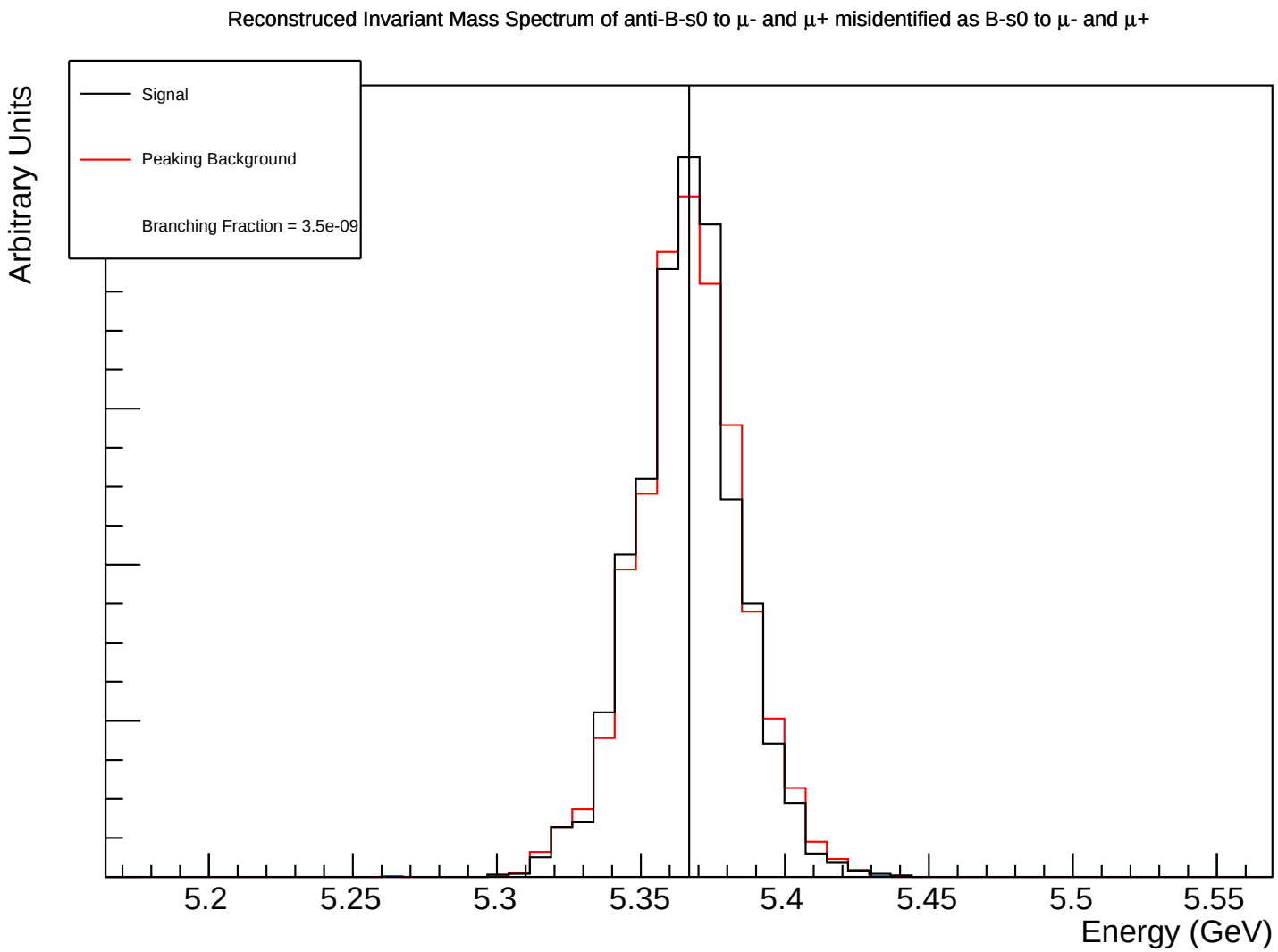

Reconstruced Invariant Mass Spectrum of anti-Λ-b0 to π- and p+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.8e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

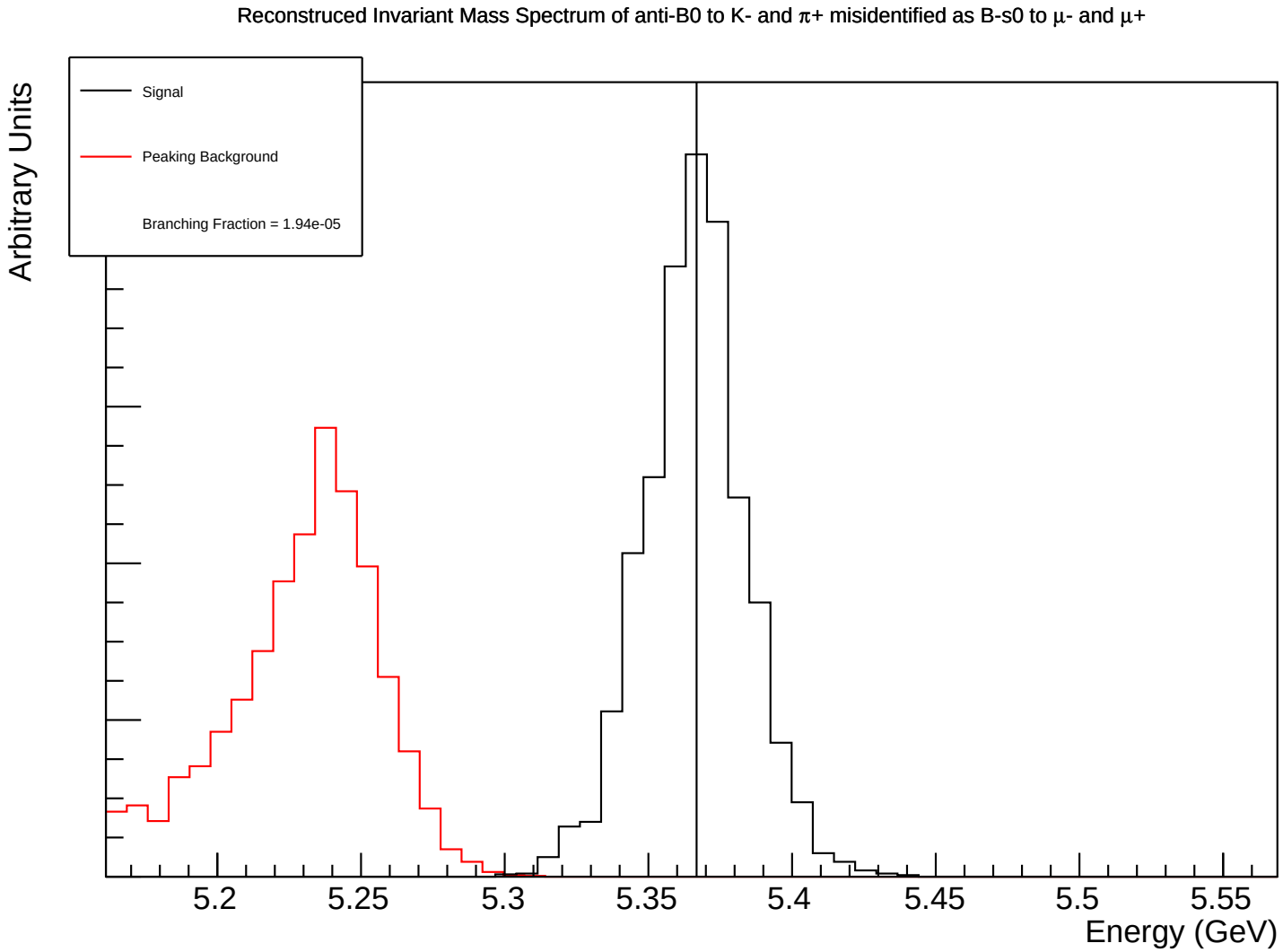

Reconstruced Invariant Mass Spectrum of anti-B-s0 to K- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4.9e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

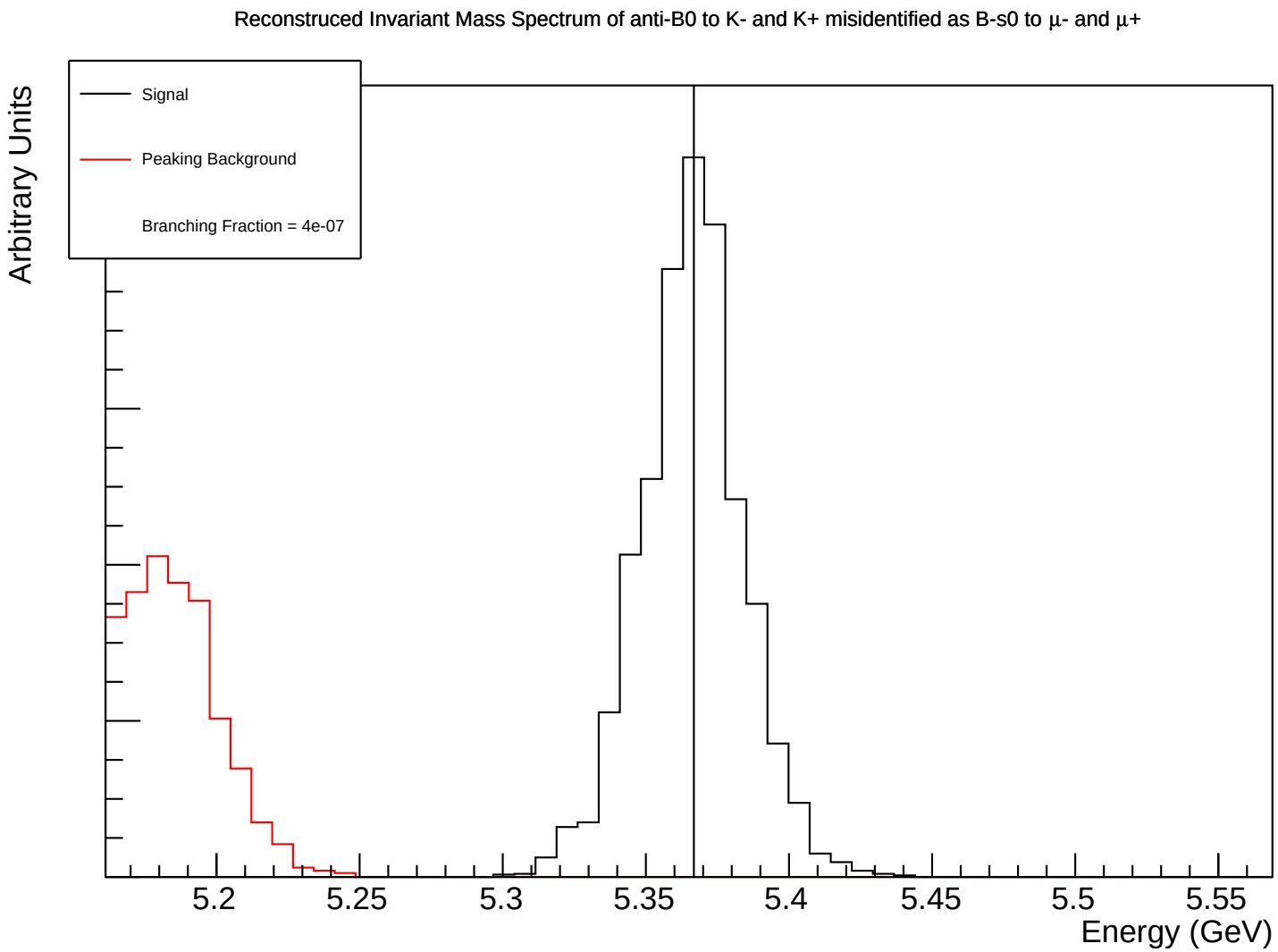

Reconstruced Invariant Mass Spectrum of anti-B-s0 to K- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.3e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

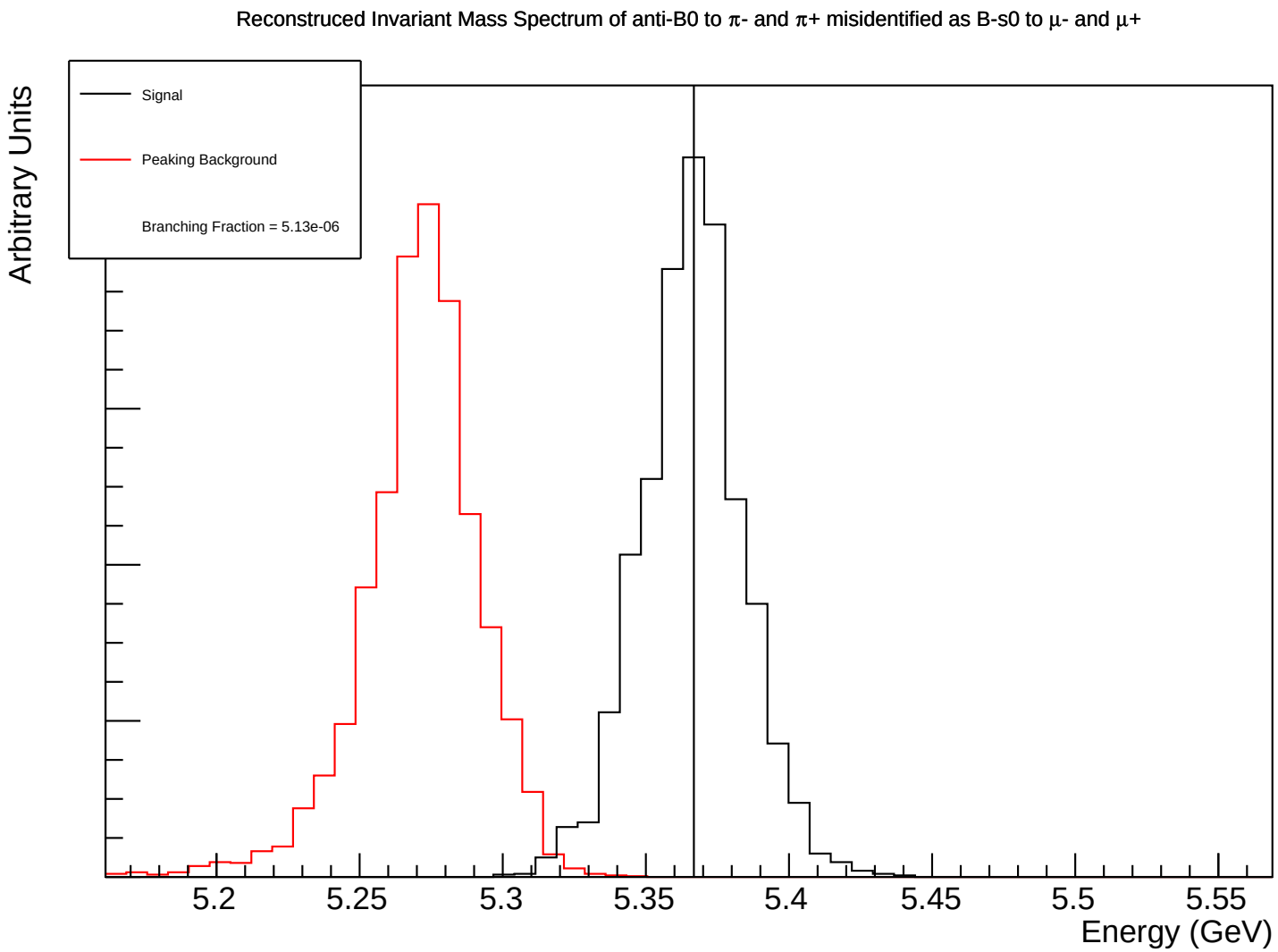

Reconstruced Invariant Mass Spectrum of anti-B-s0 to π- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1e-08
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

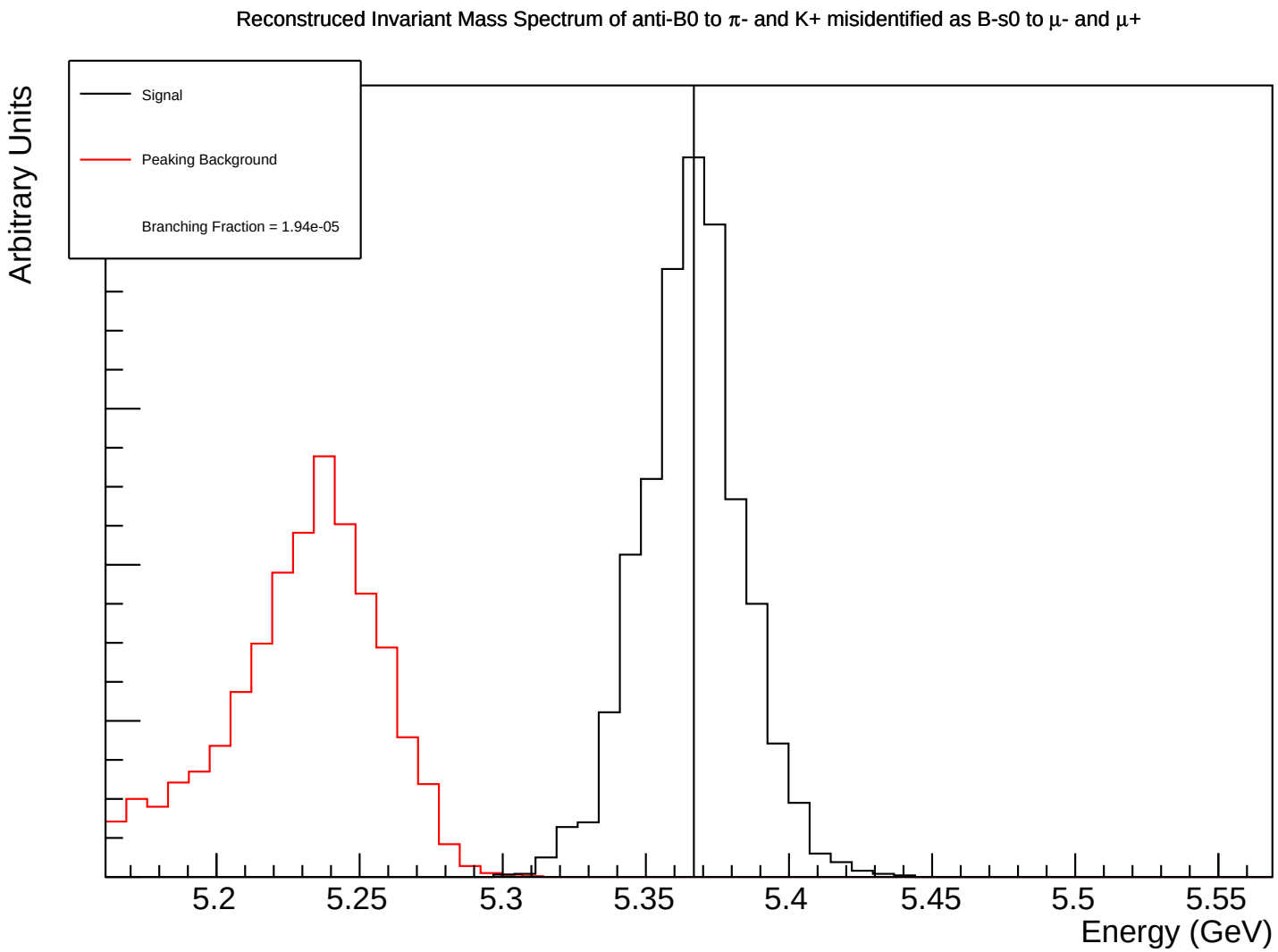

Reconstruced Invariant Mass Spectrum of anti-B-s0 to π- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4.9e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

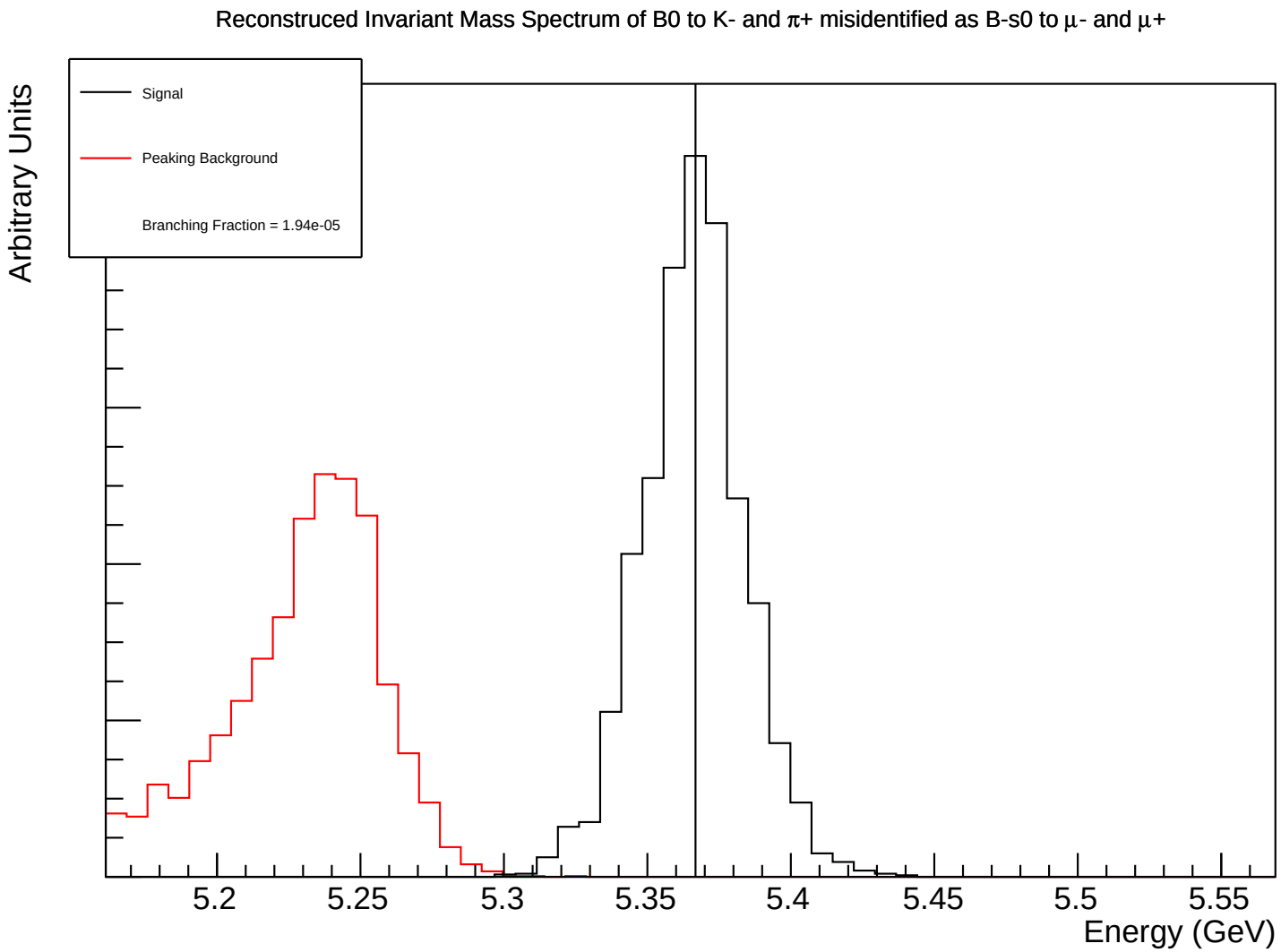

Reconstruced Invariant Mass Spectrum of anti-B-s0 to μ- and μ+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.5e-09
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

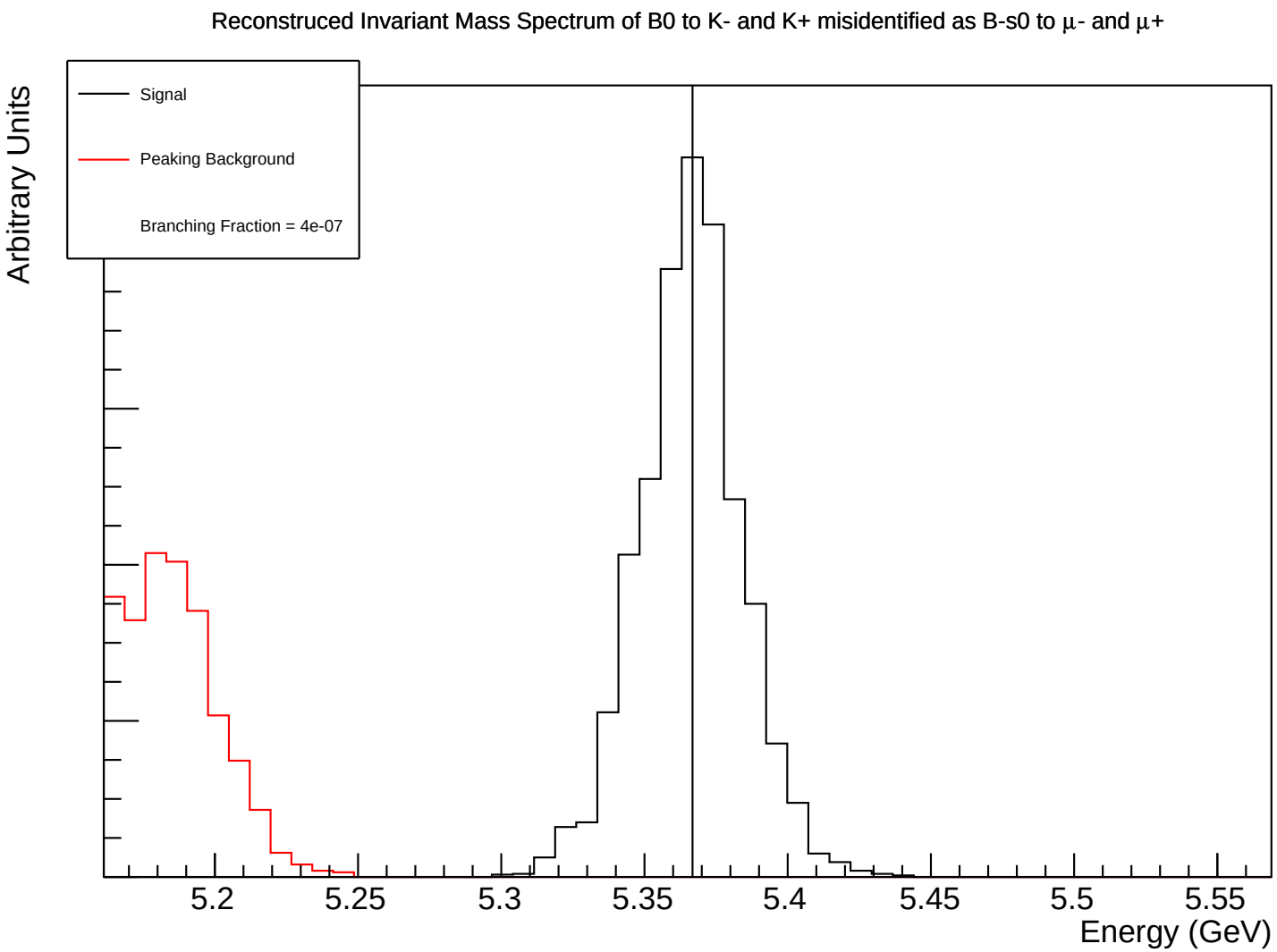

Reconstruced Invariant Mass Spectrum of anti-B0 to K- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1.94e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

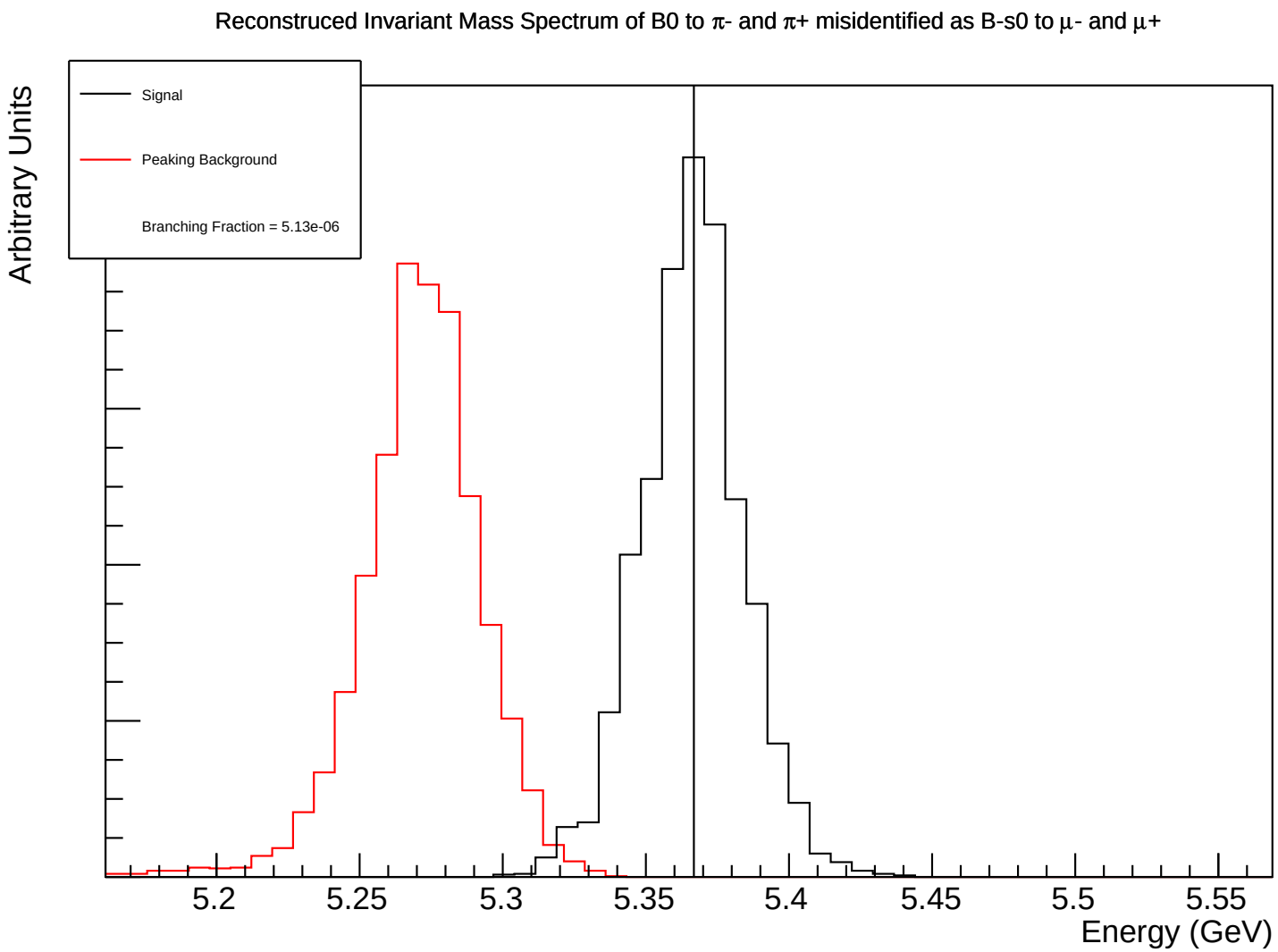

Reconstruced Invariant Mass Spectrum of anti-B0 to K- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4e-07
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

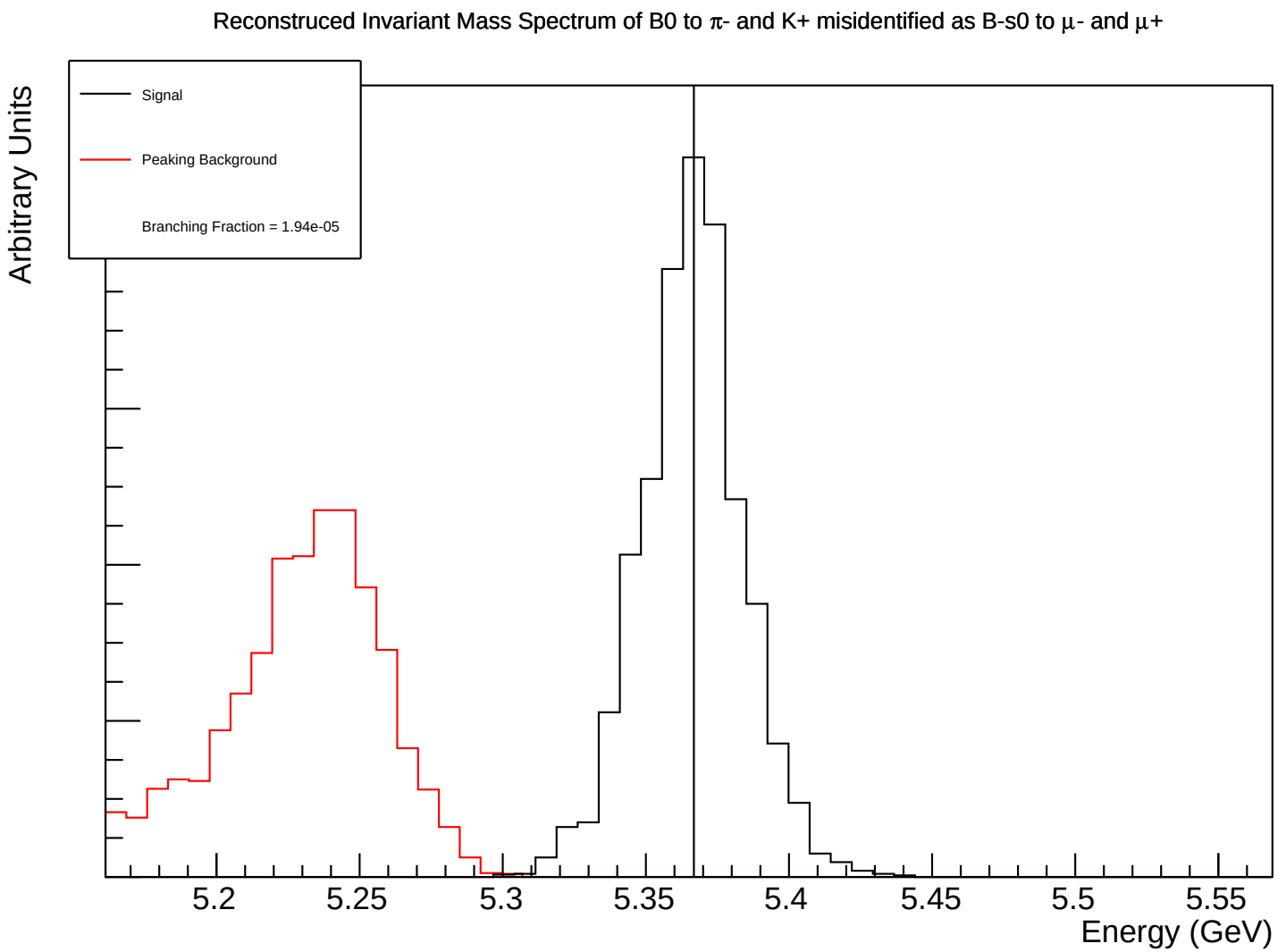

Reconstruced Invariant Mass Spectrum of anti-B0 to π- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 5.13e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

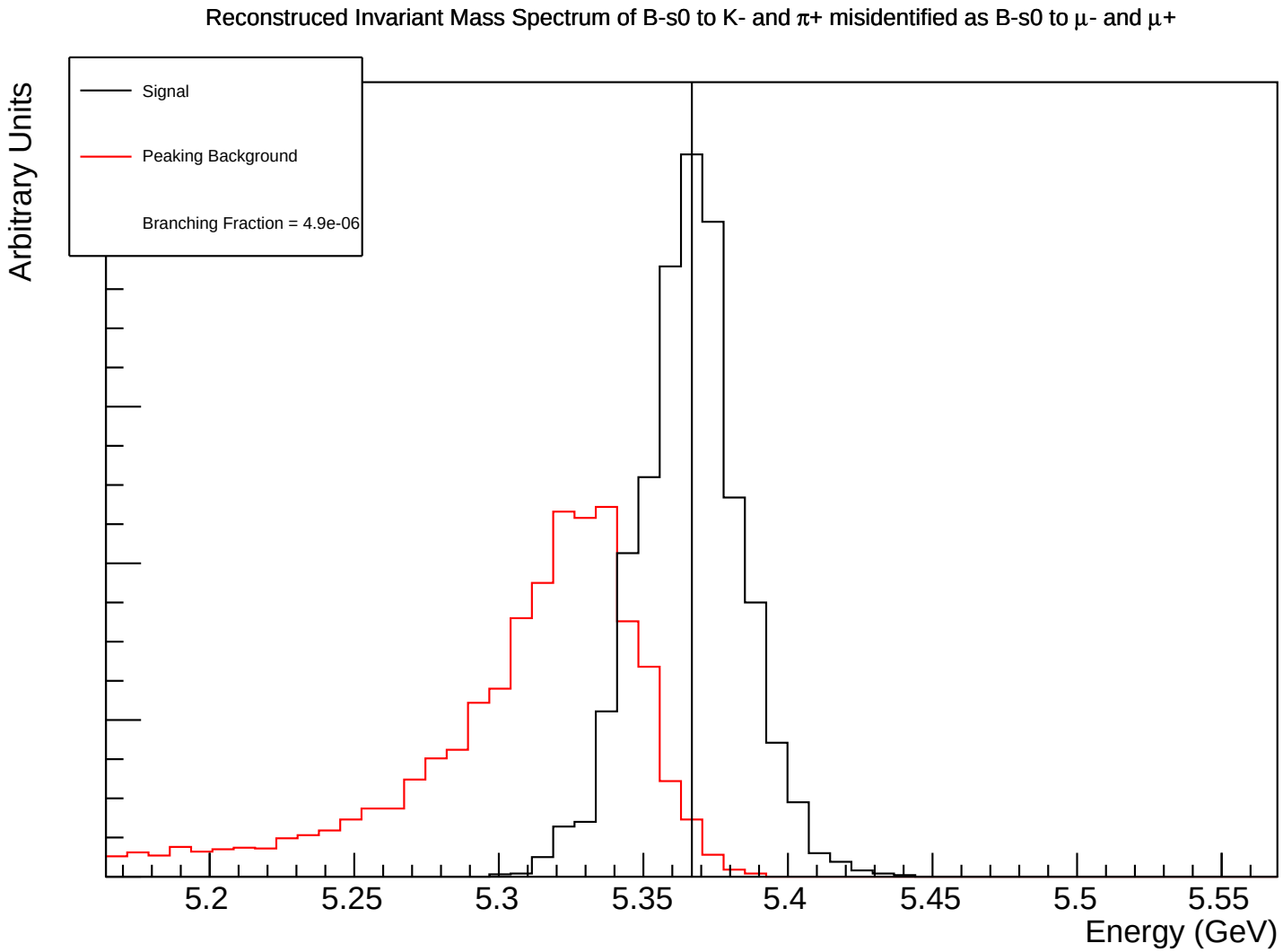

Reconstruced Invariant Mass Spectrum of anti-B0 to π- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1.94e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

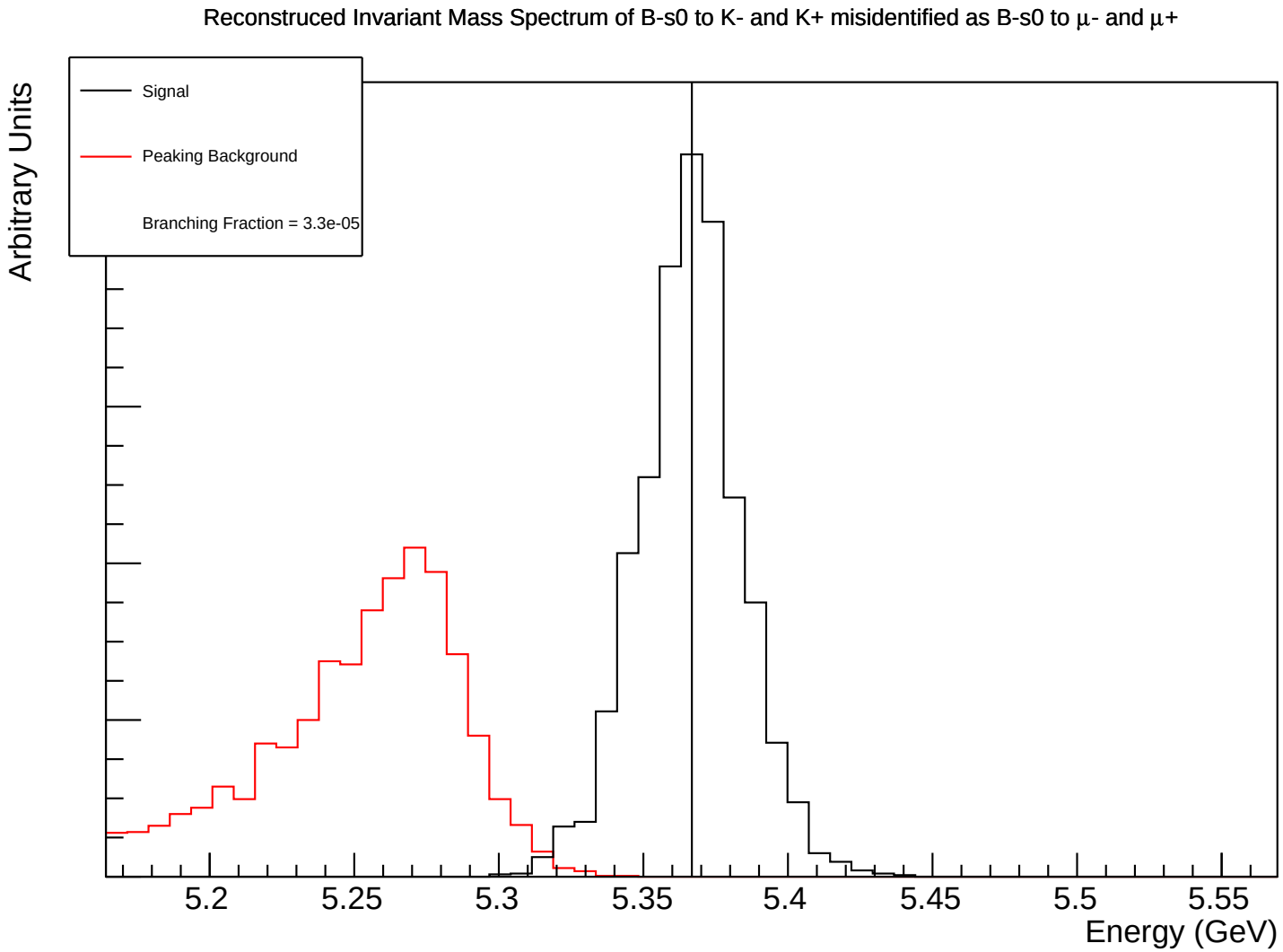

Reconstruced Invariant Mass Spectrum of B0 to K- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1.94e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

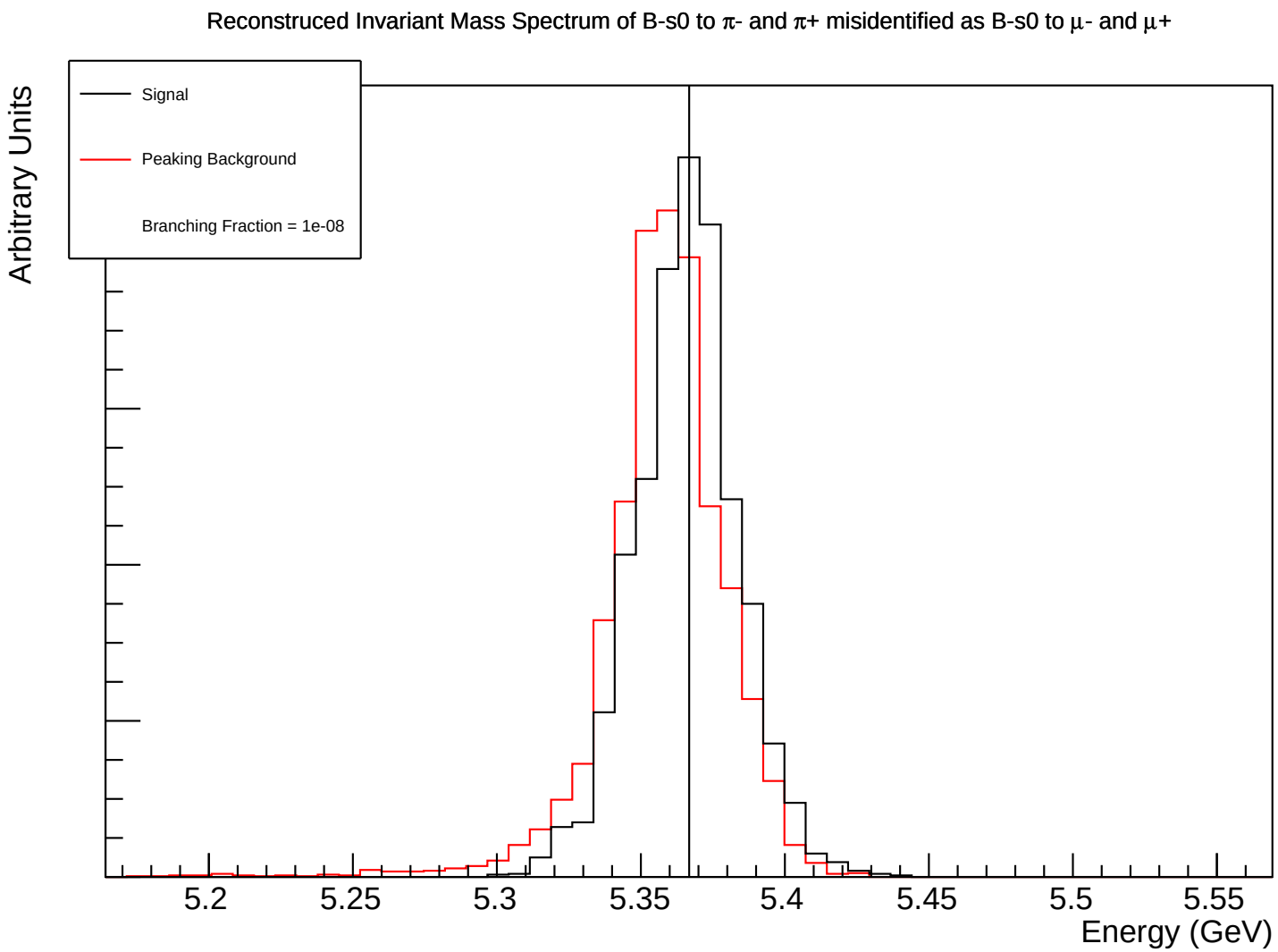

Reconstruced Invariant Mass Spectrum of B0 to K- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4e-07
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

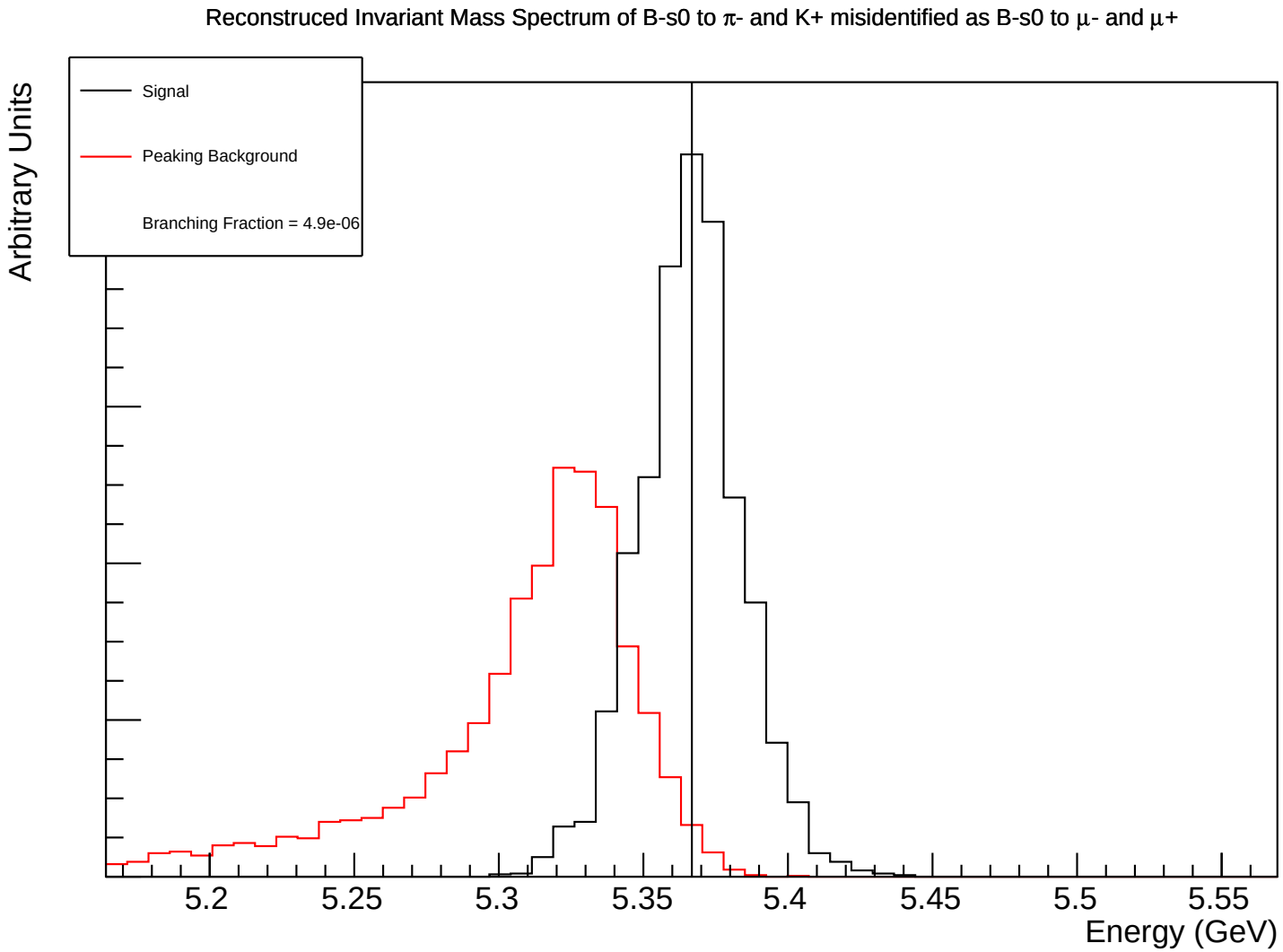

Reconstruced Invariant Mass Spectrum of B0 to π- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 5.13e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

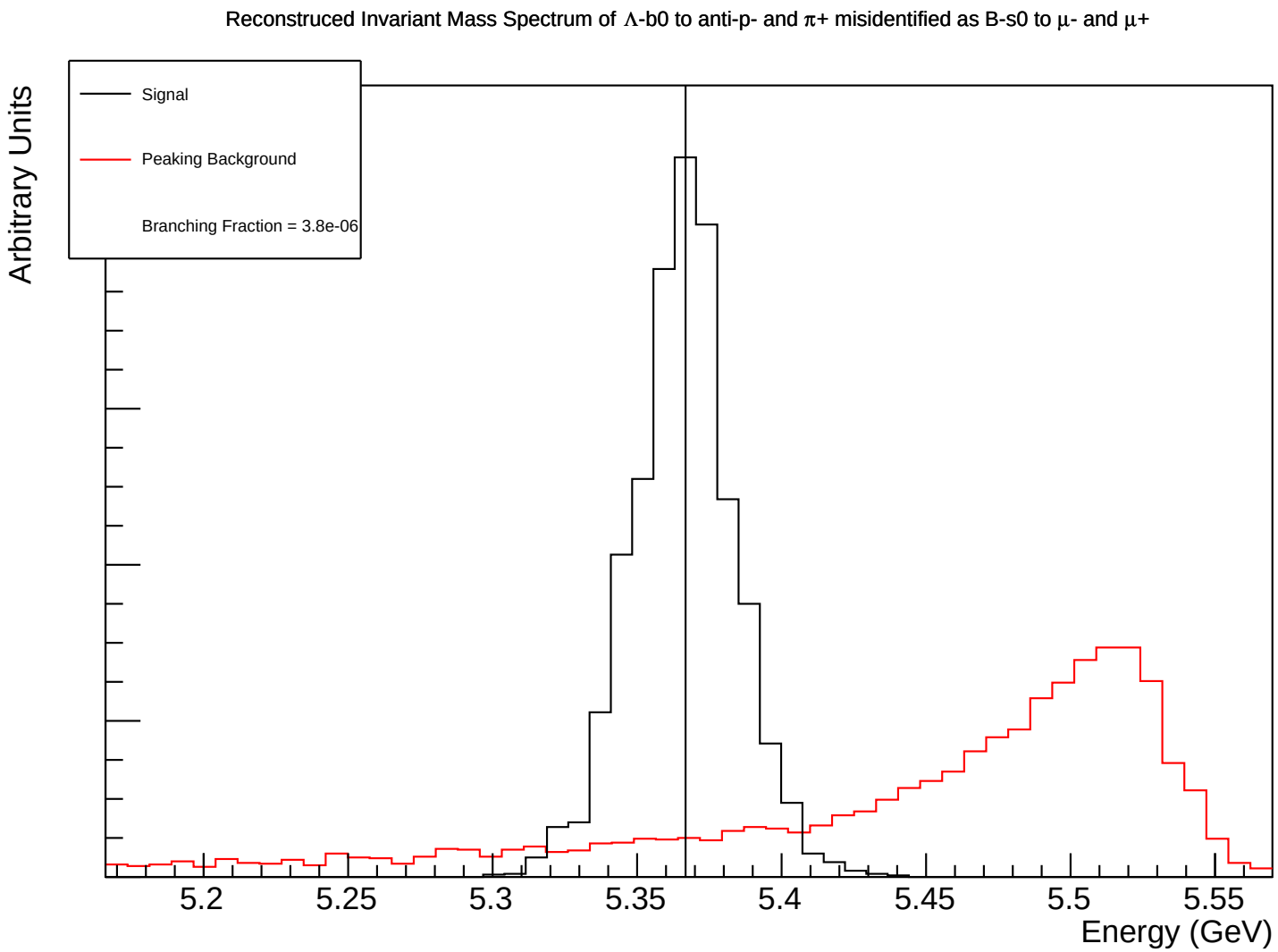

Reconstruced Invariant Mass Spectrum of B0 to π- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1.94e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

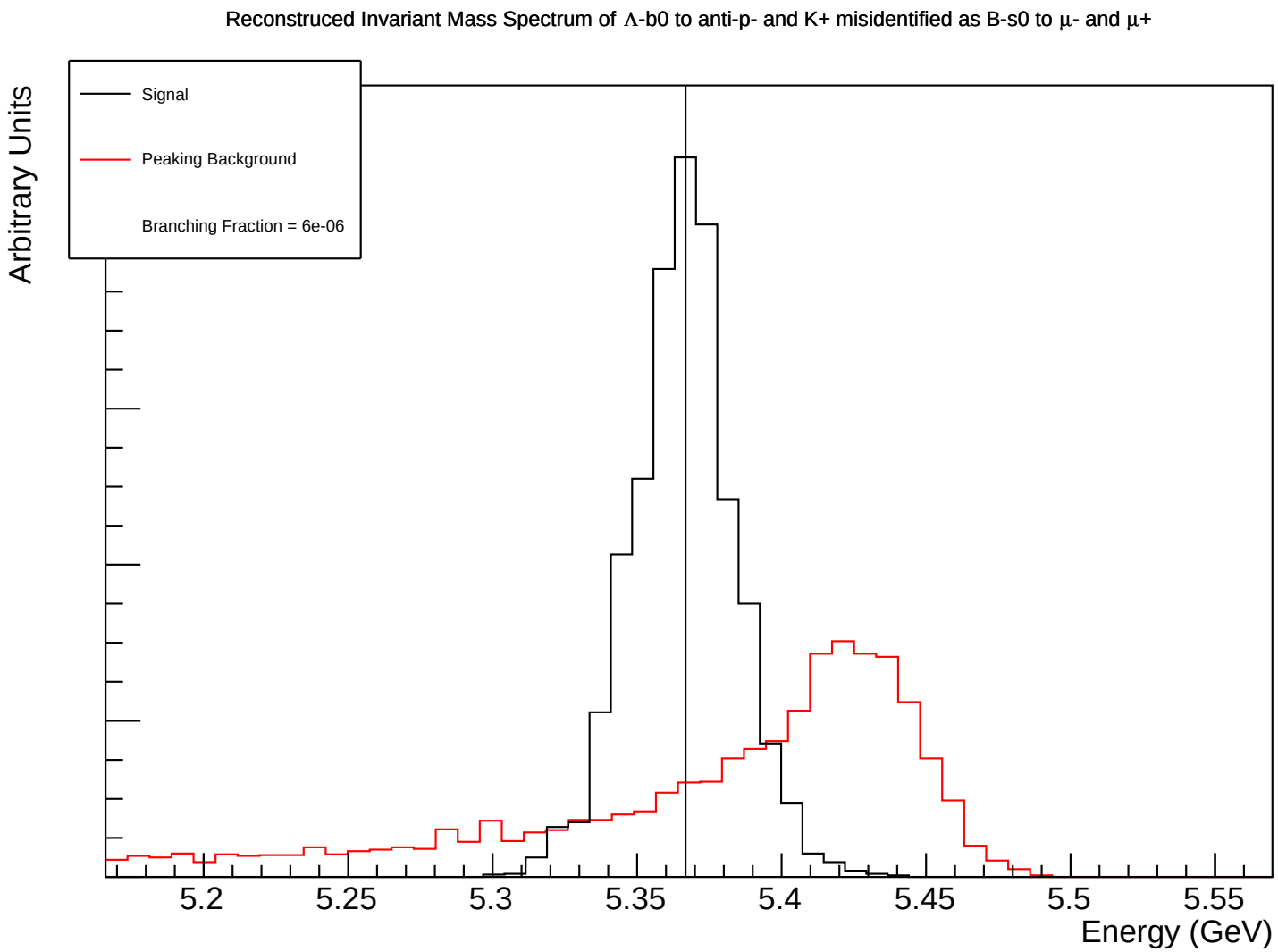

Reconstruced Invariant Mass Spectrum of B-s0 to K- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4.9e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

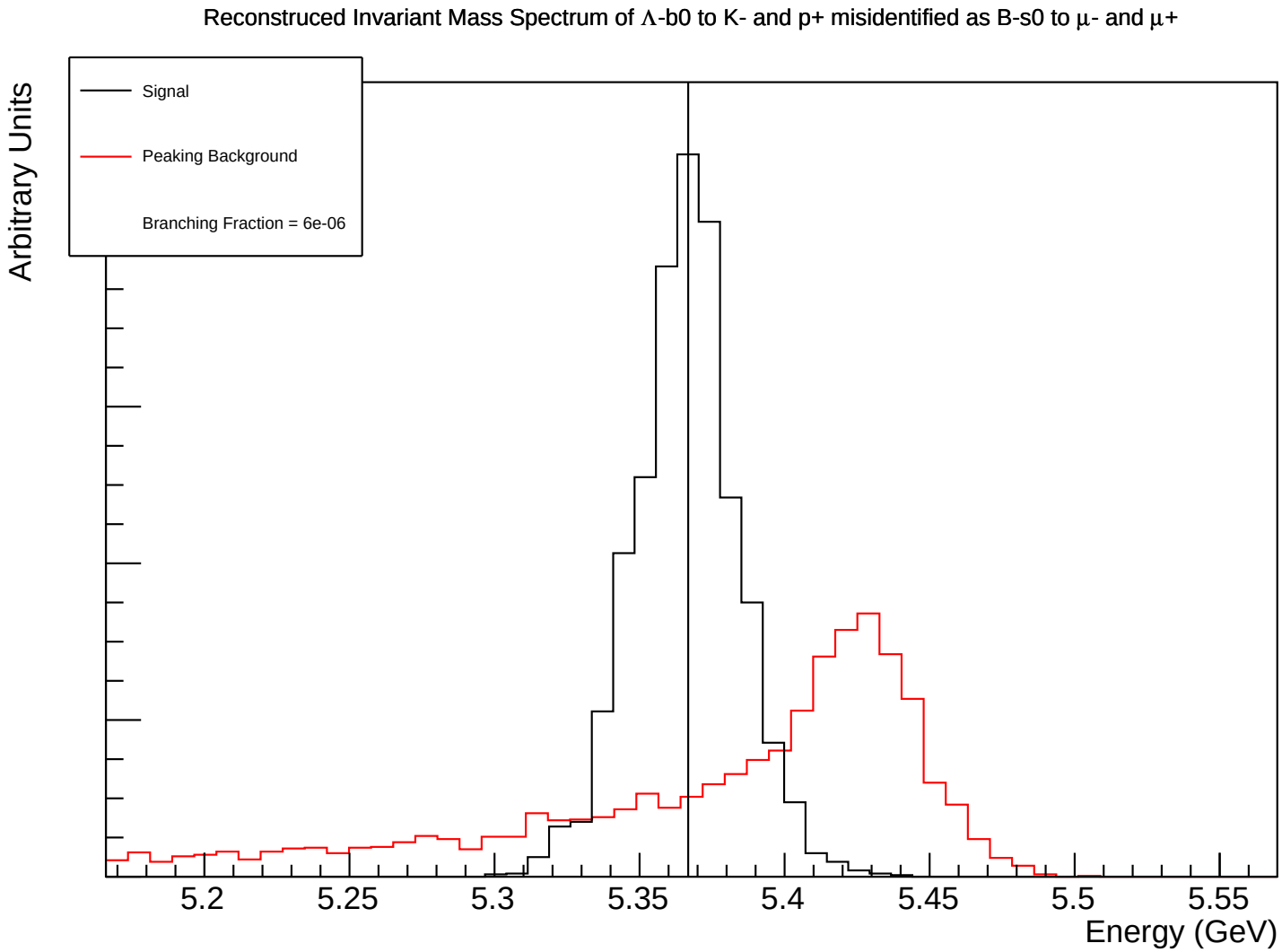

Reconstruced Invariant Mass Spectrum of B-s0 to K- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.3e-05
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)

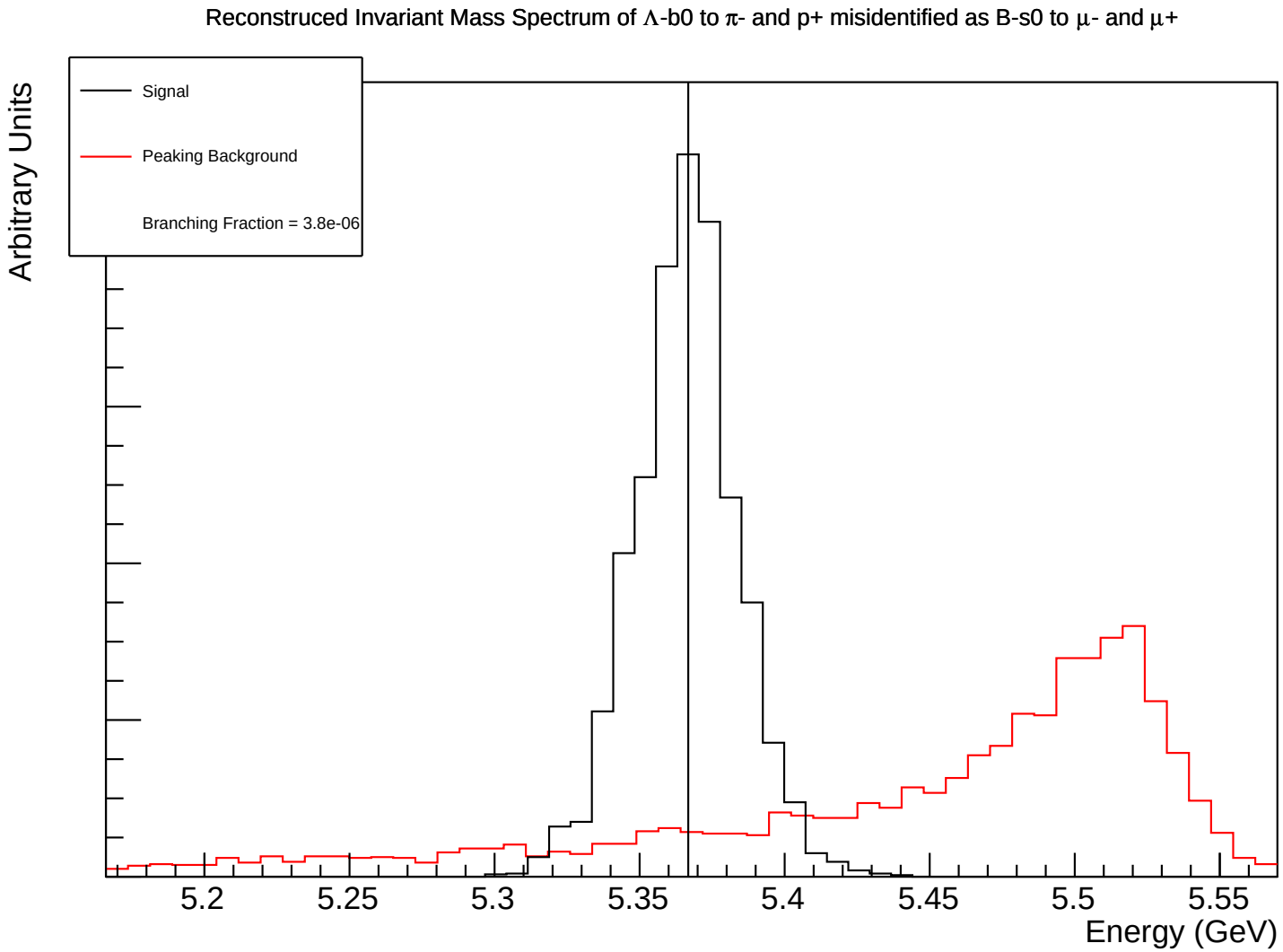

Reconstruced Invariant Mass Spectrum of B-s0 to π- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 1e-08
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)


Reconstruced Invariant Mass Spectrum of B-s0 to π- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 4.9e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)


Reconstruced Invariant Mass Spectrum of Λ-b0 to anti-p- and π+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.8e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)


Reconstruced Invariant Mass Spectrum of Λ-b0 to anti-p- and K+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 6e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)


Reconstruced Invariant Mass Spectrum of Λ-b0 to K- and p+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 6e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)


Reconstruced Invariant Mass Spectrum of Λ-b0 to π- and p+ misidentified as B-s0 to μ- and μ+
Signal
Peaking Background
Branching Fraction = 3.8e-06
Arbitrary Units
5.2
5.25
5.3
5.35
5.4
5.45
5.5
5.55
Energy (GeV)