MAT-VET-F-21031
Examensarbete 15 hp
Juni 2021

Maximum Likelihood Estimation of Hyperon Parameters in Python

Facilitating Novel Studies of Fundamental Symmetries with Modern Software Tools

Benjamin Verbeek

Abstract

Maximum Likelihood Estimation of Hyperon Parameters in Python

Benjamin Verbeek

In this project, an algorithm has been implemented in Python to estimate the parameters describing the production and decay of a spin 1/2 baryon - antibaryon pair. This decay can give clues about a fundamental asymmetry between matter and antimatter. A model-independent formalism developed by the Uppsala hadron physics group and previously implemented in C++, has been shown to be a promising tool in the search for physics beyond the Standard Model (SM) of particle physics. The program developed in this work provides a more user-friendly alternative, and is intended to motivate further use of the formalism through a more maintainable, customizable and readable implementation. The hope is that this will expedite future research in the area of charge parity (CP)-violation and eventually lead to answers to questions such as why the universe consists of matter. A Monte-Carlo integrator is used for normalization and a Python library for function minimization. The program returns an estimation of the physics parameters including error estimation. Tests of statistical properties of the estimator, such as consistency and bias, have been performed. To speed up the implementation, the Just-In-Time compiler Numba has been employed which resulted in a speed increase of a factor 400 compared to plain Python code.

Handledare: Karin Schönning
Ämnesgranskare: Martin Sjödin
Examinator: Martin Sjödin
ISSN: 1401-5757, MAT-VET-F-21031
Tryckt av: Uppsala

Contents

Supervisor: *Karin Schönning*
Co-supervisors: *Michael Papenbrock,*
Patrik Adlarson, Viktor Thorén
Subject reader and examiner: *Martin Sjödin*
ISSN: *1401-5757, MAT-VET-F-21031*

1 Introduction

Understanding the very nature of our existence is the puny undertaking of many fundamental physicists. Microcosmos, i.e. the building blocks of the Universe, follow the laws of quantum mechanics and are thus stochastic in their nature. Therefore, probability theory is a necessary and powerful tool when describing various phenomena around us. The measurable quantities, i.e. the observables, follow certain distributions, and sometimes these distributions can give information about the universe itself. For instance, the universe consists of mainly matter but barely any antimatter. Assuming equal amounts of both existed in the Big Bang, the enrichment of matter is not a trivial and even less an understood process.

Mass-wise, this matter mainly consists of protons and neutrons. Heavy relatives of these abundant particles are hyperons. They are similar to protons in many ways, their main difference being their mass: hyperons are heavier. Though protons are abundant and known since more than a century, they are stable and therefore do not reveal much about themselves. Neutrons on the other hand are unstable when not bound into an atomic nucleus, but their decays always include hard-to-measure neutrinos. In comparison, hyperons often decay into charged particles which are straight-forward to measure.

An example is the decay process of Λ-hyperons and $\overline{\Lambda}$-antihyperons. In these decay processes, the emission angles of the decay products follow a certain distribution. This can be derived using very basic assumptions. Two particular decay parameters, α_Λ and $\alpha_{\overline{\Lambda}}$, quantify the decay pattern and should have equal values but opposite sign if a certain type of symmetry between particles and antiparticles exists in the universe. If different laws of physics apply to matter and antimatter, this could explain the apparent abundance of matter in the universe. This is because a surplus could eventually have built up to produce the matter dominated universe we know. The decay of the Λ hyperons and $\overline{\Lambda}$ antihyperons is one potential place to look for indicators of these asymmetries. The decay process of spin 1/2 hyperons is relatively uncharted territory when looking for this kind of asymmetry.

This project aims to produce a program which can estimate precisely these indicative decay parameters using a maximum log-likelihood fit. The formalism used in the program is applicable to any pair of spin 1/2 hyperons decaying, which means the chances of it helping find something interesting increase. The $\Lambda\overline{\Lambda}$ pair is merely an example, and there exist a multitude of other decays which are yet to be studied which could lead to most interesting discoveries.

In general, when conducting research, *usability*, *flexibility* and *compatibility* of tools used is essential. Even a tool that produces very precise results in a specific case, it will in practice only be accessible by a limited number of people if it requires a very specific expertise to use it. Hence, the chances of the tool leading to discoveries on a broader scale remain small. The aim of this project is to create a tool that is accessible to a larger community. The fit is to be implemented in such a way that it is easy to run, modify and adapt to similar processes and co-exist with current frameworks. The hope is that this will expedite scientific progress in the field of charge-parity (CP) symmetry breaches in hyperon decays.

Recently, the virtue of user-friendly data analysis tools have received increasing recognition. Currently, when scientific papers are published within the field of nuclear physics, it is common to refrain from publishing the data and program used to fit it. A common explanation is that the data requires niche, expert knowledge to be of any interest, knowledge typically only the authors possess. Libraries used have been developed over decades are written by physicists rather than computer scientists, and are written in what are today considered low level languages. While these libraries were a step towards simplification during development, today even more efficient tools exist which allow for an upgrade. It appears to me that in the near future, publishing with data and the fitting program will be the norm. Open access to these tools could lead to easier cross-examination and revolutionary results. In the future, it is possible, if not likely, that also today's paradigm will appear unnecessarily complicated.

1.1 The Standard Model

The Standard Model (SM) in particle physics has been hugely successful in predicting experimental results. It describes the strong, weak and electromagnetic forces – i.e. all the fundamental forces except for gravity. According to the SM, all matter can be described as a construct of 17 fundamental particles and their corresponding anti-particles (some which are their own), see Figure 1. These fundamental particles described by the SM can together form so called composite particles.

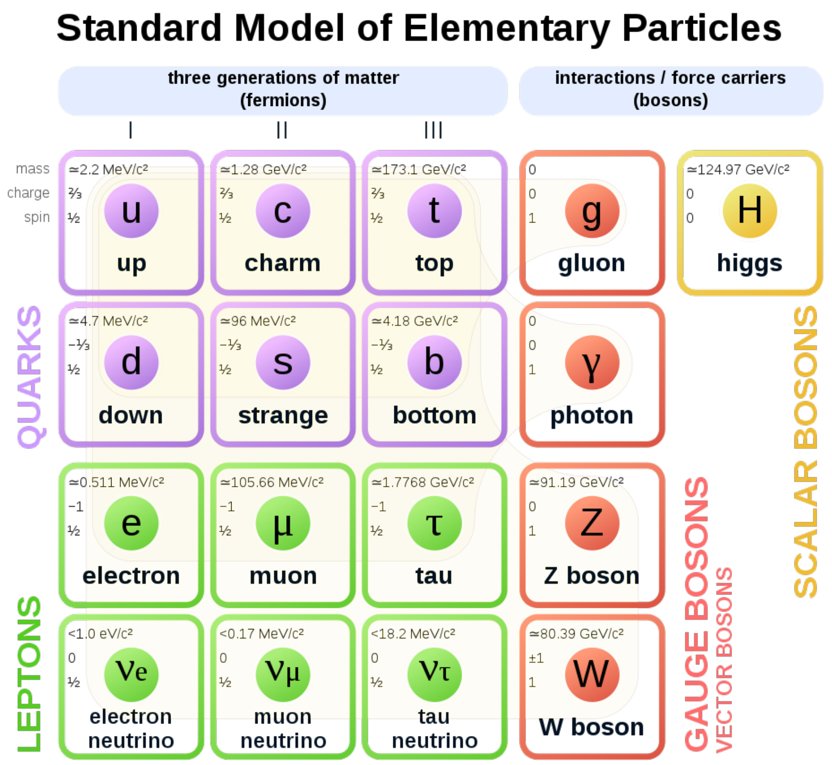

Standard Model of Elementary Particles
three generations of matter (fermions)
interactions / force carriers (bosons)
I
II
III
mass
charge
spin
≃2.2 MeV/c² ⅔ ½ u up
≃1.28 GeV/c² ⅔ ½ c charm
≃173.1 GeV/c² ⅔ ½ t top
0 0 1 g gluon
≃124.97 GeV/c² 0 0 H higgs
≃4.7 MeV/c² −⅓ ½ d down
≃96 MeV/c² −⅓ ½ s strange
≃4.18 GeV/c² −⅓ ½ b bottom
0 0 1 γ photon
≃0.511 MeV/c² −1 ½ e electron
≃105.66 MeV/c² −1 ½ μ muon
≃1.7768 GeV/c² −1 ½ τ tau
≃91.19 GeV/c² 0 1 Z Z boson
<1.0 eV/c² 0 ½ νe electron neutrino
<0.17 MeV/c² 0 ½ νμ muon neutrino
<18.2 MeV/c² 0 ½ ντ tau neutrino
≃80.39 GeV/c² ±1 1 W W boson
QUARKS
LEPTONS
GAUGE BOSONS
VECTOR BOSONS
SCALAR BOSONS

Figure 1: Fundamental particles as described by the standard model [11].

The SM separates particles into two main groups: fermions and bosons. Fermions, the three leftmost columns in Figure 1, all have their own distinct antiparticle. They have half odd integer spin and are subcategorized into *quarks* and *leptons*. The quarks cannot exist on their own but instead, they interact strongly and form composite particles such as protons. Leptons, on the other hand, can exist freely and are not affected by the strong force as far as we know. The two rightmost columns describe bosons. They are force carriers and mediate the interactions between the other particles. For example, the photon mediates the electromagnetic interaction. The newly discovered (2012) Higgs boson generates the mass of the elementary particles [9].

1.1.1 The Quark Model

More specific than the SM is the Quark Model (QM). It classifies bound systems of quarks into groups depending on symmetry characteristics, and was first presented by Gell-Mann and Zweig [4]. Most of these bound systems, i.e. hadrons, are either baryons or mesons. Baryons consist of three quarks and the two most common and well-known baryons are protons (uud) and neutrons (udd). Their corresponding (but less common) antibaryons are the antiproton ($\bar{u}\bar{u}\bar{d}$) and antineutron ($\bar{u}\bar{d}\bar{d}$). The behaviour of protons and neutrons is relatively well studied, due to their abundance.

Mesons consist of a quark and an antiquark. This also means mesons will have whole-integer spin. Examples of mesons are pions: π^+ ($u\bar{d}$), π^- ($d\bar{u}$) and π^0 ($\frac{u\bar{u}-d\bar{d}}{\sqrt{2}}$). Pions are the lightest known hadrons, meaning they are relatively easy to produce in a particle-collider. Pions π^+ and π^- are also easy to detect due to their charge. In this context, the charged π-mesons are considered to be stable due to their, on an experimental scale, relatively long lifetime.

1.1.2 Hyperons

The QM can also describe hyperons, the particles of interest in this particular study. A hyperon is a type of baryon. A hyperon is similar to a proton, but one or more of the up or down quarks has been replaced by a strange quark. This makes the hyperon significantly heavier. These particles have so far been studied less extensively.

Hyperons are relatively long lived, roughly 10^{-10} s. This means that they travel a measurable distance when moving at speeds close to the speed of light, which is the case for particles produced at many modern accelerator facilities. The hyperons are typically not measured directly but are identified by their decay products. These are typically protons, neutrons and mesons. Reactions where pairs of hyperons and antihyperons are produced can be seen as a type of laboratory environment where matter and antimatter is created on equal terms. This means their respective behaviours can be measured, quantified and compared. Hyperon-antihyperon pairs are therefore ideal for studying potential asymmetries between matter and antimatter. This is a type of measurement in particle physics which relies more on high precision and detail to hopefully detect something interesting, whereas experiments like CERN's Large Hadron Collider focus more on getting closer to the conditions of the Big Bang by going to higher and higher energies.

1.2 Why Does the Universe Consist of Matter?

Theories exist about some fundamental symmetries in the universe, some of which can be seen illustrated in Figure 2. Charge conjucation-symmetry, very roughly implies if all fundamental charges changed signs, i.e. positive become negative and vice versa, things would behave the same. Similarly, parity symmetry essentially means mirroring all spatial coordinates and everything behaving the same. A consequence of this, as seen in the image, would be the spin orientation of particles, i.e. the direction of the intrinsic angular momentum or the “inner magnet”. Combining these two symmetries yields CP-symmetry, i.e. simultaneously mirroring charges and spatial coordinates.

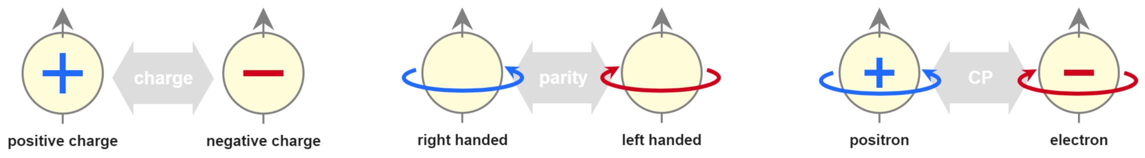


Figure 2: Visualization of C-, P- and CP-symmetry.

This symmetry is intimately connected to matter and antimatter. Imagine an alien civilization with which we can somehow communicate across the universe. How could we know they do not live in a world of antimatter? Well, for one we should definitely notice radiation from the intersection between the matter and antimatter where fierce annihilation would take place. But neglecting that, if all matter was replaced by antimatter and vice versa, would the laws of physics change? Most physical phenomena behave the same in a spatially mirrored and charge conjugated world. In fact, mirrored images are used frequently in movies, and it is impossible to tell the difference [8]. Interestingly though, CP-symmetry does not always hold [5].

A fundamental assumption in physics is that equal amounts of matter and antimatter were created in the Big Bang. If CP-symmetry were perfect, the presumably equal amounts of matter and antimatter created in the Big Bang would quickly have annihilated. However, if CP symmetry can be spontaneously broken, it is possible that matter can be enriched with respect to antimatter through a dynamic process called Baryogenesis [6]. For a time, it was believed CP-symmetry was universal and always held true. As far as we know, all interactions governed by gravity, electromagnetism and the strong nuclear force show this mirror-symmetric behaviour. As it turns out, interactions governed by the weak nuclear force do not always behave this way. This fundamental asymmetry is often called violating CP-symmetry, or CP-violation. CP-violation has already been observed, though not to a large enough extent that it explains the abundance of matter in the universe [10]. Thus, particle physicists are on the lookout for CP-violating processes which might answer this fundamental question.

A slight side note is the believed existence of an even more general symmetry: Charge-Parity-Time, also adding time reversal. This would mean reversing a reaction in time would perfectly recreate its reactants. A good summary of fundamental symmetries can be viewed via Ref. [8].

1.2.1 Recent Work in Uppsala

In particular, the hadron physics group at Uppsala University within which this project has been conducted, is studying the decay of Λ hyperons and $\overline{\Lambda}$ antihyperons. The hope is to find a strong enough CP-violation to explain why the universe consists of matter. The method used to analyze the decay is model independent in that it does not assume certain values for the decay parameters involved, nor CP-symmetry. While the SM does allow CP-violation, it does so only to a small extent. This means a finding might also contradict the SM, which would imply a veritable revolution.

1.3 Scientific Goals of this Thesis and Motivation

The goal of this project is to make the tools based on the formalism developed by the Uppsala group more accessible to other researchers, teachers and students in order to spread its use. An implementation which conducts a maximum log-likelihood fit to data according to the Uppsala method using the C++-based package ROOT [18] already existed. However, this program is bulky and requires very specific expertise, which makes it less user-friendly. Moreover, it is not maintained with respect to supported ROOT and C++ libraries. This is setting the bar high for any student projects, and so far, a large amount of time had to be spent on learning the software tools and compiling libraries. As a consequence, the physics scope of BSc and MSc theses have been somewhat limited.

To really facilitate the program's use and expedite scientific progress, an implementation which requires little setup, is easy to use and easy to modify is preferable. It should also have similar statistical precision and accuracy and run within a reasonable time frame. What is seen as ”reasonable” is highly subjective, but as a rough reference, several hours will be considered too slow.

The idea is that this can be achieved by rewriting the program in Python. Python is a high level language, meaning it requires little hardware-related work to function, such as so-called garbage collection or explicit variable initiation. Python has simple syntax, is easy to setup and run and is growing in popularity, both amongst researchers and in educational facilities. This approach enables students and teachers with less experience to use and analyze the method which allows more people to join the research groups. It also allows for putting more focus on the physics problem itself rather than compiling and running a program. Quality of research improves as researchers are not required to write their own code or download old, unmaintained software. Being able to confidently use this program also lays the foundation to more complicated cases and future research. Maintainability becomes more and more valuable at the long time-scales that this sort of research is associated with. Furthermore, the chances of making interesting discoveries increase. Accessibility is an essential aspect in all fields, and the user-friendliness of the Python programming language makes for a suitable tool to improve this.

Summarizing, a complete program should:

- Be able to make a multidimensional parameter estimation.
- Be easy to setup and run.
- Have a similar numerical performance compared to the current C++/ROOT program. That means error estimates and a covariance matrix as well as similar errors for similar data sizes.
- Be sufficiently fast to analyse realistically sized data samples within reasonable time.
- Be easy to read, maintain and modify.

2 Formalism

In this section, all implemented physics-related formulas will be presented, as well as some physical background. Formulas are split into different event types: 1D, 2D and 5D. These all essentially stem from the total formalism only simplified down to various levels of complexity.

2.1 Hyperon anti-Hyperon Production from e^-e^+ annihilation

One way to produce a hyperon anti-hyperon pair is through e^-e^+ annihilation. At certain energy levels, a type of resonance called J/ψ can be produced from the energy that is liberated in the annihilation, see Figure 3. The J/ψ consists of a charm- anticharm quark pair and has the same quantum number as the photon, i.e. spin 1 and negative parity. It has a relatively high probability of decaying into a $\Lambda\overline{\Lambda}$-pair, namely around $(1.89 \pm 0.09) \times 10^{-3}$ [12]. This process is shown in Figure 3. One experiment that is optimised for J/ψ production is the BEijing Spectrometer 3 (BESIII) in China, where the Uppsala hadron physics group is a member institute since 2012. The Uppsala group has taken a particular interest in the $\Lambda\overline{\Lambda}$-pair produced either from J/ψ decays [13] or from virtual photons [14]. The data analyzed by this program is already calibrated and filtered for relevant events, and the task of the program is to estimate physics parameters, which is the final step of the experimental chain.

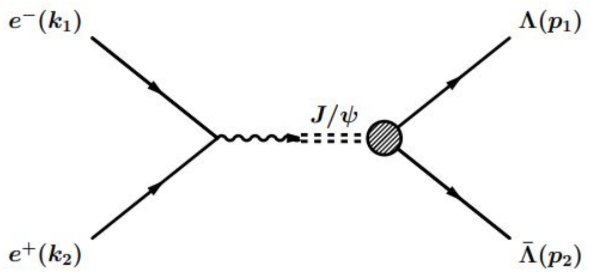


Figure 3: Feynman diagram describing the reaction $e^-e^+ \rightarrow \Lambda\bar{\Lambda}$ mediated by J/ψ resonance [2].

As can be seen in Figure 3, the e^+e^- pair annihilates to produce J/ψ, which instantly (10^{-20} s [12]) decays. With a good event selection filter, a pure set of these events can be obtained. As depicted in Figure 4, $\Lambda\overline{\Lambda}$ each have a probability of around $(63.9{\pm}0.5)\%$ [12] of decaying to $p\pi^-$ and $\bar{p}\pi^+$ respectively. One can either try to detect both charged decays (double-tag) or only one (single-tag). In the latter case, the decay that is ignored does not necessarily have to consist of charged particles, but can also be e.g. $n\pi^0$ ($(35.8 \pm 0.5)\%$ [12]).

2.2 Formalism for Double-Tag (5D)

The complete, 5D or double-tag formalism can be expressed in two different ways: in the covariant and in the helicity form. They describe the same physics, but from different reference frames. In the covariant one, all angles are defined in the e^+e^- centre-of-mass (CM) system. In the helicity formalism, the angles are defined in three different reference frames: the first is the e^+e^- CM system, the second is the hyperon rest frame and the third the antihyperon rest frame. When all final state particles are measured in the detector, it is called "double tag" because both the hyperon and the antihyperon and their subsequent decays are considered. The notation that will be used is, for the scattering angles: $\boldsymbol{\xi} = (\theta, \theta_1, \phi_1, \theta_2, \phi_2)$, where the various angles correspond to different decay particles; subindex 1 corresponds to Λ and 2 to $\bar{\Lambda}$. Precise definitions of these can be found in the derivation of the respective formalisms in their respective papers. The physics parameters are denoted: $\boldsymbol{\beta} = (\alpha, \Delta\Phi, \alpha_1, \alpha_2)$, where α and $\Delta\Phi$ are production related functions of the energy, whereas α_1 and α_2 are decay constants. Occasionally I will simply write the distribution function $\mathcal{W}$ as dependent on both $\boldsymbol{\beta}$ and $\boldsymbol{\xi}$, i.e. $\mathcal{W}(\boldsymbol{\beta}, \boldsymbol{\xi})$, since the estimation varies $\boldsymbol{\beta}$. α_1 and α_2 are the asymmetry parameters for the $\Lambda \to p\pi^-$ and $\bar{\Lambda} \to \bar{p}\pi^+$ decays respectively. If $\alpha_\Lambda \neq \alpha_{\bar{\Lambda}}$, this would imply CP-violation.

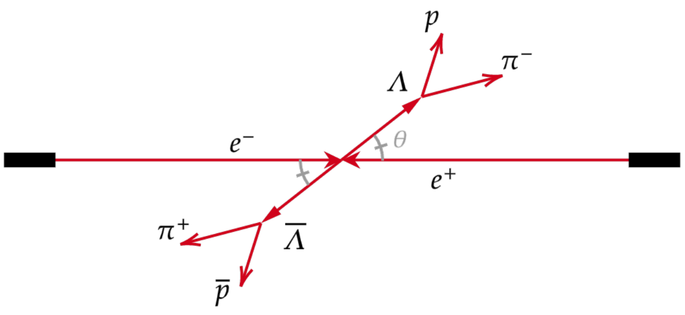


Figure 4: Schematic illustrating the decay $e^+e^- \to J/\psi \to \bar{\Lambda}\,(\to \bar{p}\pi^+)\,\Lambda\,(\to p\pi^-)$ (double-tag). Both the particle and the antiparticle decay is considered

2.2.1 The Covariant Formalism

The covariant formalism can be seen as somewhat easier to read. It is covariant in the sense that all angles are defined in the same reference-frame, namely that of the e^+e^--beam. It follows as presented by Fäldt and Kupsc [2]. It describes the cross-section distribution for spin 1/2 hyperons, such as the $e^+e^- \to J/\psi \to \bar{\Lambda}\,(\to \bar{p}\pi^+)\,\Lambda\,(\to p\pi^-)$ reaction. The exact derivation of the formalism is not covered here. Below follow the equations.

$$\begin{aligned}\mathcal{W}(\boldsymbol{\xi}) &= \mathcal{F}_0(\boldsymbol{\xi}) + \alpha\mathcal{F}_5(\boldsymbol{\xi}) \\ &\quad + \alpha_1\alpha_2\left(\mathcal{F}_1(\boldsymbol{\xi}) + \sqrt{1-\alpha^2}\cos(\Delta\Phi)\mathcal{F}_2(\boldsymbol{\xi}) + \alpha\mathcal{F}_6(\boldsymbol{\xi})\right) \\ &\quad + \sqrt{1-\alpha^2}\sin(\Delta\Phi)\left(\alpha_1\mathcal{F}_3(\boldsymbol{\xi}) + \alpha_2\mathcal{F}_4(\boldsymbol{\xi})\right)\end{aligned} \tag{1}$$

$$\begin{aligned}\mathcal{F}_0(\boldsymbol{\xi}) &= 1 \\ \mathcal{F}_1(\boldsymbol{\xi}) &= \sin^2\theta\sin\theta_1\sin\theta_2\cos\phi_1\cos\phi_2 + \cos^2\theta\cos\theta_1\cos\theta_2 \\ \mathcal{F}_2(\boldsymbol{\xi}) &= \sin\theta\cos\theta\,(\sin\theta_1\cos\theta_2\cos\phi_1 + \cos\theta_1\sin\theta_2\cos\phi_2) \\ \mathcal{F}_3(\boldsymbol{\xi}) &= \sin\theta\cos\theta\sin\theta_1\sin\phi_1 \\ \mathcal{F}_4(\boldsymbol{\xi}) &= \sin\theta\cos\theta\sin\theta_2\sin\phi_2 \\ \mathcal{F}_5(\boldsymbol{\xi}) &= \cos^2\theta \\ \mathcal{F}_6(\boldsymbol{\xi}) &= \cos\theta_1\cos\theta_2 - \sin^2\theta\sin\theta_1\sin\theta_2\sin\phi_1\sin\phi_2\end{aligned} \tag{2}$$

Notice how functions $\mathcal{F}_i$ only depend on the measured angles $\boldsymbol{\xi}$, and not the parameters $\boldsymbol{\beta}$. For the example $e^+e^- \to J/\psi \to \bar{\Lambda}\,(\to \bar{p}\pi^+)\,\Lambda\,(\to p\pi^-)$, subindex 1 and 2 for θ, ϕ, α correspond to Λ and $\overline{\Lambda}$ respectively. These subindices are also used in the Python implementation. The exact definitions of these angles can be found in Ref. [2].

2.2.2 The Helicity Formalism

The helicity formalism results in a slightly more bulky expression. Also this describes the cross-section distribution for spin 1/2 hyperons, such as the $e^+e^- \to J/\psi \to \bar{\Lambda}\,(\to \bar{p}\pi^+)\,\Lambda\,(\to p\pi^-)$ reaction, now from a particle-local reference frame. A sketch of how the angles in $\boldsymbol{\beta}$ are defined can be found in Figure 5. A detailed derivation and more precise explanation of the terms can be found in Ref. [3]. Note that while these angles are denoted in the same way for the covariant formalism, they are defined slightly differently.

$$\mathcal{W}(\boldsymbol{\xi},\boldsymbol{\beta}) \propto \sum_{\mu,\bar{\nu}=0}^{3} C_{\mu\bar{\nu}}\,(\theta)\, a^{\Lambda}_{\mu 0} a^{\bar{\Lambda}}_{\bar{\nu}0} \tag{3}$$

$$\begin{aligned} a_{00} &= 1, \\ a_{10} &= \alpha_D \cos\phi_D \sin\theta_D, \\ a_{20} &= \alpha_D \sin\theta_D \sin\phi_D, \\ a_{30} &= \alpha_D \cos\theta_D \end{aligned} \tag{4}$$

Showing all used non-zero terms for the case where the e^+ and e^- beams are unpolarized. Many elements in C are zero since $J/\psi \to \Lambda\bar{\Lambda}$ is parity-conserving.

$$\begin{aligned} C_{00} &= 2\,(1+\alpha\cos^2\theta) \\ C_{02} &= 2\sqrt{1-\alpha^2}\sin\theta\cos\theta\sin(\Delta\Phi) \\ C_{11} &= 2\sin^2\theta \\ C_{13} &= 2\sqrt{1-\alpha^2}\sin\theta\cos\theta\cos(\Delta\Phi) \\ C_{20} &= -C_{02} \\ C_{22} &= \alpha C_{11}, \\ C_{31} &= -C_{13}, \\ C_{33} &= -2\,(\alpha+\cos^2\theta) \end{aligned} \tag{5}$$

These equations show the specific $\Lambda\overline{\Lambda}$-decay case, while the formalism derived in [3] are more general. Here, sub-index D in a_{i0} corresponds to Λ or $\bar{\Lambda}$ depending on the super-index of a in equation 3. C is the spin-correlation matrix describing $J/\psi \to \Lambda\bar{\Lambda}$. It describes how the spin of Λ and $\bar{\Lambda}$ are entangled. $a^{\Lambda}_{\mu 0}$ ($a^{\bar{\Lambda}}_{\bar{\nu}0}$) is the spin-density matrix for the hyperon (antihyperon).

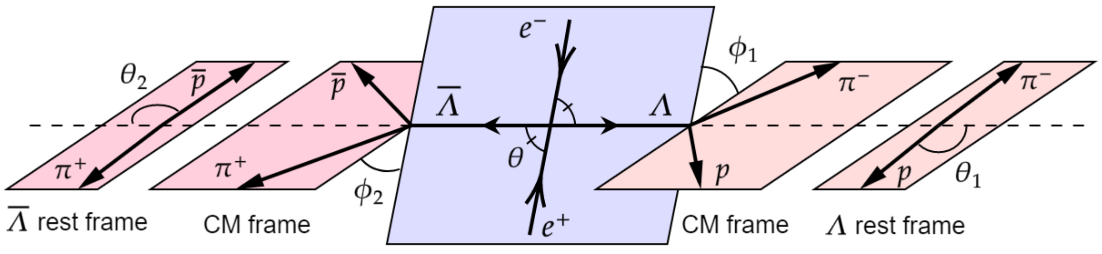


Figure 5: Sketch of angle definition for the helicity formalism for the example $e^+e^- \to J/\psi \to \bar{\Lambda}\,(\to \bar{p}\pi^+)\,\Lambda\,(\to p\pi^-)$.

2.3 Formalism for Single-Tag (2D)

Let us now look into the case when the decay product of either the hyperon or the antihyperon is detected – thus the name “single tag”. The single-tag formalism can be derived by integrating equation 1 and 2 over θ_2 and ϕ_2. This means that only the angles θ_1 and ϕ_1 are measured while θ_2 and ϕ_2 can take any value. Then, angles θ_1 and ϕ_1 can be substituted by the proton angle θ_p defined as angle between the proton and the production plane spanned by the e^-e^+-beams and $\Lambda\overline{\Lambda}$. Here, either $\alpha_\Lambda = -\alpha_{\bar{\Lambda}} = \alpha_{\Lambda/\bar{\Lambda}} = 0.754$ is assumed or one solves for the product $\alpha_{\Lambda/\bar{\Lambda}} \sin(\Delta\Phi)$ together. As illustrated in Figure 6, in this case only one of the two decay-chains is considered, which is why the process is often also called single-tag.

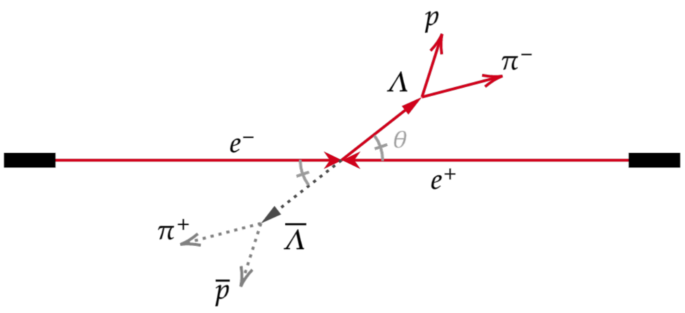


Figure 6: Schematic illustrating single-tag. Only one of the decay chains is considered (not necessarily Λ).

$$\begin{aligned} \mathcal{W}_{\Lambda/\bar{\Lambda}}(\boldsymbol{\xi}) &= \mathcal{F}_0(\boldsymbol{\xi}) + \alpha\mathcal{F}_2(\boldsymbol{\xi}) \\ &\quad + \alpha_{\Lambda/\bar{\Lambda}}\sqrt{1-\alpha^2}\sin(\Delta\Phi)\mathcal{F}_1(\boldsymbol{\xi}) \end{aligned} \tag{6}$$

$$\begin{aligned} \mathcal{F}_0(\boldsymbol{\xi}) &= 1 \\ \mathcal{F}_1(\boldsymbol{\xi}) &= \sin\theta\cos\theta\cos\theta_p \\ \mathcal{F}_2(\boldsymbol{\xi}) &= \cos^2\theta \end{aligned} \tag{7}$$

The formalism simplifies to only being dependent on $\boldsymbol{\xi} = (\theta, \theta_p)$ and $\boldsymbol{\beta} = (\alpha, \sin(\Delta\Phi))$, i.e. 2 degrees of freedom. This formalism has fewer components and hence, its implementation is less error-prone. This case uses numerical integration with normalization-data which together with the parts already implemented in 1D give all the puzzle-pieces needed for the 5D-implementation.

2.4 Formalism for 1D

Let us only consider the production of the hyperon-antihyperon pair, and disregard their subsequent decays. In an experiment, this basically means that one reconstructs the Λ or $\overline{\Lambda}$ from their decay products, without looking into the decay distributions themselves. A two-body reaction like $e^+e^- \to \Lambda\bar{\Lambda}$ can be described by one independent variable, i.e. the emission angle θ of either the produced hyperon or the antihyperon. Once this is known, everything else can be determined from pure kinematic relations. The 1-dimensional formalism can be derived from equations 7 and 6 by integrating over θ_p (which integrates to zero). The distribution follows

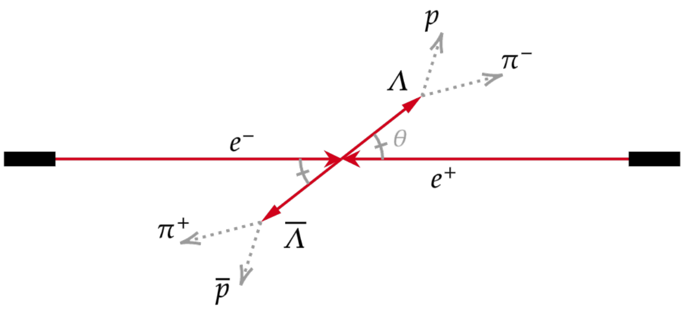


Figure 7: Schematic illustrating the 1D case. Only the production is considered.

$$\mathcal{W}(\boldsymbol{\xi}) = 1 + \alpha \cos^2 \theta. \tag{8}$$

Since $\cos\theta$ is linearly distributed, it can be integrated analytically to perform the normalization:

$$N_{orm} = \int_{-1}^{1} 1 + \alpha x^2 \, dx = \frac{6 + 2\alpha}{3} \tag{9}$$

We obtain the normalisation constant N_{orm} and can then construct the probability density function (PDF) P. Since the probability of observing a point anywhere must be 1, we know $\int_\xi P \, d\xi = 1$. This amounts to dividing $\mathcal{W}$ by N_{orm}, i.e.:

$$P(\boldsymbol{\xi}, \boldsymbol{\beta}) = P(\theta, \alpha) = \frac{\mathcal{W}(\alpha, \theta)}{N_{orm}} = \frac{3}{6 + 2\alpha}(1 + \alpha \cos^2(\theta)) \tag{10}$$

Applying the parameter-estimation then simply becomes a matter of evaluating P at each point in a randomly generated set X and see if the same α is returned.

3 Parameter Estimation

Parameter estimation is used across all industries: anything from estimating the growth rate of a virus to predicting stock market fluctuations. Here, we are looking to estimate parameters of a decay process for nuclear physics data. The contents of this section aim to explain the method used for parameter estimation. The choice of method is the same as the C++-implementation. Being logically intuitive, maximum likelihood estimation is widely used for statistical inference. The distribution function $\mathcal{W}(\boldsymbol{\beta}, \boldsymbol{\xi})$ mentioned here is described in detail in section 2.

3.1 Method of Maximum Log-Likelihood

Given a set of observed data and the shape of the distribution it stems from, we want to determine the parameters that best describe these data. The idea of maximum likelihood estimation (MLE) is, as the name implies, to maximize the likelihood for a set of parameters. This means constructing a likelihood function of making a certain observation for different parameter values and select the parameters that are most probable. A simple example where one can apply MLE would be estimating the parameters of a normal distribution: the mean and the width, which equals the standard deviation of the sample.

A PDF can be used to construct a likelihood function: it returns the probability of observing a certain point for a given parameter value. Observing an independent set of points is just the product of their probabilities. Thus, we could maximize for this product.

$$L(\boldsymbol{\beta}|\boldsymbol{\xi}) = \prod_{i=1}^{n} P(\boldsymbol{\xi}_i|\boldsymbol{\beta}) \tag{11}$$

However, for large samples, this product will become so tiny it becomes practically impossible to calculate with a computer having limited precision. We would be multiplying several thousands of points, each of a value well below 1 and greater than zero. This is why we are considering the maximum *log*-likelihood (MLL). Since the logarithm is monotonously increasing, the maximum of a function will be located at the same point as the logarithm of that function [1]. It is thus convenient to instead use

$$\log(L(\boldsymbol{\beta}|\boldsymbol{\xi})) = \sum_{i=1}^{n} \log(P(\boldsymbol{\xi}_i|\boldsymbol{\beta})) \tag{12}$$

In Equation 11 and 12, let $L(\boldsymbol{\xi}|\boldsymbol{\beta})$ denote the likelihood of observing data set $\boldsymbol{\xi}$ of size n given a set of parameters for the distribution $\boldsymbol{\beta}$, and P be the PDF. Analytically, the maximum likelihood parameters are found by taking the derivatives of the likelihood function with respect to the parameters and set the derivatives to zero. In reality, an analytical solution is not always feasible and instead, numerical methods need to be used. Hence, to write a program conducting a MLL estimation for spin 1/2 hyperon decays, the following is needed:

- a data set with the measured variables $\boldsymbol{\xi}$
- a probability density function in terms of points ξ_i in $\boldsymbol{\xi}$ and some physics parameters $\boldsymbol{\beta}$ to be estimated

For the formalism applied here, $\boldsymbol{\beta}$ will be 4-dimensional and each point in $\boldsymbol{\xi}$ 5-dimensional. For this formalism we do not calculate the PDF analytically, but use the non-normalised form $\mathcal{W}$. To obtain the PDF, this distribution needs to be normalized, as described in the following section (3.1.1).

3.1.1 Normalization

In order to get the PDF needed to run a MLL-fit, the distribution function $\mathcal{W}$ needs to be normalized. By definition, the PDF integrates to 1 since the total probability must be 100%. Thus, normalization just amounts to dividing $\mathcal{W}$ by

$$N_{orm}(\boldsymbol{\beta}) = \int_{\boldsymbol{\xi}} \mathcal{W}(\boldsymbol{\beta}, \boldsymbol{\xi})\, d\boldsymbol{\xi} \tag{13}$$

i.e. the integral over all of the variable-space $\boldsymbol{\xi}$. Here, N_{orm} is the normalization factor. In some cases this integral can be computed analytically, and in some any integrator would do the trick. For the applications this project focuses on, a Monte-Carlo (MC) integration method has to be used. The data points, $\boldsymbol{\xi}$ to be analyzed are measured in a detector that has a finite efficiency. This is due to imperfections and "blind spots" where there are constructional limits, for example where the beams enter. This should be accounted for in the likelihood function defined in Equation 11 by multiplying the PDF P_i with a factor ε_i that quantifies the efficiency. The efficiency is a function of the measured quantities ξ_i, hence $\varepsilon_i = \varepsilon(\xi_i)$. However, taking the logarithm of the likelihood rather than the likelihood itself, results in an extra term with a sum of efficiencies that does not depend on the physics parameters $\boldsymbol{\beta}$. The derivative of this sum with respect to the physics parameters is therefore zero, and hence it does not affect the location of the maximum but only the normalisation of the likelihood function. Conducting the normalization using an MC-method allows the use of a set of normalization data that is weighted in a way to compensate for imperfections in the detector. This normalization data is typically generated by simulating the detector.

MC integration means taking a random set of linearly distributed points over all of the variable domain, $\boldsymbol{\xi}$. This means generating a large set of random points – so large that any errors from the MC-integration become small in relation to the data to be fitted. The function to be integrated is then evaluated at each of these points, and the average A_{vr} taken. This means we now have an average value of the function over the domain, so to finalize the integration we simply multiply by the "area" k under the curve. This "area", in any dimension, is just the product of the parameter ranges: in 2D this is an area, in 3D a volume and so on. However, for use in a log-likelihood function, no multiplication by k is necessary.

$$\log(A_{vr}(\boldsymbol{\beta}) \cdot k) = \log(A_{vr}(\boldsymbol{\beta})) + \log(k) \tag{14}$$

Notice that k is constant in every normalization whereas A_{vr} depends on $\boldsymbol{\beta}$. By similar reasoning as for the efficiency factor ε, we simply get $N_{orm} = A_{vr}$.

Note how N_{orm} in equation 13 is dependent on $\boldsymbol{\beta}$. This means, when running the MLL optimization, N_{orm} has to be recalculated for each new guess of $\boldsymbol{\beta}$. As we will see in section 6 this is one of the most time-consuming part of the program.

3.2 Consistency and Bias

In order to verify the validity of a statistical fit and be able to draw conclusions from it, common methods to quantify its performance are consistency and bias. The task is to estimate some quantity β. A common way to denote the estimator for this quantity is with a hat, $\hat{\beta}$, to not confuse it with the actual, true value β. For an estimator to be considered useful, consistency is usually a requirement. If an estimator $\hat{\beta}$ is to be considered consistent, it should converge to the true value β for large sample sizes n. Since we often cannot know the true value of β, by convergence we mean the probability P that their difference is greater than some arbitrary $\epsilon > 0$ goes to zero. This can also be written as

$$\lim_{n\to\infty} P(|\hat{\beta} - \beta| > \epsilon) = 0 \tag{15}$$

When testing for consistency, it is common to apply the estimator to differently sized n and see if this limit appears to hold.

Bias is another type of statistical measure. It is defined as

$$b = E[\hat{\beta}] - \beta \tag{16}$$

where $E[\hat{\beta}]$ is the expected value of several measurements of finite sample size. We call an estimator unbiased if $b = 0$ for any sample size n and asymptotically unbiased if $b = 0$ as $n \to \infty$. Having an unbiased estimator is particularly valuable if one wants to compare results between experiments. Generally if the bias of an estimator is small relative to its standard deviation, it is considered acceptable [1].

4 Software Tools

In this section, the Python libraries and other tools used in the program will be described. All of the used packages are easy to install with Python `pip`, and are relatively standard in these types of applications. A general description of the packages as well as their main use in this project is explained below. For more detailed information, the reader is referred to references given in each case. The C++/`ROOT`-implementation is based on the helicity formalism described in equation 3. So far, the helicity formalism has also been implemented in Python, but it is likely the covariant formalism will also be in the near future.

4.1 Existing C++ Implementation, `ROOT` and `MINUIT`

The C++-program uses the library `ROOT` for function optimization and error estimation. `ROOT` is a data analysis framework developed at CERN mainly for high energy physics, and is open source [18]. The implementation uses `ROOT`'s `MINUIT` library, which includes the `Minuit` function optimizer and error estimator.

The `ROOT` library is undeniably highly efficient in operation. While its tools are well-tested and in many ways developed specifically for analyzing nuclear physics data, it has its drawbacks. This implementation requires a lot of setup such as a C++ compiler and a `ROOT` installation. Furthermore, it is relatively difficult to modify since C++ as a language is more syntactically complex compared to for example Python. `ROOT` is also a very large, non-modular library, meaning the user needs to download multiple gigabytes only to use one small part of it.

While the C++-implementation is highly functional and performs well for the task it was designed for, it requires a relatively solid experience and knowledge of `ROOT`, alternatively access to frequent expert guidance to set up and modify. It is neither open access nor easily maintained. This has resulted in other research groups writing their own programs from scratch, which makes cross examination and cooperation harder and results more prone to hard to find code errors. While this implementation has been used to verify the results of the Python program, new Python libraries and tools were needed. A description of these follows.

4.2 `Numpy`

`Numpy` is perhaps the most well-known Python library for processing data. It is free, open-source software which aims to ease scientific computing in Python [16], and is continuously being updated and improved. The library is especially powerful when working with large sets of numerical data, and its data type `numpy.array` is a highly efficient way of storing and working with data. The `numpy.arrays` can be worked with similarly to Python lists, but have a well-defined size and data type. Additionally, it has several built-in functions which allow for fast `numpy.array` to `numpy.array` operations. Many of its strengths are comparable to MATLAB, but in addition it is free of charge and open source. Installing `Numpy` can be done easily using `pip`, the package size is roughly 16 MB [17], and as mentioned is very standard.

4.3 Scipy

`Scipy` is a ecosystem of open-source Python-software mainly for data analysis [19]. As opposed to `Numpy`, which mainly handles data management through vectorization, `Scipy` offers an array of tools to manipulate this data in various ways. The aforementioned ecosystem includes `Numpy` and the plotting library `matplotlib`, which `Scipy` is built upon and completely compatible with. Also this library is very standard for scientific computing in Python. In this project in particular, the `optimize` package and its function minimizer `minimize` is used. `scipy.optimize.minimize` supports several different minimization methods, though the default one, "BFGS", is used here. The function returns the minimized function value as well as the function parameters, together with an approximation of the inverse Hessian matix and Jacobian which could naively be used for error estimation.

4.4 iminuit

The `iminuit` package can be used for function minimization and error estimation of likelihood fits [20]. `iminuits` main class, `Minuit`, is very similar to the method used in the C++/`ROOT` implementation of the program, and is maintained by the CERN `ROOT`-team. Particularly useful is the error estimation and function minimizing for parameter estimation. The minimizing algorithm, called `Migrad`, is a local minimizer. It uses a combination of Newton steps and gradient-descents to look for a local minimum from a given starting point [20]. The `iminuit` error estimator is called `hesse`, and it can continue an error estimation from a function optimization with `Migrad`. Importantly, `iminuit` is not dependent on `ROOT`. However, `iminuit` yields the benefits of `ROOT MINUIT` in being highly efficient, but lets go of the heavy drape of the full `ROOT`-library and C++ language.

4.5 Compiled vs Interpreted Programming Languages

A difficulty when analyzing large sets of data in Python is the execution time. Python, out of the box, is slow. There are ways to speed this up, such as the vectorization introduced by `Numpy`, but in essence the issue lies at the heart of the Python programming language itself: it is an interpreted language. To get an idea for the type of difficulty this entails, consider for instance the following piece of code:

```python
def sumEval(LargeDataSet, func):
    s = 0
```

```
3     for data in LargeDataSet:
4         s += func(*data)    # <--- !!!
5     return s
```

It evaluates a function `func` at every point `data` in some `LargeDataSet` and returns their sum. In every iteration of line 4, Python needs to type-check, i.e. make sure the arguments to `func` and the addition to `s` are legal, and translate to machine code. As opposed to compiled languages like C or C++, Python converts human-readable code to machine code only at runtime. This takes time, which is why function calls in Python are slow. This slowness is one of the main drawbacks of an interpreted language. In a compiled language, the data type of `data` and `s` would be predefined and fix, and `func` pre-compiled. This generally allows for a significant speed improvement over an interpreted language.

So why are not all programming languages compiled? The compiled-style syntax is generally more complicated and syntactically strict. Since these compiled languages do not run the code before compiling, it also means errors in the code can be hard to pinpoint as error-messages often just say compiling failed, not where in the code it failed. The compiler itself also needs to be installed and called before running a program, adding an extra step before execution. Thus, Python being interpreted has plenty of advantages when it comes to accessibility, which is one of the main reasons why Python was selected as the language of choice in the first place. Benefits include:

- easy debugging since the interpreter can identify the precise line where the program failed
- no need to compile the code before running it and thus no need for a separate compiler
- dynamical data types, i.e. variables need not be initialized and can change type

This all makes Python one of languages with the simplest syntax and it is considered very user-friendly. However, this also means Python evaluates each line of code at a time and can make no presumptions about the data the program is running for: it could be anything. As such, no computational "shortcuts" can be taken during execution and thus, in particular the iterated function evaluations take a long time.

The kind of operations described in the previous snippet are used extensively in the MLL-fit. While this slowness is not devastating, it is undesirable. Code that runs too slow is hard to debug and modify, which is one of the things this project sought to facilitate. Fortunately, thanks to significant efforts of the community of Python developers, there are solutions to this. One solution which does not require us to instead use a compiled language is the `Numba` library. This permits the use of for-loops and similar simple syntax, and is used in this implementation. Other options include `Numpy` and Pandas dataframe [21], which essentially are pre-compiled libraries. Many of these alternatives are actually written in C or C++. Typically, these solutions do not use for-loops. While this particular implementation could likely have done without for-loops, `Numba` was deemed an easy-to-read, high-level solution.

4.6 Numba

The open-source Python compiler Numba "Just-In-Time" (JIT) compiles a subset of Python an Numpy code and translates it to fast machine code. JIT means the code is not pre-compiled but rather is compiled at runtime. Numba uses Python function decorators to compile only select parts of code by reading the bytecode and automatically optimizing it. Function decorators follow the syntax: `func = decoratingFunc(func)`, which in Python is equivalent to writing `@decoratingFunc` above the definition of `func`. It essentially modifies a function in some way, here turning it into a faster version of itself. Every time the decorated function is called, this compiled version used. This can give a speed increase of up to several orders of magnitude, which is significant [22].

4.6.1 Compiled-Style Syntax

Numba is compatible with default Python, the Numpy library, and a few other libraries. Only requiring a function decorator, it is almost as easy as adding it to a Python function and getting a notable performance-boost. However, there is a small caveat: As mentioned in section 4.5, the issue fundamentally lies in the way Python is built. In order to really make a performance difference, some of the interpreted-style programming needs to be put aside. See the following code snippet for example:

```python
 1  # Example of a dynamically written function
 2  def maxOfList():
 3      result = 'Nothing yet'
 4      lst = [1, 8, 3, 9, 14]
 5      for x in lst:
 6          if result=='Nothing yet':
 7              result = x
 8          elif x > result:
 9              result = x
10      return result
```

The issue here lies in the variable `result`: it changes types. As mentioned, this is something that is not permitted in compiled languages, and for good reason. When trying to optimize this code, Numba will for example look at the `result` variable and see that it is compared to both string-type (line 6) and integer-type data (line 8). Python will not return any errors, since it will never actually have to compare different data types. However, when compiling, this is impossible to know without running the code. Thus, in order to be able to make these assumptions, fix data types need to be used. Python lists are a prime example of non-fix data types. They can contain different data types, sizes, and can even change these during runtime. Iterating over a Python list is therefore very hard to significantly optimize. Numba offers some of their own objects for this, such as "typed lists", but a much faster and easier solution in this project is a standard numpy array which Numba is compatible with. Thus, a stricter, compiled-styled syntax need to be used. Inspiration for the snippet above and more about how to satisfy Numba can be found in Ref. [24].

4.6.2 Use

The way Numba optimizes can lead to a style of code not typically used in Python for the type of task, simply because it is more compatible with Numba. However, this style is typically simpler, for example using for-loops where numpy-vectorization would otherwise be used. This might look odd to an experienced data analyst, but in a sense makes things even easier to read for a beginner. Importantly, the actual syntax still is normal Python; no Numba-specific syntax is used. Another advantage is the fact that Numba only compiles the part of the code that is slow. Since it decorates functions, parts of the program that slow down execution can be separated out into their own function. Examples in this project are the Monte-Carlo integration or the log-likelihood sum. These functions are @JIT-decorated and then only this part of the code needs to follow the stricter compiled-styled syntax which Numba requires, essentially leaving the rest of the work progress totally unchanged. Similar tools like PyPy [23] exist, but for example this speeds up the entire program without any modification, and limits libraries that can be used. It also has a much smaller effect, since it cannot make the fundamental compiled-style optimizations Numba can. Additionally, it can be more challenging to use. The first iteration Numba runs any piece of code will typically take significantly longer (not than without Numba, but compared to later Numba iterations), since Numba compiles during this time.

5 Implementation

When writing the program, testing was essential; firstly to make sure no errors occurred, and secondly to ensure the validity of the solution and compare results to those obtained by the C++/ROOT program. This testing data was supplied by Viktor Thorén (2D) and Patrik Adlarson (5D), and the generation of the testing data was outside the scope of the implementation.

5.1 Method Choices and Work Progress

Summarizing the general requirements and main components for the respective cases 5D, 2D and 1D:

- **1D**: Generate input data, implement 1D formalism, analytic normalization, MLL-function optimization, time not an issue
- **2D**: Read input from file, implement 2D-formalism, MC-normalization, MLL-function optimization, Numba for speed
- **5D**: Read input from file, implement 5D-formalism, MC-normalization, MLL-function optimization, Numba for speed

5.1.1 The 1D Case

In order to start simple, the 1D problem described in section 2.4 was implemented as a proof-of-principle. Implementing the formalism in this case is less error-prone compared to the 2D case, and thus focus could be put on the MLL-estimator. This allowed for an outline of the program to be established, and the concepts to be studied and understood.

5.1.2 The 2D Case

Moving to 2D requires a more generalized approach than the 1D case while still being simpler to debug than the 5D version. Here, the MC-integration was implemented as described in section 3.1.1. The numerous iterative function evaluations at this point served as an example of Python's poor capability of handling large sets of data and function calls. Execution times reached several hours during testing (see computer specifications in Appendix A.2) and became a limiting factor in the development and testing process. This called for efforts to optimize the run-time. Initially, vectorization through `Numpy` was considered, but it did not show too significant a performance improvement and required a lot of `Numpy`-specific syntax – not ideal for readability. This resulted in a reduction of the run-time to be in the ball-park of minutes, which is a clear improvement though still slower than the C++ program. Therefore, `Numba` was introduced. This yielded an even greater performance boost, reducing execution time to seconds. Also here, `Numpy` arrays were selected to be the main way of handling the in-data, since it proved efficient, straightforward and compatible with `Numba`. `Numba` typed-lists were initially used, but proved slower to convert to. At this point, the `scipy.optimize.minimize`-method returned an inverse Hessian matrix which was thought to be usable for approximating the covariance matrix and thus use for error estimation. No analysis of statistical precision and accuracy, i.e. consistency and bias, was conducted in the 2D-case.

5.1.3 The 5D Case

The syntactical evolution from 2D to 5D is rather insignificant: change input data, change $\mathcal{W}$ and change some `Numpy` array sizes to 5. For optimization, `iminuit` was selected since it closely resembled the method used in the C++/`ROOT`-program and thus was known to be working. It offers both parameter estimation and error estimates. However, `Minuit` turned out to require a relatively good initial guess, i.e. close to the true values of the parameter. Therefore, `scipy.optimize-minimize` was used to supply an estimate for the starting point for the `Minuit` fit. This allowed initial guesses to be about one order of magnitude further away from the true value compared to just using `Minuit`.

Just using `scipy.optimize.minimize` for optimization and error estimation was also tested. However, when it came time to run statistical analyses, the `scipy.optimize.minimize` inverse Hessian proved highly unreliable. Error estimations were wildly inconsistent and highly dependent on initial guess. It turns out, the way `scipy.optimize.minimize` calculates its inverse Hessian is not suitable for the kind of iterative optimization conducted here. `Scipy`'s optimizer estimates the inverse Hessian iteratively. If a solution is found in relatively few iterations there will be no time for the Hessian-estimation to build up proper information, as seems to be the case here [19]. It is worth noting that there likely is a way to get this to work but, considering a known-to-work alternative was close at hand, it was not investigated extensively. In the end, `scipy` was just used as a secondary optimizer, able find an initial guess for `iminuit`.

The Python implementation was developed without mimicking the C++-code in detail. The reason the C++-implementation was not used as a guide more was partly for precisely the same reasons this project started: it was complicated to understand and run for an inexperienced user (e.g. the author, initially). Another reason was to try to remain unbiased during development and hopefully find alternative methods which could then be compared to the corresponding C++-solutions and the best be selected. The idea was that this would lead to more efficient and a more “Pythonic” implementation, without confusing artefacts of C++-styled code.

5.2 Outline of the Code

Here, first the main algorithms used will be presented as pseudo-code, after which a more code-like outline of the entire program is presented. In Algorithm 1 below, the MC-integration used to find the normalizing factor N_{orm} for $\mathcal{W}$ is presented.

Algorithm 1: Monte-Carlo integration

```
Data: $\boldsymbol{\beta}$ guess, normalization data, $\mathcal{W}$-function
Result: $N_{orm}$
for $\boldsymbol{\xi}$ in normalization data do
    evaluate $\mathcal{W}$ at $(\boldsymbol{\beta}, \boldsymbol{\xi})$
    add evaluated value to sum
end
Return sum divided by size of normalization data
```

The algorithm used to calculate the negative log-likelihood at some guess $\boldsymbol{\beta}$ is presented as Algorithm 2. Here, the normalization data is a list, each element being a 5-tuple.

Algorithm 2: Negative log-likelihood summation

```
Data: $\boldsymbol{\beta}$ guess, $\boldsymbol{\xi}$-set, $\mathcal{W}$-function, $N_{orm}$
Result: negative log-likelihood of observing $\boldsymbol{\xi}$-set with parameters $\boldsymbol{\beta}$ guess
for $\xi_i$ in $\boldsymbol{\xi}$-set do
    add minus $\log(\mathcal{W}(\boldsymbol{\beta}, \boldsymbol{\xi}))$ to the resulting log-sum
end
add $n \cdot \log(N_{orm})$ to sum
Return sum
```

In Algorithm 2, $\boldsymbol{\xi}$-set is the signal data. In the 5D program it is a numpy array of length n, each element being a 5-tuple. $\boldsymbol{\beta}$-guess is a 4-tuple.

Algorithm 3: Reading of input data from .txt-file

```
Data: Filename, .txt-file containing data to be read, numberedInput boolean
Result: numpy array with $n$ rows, each containing input data points
outputList = [ ]
for line in open(filename) do
    split line into a list of individual numbers
    if numberedInput is True then
        ignore the first element in the split line
    end
    append split line to outputList
end
Convert list of lists to numpy array
Return numpy array
```

Algorithm 3 above is used to read both the signal data $\boldsymbol{\xi}$-set and the normalization data.

Below, an outline of the entire (5D) program is shown. It shows the most important functions used to make the code run. In the actual program. `WDoubleTag` is the double tag example, but can be replaced by any distribution function.

```
def WDoubleTag(alpha,dPhi,alpha1,alpha2, th,th1,ph1,th2,ph2):
    # returns W(beta, xi). Large expression.

def MCintegral(beta, normAngs, Wfunc):
    # eval W(beta) at each point in normAngs and averages
    # returns normalization

#def negLL(beta, xi_set, pdf, normSep=False, normAngs=[]):
def negLL(beta):
    # If normSep = True: normalize separately using normAngs
    #   # using MCintegral (at beta)
    # Iterates over all data points,
    #   # sums -log(W(*beta,*xi)/Norm) for xi in xi_set
    # returns (negative) likelihood for given beta

def main():
    # read input data from .txt (or .root)-files
    # define optimization parameters (tolerance, initial guess, bounds)
    # scipy.optimize.minimize negLL-function for improved intial guess of beta
    # result is used as a starting guess for Minuit which finishes the estimation,
    # and also makes an error estimation.
    # Present results: estimation ± sigma, covariance matrix.
```

Note that the general working principle is identical for the 2D and 5D programs, though the `WDoubleTag`-function becomes `WSingleTag`. Some variable names have been changed in order to be more consistent with the naming in this report. The actual code uses more general terminology when applicable, e.g. `par` instead of `beta` and `var` instead of `xi`. At the very start of the program, a few variables for the estimation can be defined:

- `dataFrom, dataTo`: Integers. Mainly for program analysis. Can specify an index-range subset of input data to use for estimation. Set to `0,0` to use full data set.
- `normFrom, normTo`: Integers. Mainly for program analysis. Can specify an index-range subset of input normalization-data to use for estimation. Set to `0,0` to use full normalization set.
- `dispIterInfo`: Boolean. If `True`, prints the current LL-value, normalization and time taken for each in every iteration. If `False`, just shows a loading bar.
- `use_scipy_for_initial_guess`: Boolean. If `True`, will use the initial guess with `scipy.optimize.minimize` with high tolerance and then use its result as the stating value for the `Minuit` optimizer. If `False`, it just uses `Minuit`.

- `signalData_filename`: String. Filename of signal-data, with complete specified path if desired.
- `normData_filename`: String. Filename of normalization-data, with complete specified path if desired.
- `directoryPathForSearch`: String. Path to a directory (could be ”`C:/`”) for where to search for the input-files if their complete path is not specified. This search will take some time if the directory to search in is large or at a high level.
- `numberedInput`: Boolean. If `True`, will skip the first column when reading the input data.
- `initGuess`: Tuple of floats. Specify initial guess of parameters to be used.
- `bnds`: Tuple of two-tuple of floats. Bounds on the parameters, needed only for `Scipy`.
- `ftol`: Float. Tolerance for `Scipy`.
- `parNames`: Tuple of strings. Names of parameters to be estimated. For display of output.

Note that the order of input-data should be the same as used in the function `W`. Below each function and their functionality is discussed.

5.2.1 `WDoubleTag`

The `WDoubleTag` function takes as input $(\boldsymbol{\beta}, \boldsymbol{\xi})$ and returns the function value at that point for those parameters. This is precisely what is described in section 3. The actual expression is defined using help-functions as described by equation 4 and 5 to improve readability and simplify debugging. It resembles these equations very closely. This function is called by both `MCintegral` and `negLL` which both need to make serveral function evaluations of $\mathcal{W}$.

5.2.2 `MCintegral`

This function calculates the Monte-Carlo integral for some function `Wfunc` defined with parameters `beta`, just as descibed in Algorithm 1. This function $\mathcal{W}(\boldsymbol{\beta})$ is evaluated at each point in `normAngs`, a set of normalization angles provided as input to the program. The function takes the sum of all possible function values and divides by the amount of calls (`len(normAngs)`). This resulting value is used for normalization of $\mathcal{W}$, as described in section 3.1.1. Note that in accordance to equation 14, the obtained value is not the actual integral estimation since the range of each variable is not taken into account.

5.2.3 `negLL`

The `negLL` function is in a sense the very core of this fit. It is the function using all the previously described functions to return an actual likelihood-value, given a parameter guess $\boldsymbol{\beta}$. Algorithm 2 describes the main functionality of `negLL`. The name `negLL` is chosen because it returns the *negative* log-likelihood. The reason for this is simply that a function minimizer rather than a maximizer is used, so finding the minimum of the negative likelihood is equivalent to finding the maximum of the positive likelihood.

As can be seen in the code outline, another set of input parameters was intended for `negLL`. Data used by the function would ideally have been sent in as function parameters. This was not achieved; Since the `Minuit` optimizer only permits the parameters it optimizes for to be function input, only $\boldsymbol{\beta}$ could be sent as a function parameter. The inelegant way around this was to define the other wanted input parameters as global variables, so the function can still access them. Using global variables is generally considered bad practise. This is because if the code is ever imported to another program, it could overwrite existing variables without notifying the user. Currently, these global variables are only ever initiated if running the `main` function. This slightly decreases the risk of unwanted import overwrites.

The actual functionality of the `negLL`-function is rather simple, and it pretty much follows Equation 12. Firstly, since $\mathcal{W}$ needs to be normalized, `normSep = True` and `MCintegral` is called which returns the normalization factor. Secondly, it calls a help function `iterativeLL` which iterates over each data-point `xi` in `xi_set` and sums `-log(WDoubleTag(*beta, *xi))`[1]. The reason why a separate help function is used is because it enables `Numba` to optimize it while the rest of the function `negLL` does not need to follow stricter compiled-styled syntax. This is also why it is done iteratively rather than e.g. with a vectorized approach. After receiving the negative log-sum, `negLL` takes `+ log(norm)*len(xi_set)` to refrain form having to execute a division in each term. This is fine since the logarithm of a product is just the sum of the logarithms of the terms. Since N_{orm} is the same in each product for a given likelihood-sum, it is equivalent to adding `log(norm)*len(xi_set)`, as show below:

$$\sum_{i=1}^{\texttt{len(xi\_set)}} -\log\left(\frac{\mathcal{W}(\boldsymbol{\beta},\xi_i)}{N_{orm}}\right) = \sum_{i=1}^{\texttt{len(xi\_set)}} \left(-\log(\mathcal{W}(\boldsymbol{\beta},\xi_i))\right) + \log(N_{orm}) \cdot \texttt{len(xi\_set)} \tag{17}$$

Finally, `negLL` returns this value, which is precisely the negative log-likelihood of observing `xi_set` given parameters `beta`.

In the code, `negLL` consist of some more lines of code which print information about the calculation. Depending on the variable `dispIterInfo`, it either displays the time it takes to compute the normalization factor and the likelihood sum as well as their respective values, or it just displays a loading bar. This functionality was added so that the user-end output would not get overwhelmed by iteration information that is most likely not interesting for anything but debugging or time analysis. However, it is convenient to have some kind of indicator that the program is doing something, why the loading bar was implemented.

[1]Recall that '*' packs or unpacks variables.

5.2.4 main

The main function ties everything together and makes the appropriate function calls. First off, signal- and normalization-data is read from a .txt- or .root-file. For the .txt-case this is performed in one long but efficient line of code which might warrant some explanation:

```
data = [ list(map(float,i.split()))[numberedInput:] for i in open(filename).readlines() ]
```

This line of code reads data precisely as described in Algorithm 3: from filename and stores it in data as a list of lists. The list is immediately converted to a Numpy array. Outermost is a list comprehension, which means whatever value the iterable inside the square-brackets takes will be the element of a list. The iterable is here each line in the .txt-file. This line is a string of values separated by a space, tab, or a comma, and is split into separate elements. These elements are converted to a floating-point number using map and finally the resulting map-object is converted to a list. The numberedInput variable is 1 if the first column is just enumeration of the in-data, and 0 otherwise. In that case, we do not want to store it as data. The part [numberedInput:] splices the resulting list of data in such a way that the first column is skipped if the input is numbered. This is done for the signal-data and for the normalization data. This reading and converting of data is also clocked and progress messages are printed.

Next, some parameters for the fit are defined: initial guess, bounds on beta, and tolerance. Now, if "use_scipy_for_initial_guess" is set to True, the Scipy optimizer is called with a relatively high tolerance. Its result is then used as the starting guess for Minuit.migrad, which is called to finish the optimization. Then, Minuit.hesse is called to estimate the errors. Finally, results are displayed as the estimates for the parameters beta $\pm\sigma$ as well as the covariance matrix. This concludes the main function and the entire Python program. The time taken is displayed.

6 Results

A sample output of the code can be seen in the Appendix. All results presented are run with the following specifications:

- run on 2021-05-14 version of code
  (commit "be3713b643f215a0d7cc174094d95f1aeaa40d33" on GitHub)
- Python 3.9.0 32-bit
- Processor Intel Pentium CPU 4405U @ 2.10GHz 2.11 GHz, 64-bit
- Windows 10 Education, 20H2, OS-version 19042.928

All time measurements were performed consecutively in order to limit the impact of potential background processes or other variables. The performance of the code was quantified by analyzing execution time, consistency and bias. The program configuration used are as specified in Table 1. A starting guess close to the true value was selected to slightly speed up the execution time during testing and to ensure that error estimation was valid also in this case.

Table 1: Program settings used when running tests.

Parameter	Value
Input for MC-generated test-data (alpha, dPhi, alpha1, alpha2)	(0.461, 0.740, 0.754, -0.754)
Initial guess: (alpha, dPhi, alpha1, alpha2)	(0.46, 0.7, 0.7, -0.7)
Tolerance (for `Scipy`)	10e-3
Tolerance (for `Minuit`)	Default
Bounds (for `Scipy`) (lower, upper)	((-1,1),(-PI,PI),(-1,1),(-1,1))
Input-data filetype	.txt

6.1 Time Analysis

For time analysis, only the `iminuit`-solver was used. Time per iteration was measured since it makes the calculation less dependent on the start values; an initial guess far away would increase the number of iterations rather than the time per iteration. Since the execution time depends on a lot of factors, e.g. the type of machine used and active background processes, the results should be interpreted qualitatively rather than quantitatively. With the setup described by Table 1, the 5D program took about 90 iterations to estimate the parameters and about another 20 to estimate the errors. This varied only little for different sizes of the input. The C++-program requires around 300 iterations to conduct the same estimation.

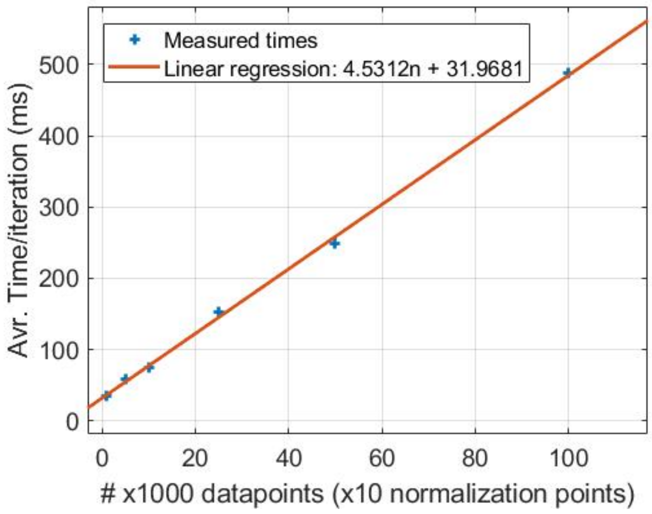


Figure 8: Measured time per iteration for varying sizes of input data, with `Numba`. Data used for plot can be found in Appendix B.1.

As can be seen in Figure 8, given the aforementioned running conditions, an iteration on average takes about 4.50 ms per thousand data points, using 10 times as many normalization points. The first iteration in each run in general takes significantly longer since `Numba` has to compile the `@JIT`ted functions.

We can also compare the execution time per iteration for various data sizes, similarly to Figure 8, without `Numba`. This comparison can give a rough idea of the impact of `Numba`.

It should be noted that this comparison is not entirely fair, since the time measurements in Figure 9 were performed by commenting out all `@JIT`-decorators. This means the tested code is still adapted to be compatible with `Numba`, e.g. through extensive use of for-loops. Without `Numba`, this is not the fastest way to run Python, though it is an option. For example using `Numpy` vectorization could likely speed execution up by about one or even two orders of magnitude, though not much more than that.

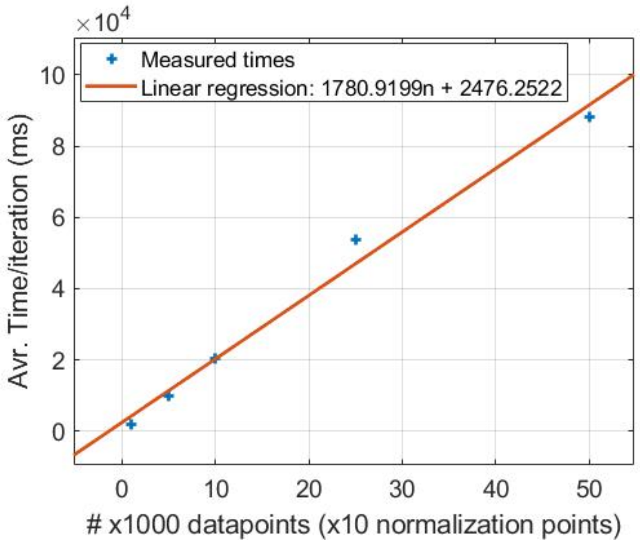


Figure 9: Measured time per iteration for varying sizes of input data, without `Numba`. Data used for plot can be found in Appendix B.1.

We find that an iteration on average takes about 1800 ms per thousand data points. This means that adding the `@JIT` decorators results in a reduction of the execution time by a factor of roughly 400. To save time when measuring the very slow non-`Numba` code, the program was stopped after a few iterations when the average time per iteration appeared to stabilize. Since the most time consuming part is the numerical calculations and not in the optimizer itself, this should be a good enough approximation.

On average, when running the program with `Numba` using 100k data points and 1000k normalization points, the execution time was around 90 seconds. About 10 seconds out of 90 are used for reading the input data. Input time can be decreased by about a factor 10 by using .root-files instead of .txt-files as input, though these files generally require more setup for the program to run. It is also worth noting that the main time sink is the MC-normalization described in Algorithm 1, due to the large data sizes needed. In particular the repeated function evaluation of $\mathcal{W}$ is very time consuming. Using a 10 times larger set of normalization data, MC-integration also takes about 10 times longer than the log-sum described in Algorithm 2.

It is somewhat complicated to accurately compare the 1D, 2D and 5D implementations, since they have some different methods and are at different levels of maturity. The general trend observed when debugging though was a slight increase in time per iteration and more prominently an increase in iterations needed to find a solution.

6.2 Statistical Properties

In this section, the numerical performance of the program is analyzed. This is done by evaluating bias and consistency, as described in section 3.2. Both measurements were done by splicing the testing data into smaller subsets and process each subset with the program. In all cases, the full set of normalization data (one million points) was used.

6.2.1 Consistency

Consistency was measured by fitting different subsets of the test data and noting their estimated values and standard deviations. Each data size was run once.

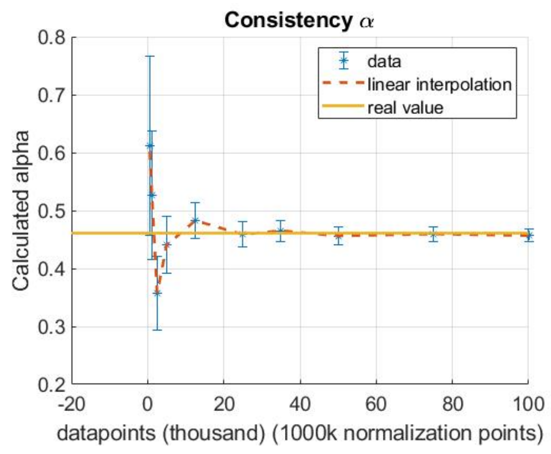
Table 2: Estimated parameter values and their standard deviation for various sizes of input data. 1000k normalization points in each case. The table presents a subset of the data used to create plots in Figure 10. Both Python and C++/`ROOT` results are shown, for comparison.

N	α	$\Delta\Phi$ [rad]	α_1	α_2
Input (Python)	**0.461**	**0.740**	**0.754**	**-0.754**
100 000	0.457 ± 0.011	0.745 ± 0.024	0.766 ± 0.022	-0.739 ± 0.021
50 000	0.456 ± 0.015	0.772 ± 0.034	0.767 ± 0.030	-0.731 ± 0.029
25 000	0.459 ± 0.022	0.71 ± 0.05	0.72 ± 0.04	-0.79 ± 0.05
5 000	0.44 ± 0.05	0.82 ± 0.12	0.8 ± 0.1	-0.66 ± 0.08
2 500	0.36 ± 0.06	0.50 ± 0.11	0.83 ± 0.19	-0.75 ± 0.17
500	0.61 ± 0.16	0.63 ± 0.31	0.9 ± 0.5	-0.7 ± 0.4
Input (C++)	**0.461**	**0.740**	**0.754**	**-0.754**
100 000	0.457 ± 0.011	0.745 ± 0.024	0.766 ± 0.022	-0.739 ± 0.021
50 000	0.456 ± 0.015	0.772 ± 0.034	0.767 ± 0.030	-0.731 ± 0.029
25 000	0.449 ± 0.021	0.74 ± 0.05	0.74 ± 0.04	-0.78 ± 0.04

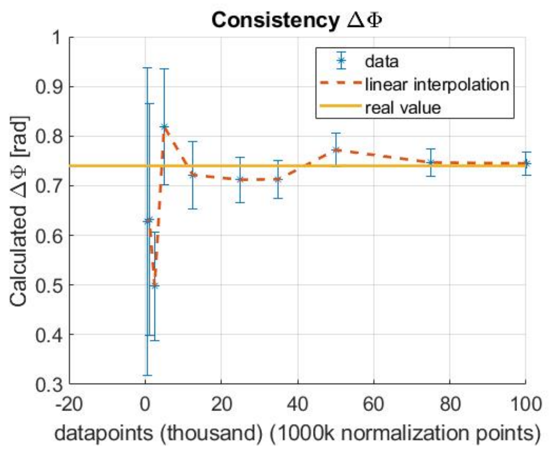
As can be seen in Figure 10, the estimated parameter value is always well within one standard deviation of the real value. We can also visually see the errors decreasing as the number of data points used increases. By the data in Table 2, the error estimation roughly follows $\frac{1}{\sqrt{N}}$, where N is the size of the data set used. This is as expected [1]. Comparing these results to ones found by the C++/`ROOT`-implementation (generated with the same input data), we see results are very similar. In fact, for large sizes of the input, they are identical. By the definition of equation 15, the estimator is found to be consistent.

6.2.2 Bias

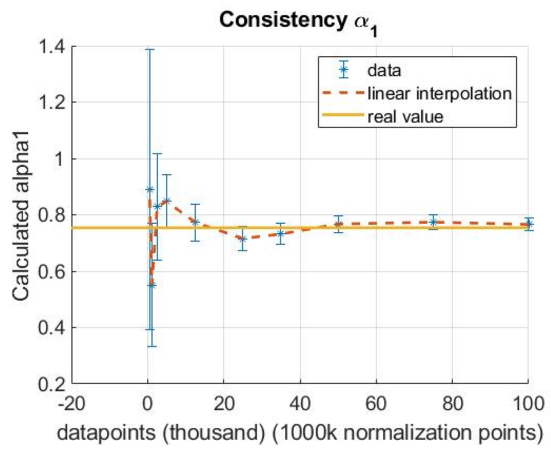
Since we can consider each subset of the generated data to be statistically independent, the expected value $E[\hat{\beta}]$ is simply the average. Ten different subsets of the test data were used for data sizes 1 000 and 10 000 points. The results, including uncertainties, are presented in Figure 11. In principle, for a proper estimate of $E[\hat{\beta}]$, infinitely many estimations and data sets should be used. However, in this study we only use 10. From Figure 11 we can still clearly see that the difference between the estimations and the true value is within one standard deviation for that set. $E[\hat{\beta}]$ appears to decrease rapidly as N



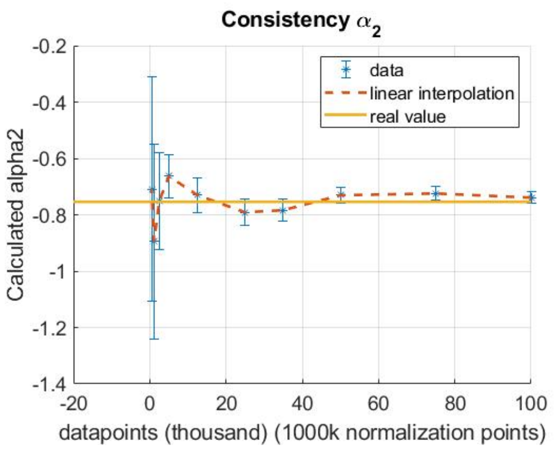
(a) Consistency of α



(b) Consistency of $\Delta\Phi$



(c) Consistency of α_1



(d) Consistency of α_2

Figure 10: Parameter and error estimates for each parameter in $\boldsymbol{\beta}$ for various data sizes. The error-bars show one standard deviation. The red line is a linear interpolation between the measured data for visual aid. The yellow line represents the value with which the tested data was generated.

increases, and for the 10k measurements, it is mostly indistinguishable from the true value in Figure 11 which indicates that $\hat{\beta}$ is asymptotically unbiased. We do expect some bias for very small sample sizes, since the MLL-estimator assumes the data is approximately Poisson-distributed which generally only holds for large sample sizes.

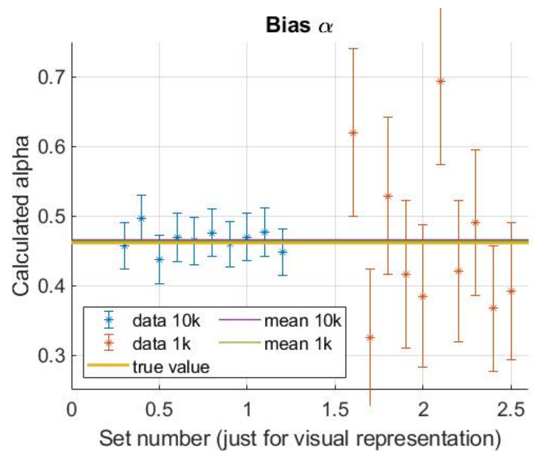
7 Discussion

In all, a functional program was produced. Its requirements are significantly simpler compared to having a C++-compiler and downloading and configuring all of `ROOT`. While the `iminuit` library is related to `ROOT`, it is not at all dependent on it and circumvents the issue of `ROOT` being large and non-modular. No significant surprises in the results can be seen.

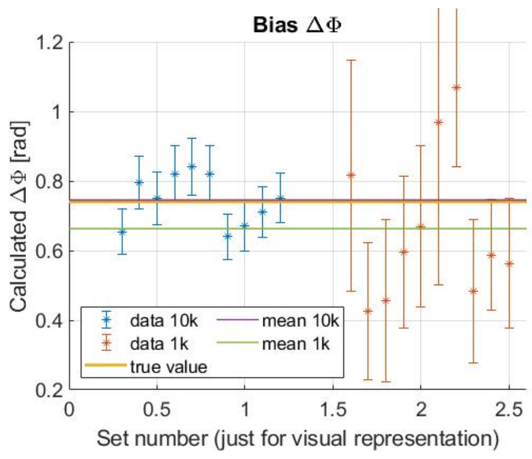
Regarding execution time; from Figure 8 and the fact that the number of iterations needed to estimate the parameters is relatively invariant, one can deduce that the time complexity of the fit is $O(n)$, i.e. linear. This is as good as it can be given that it is inevitable that each point in the data set is visited in each iteration. The time per iteration is on average 4.53 ms per thousand data points. Since there are up to 120 iterations to complete, this results in an execution time of one to a few minutes, which is acceptable. However, this also depends on other run-time variables such as the machine used and data size. For instance, when the program was tested on another PC, execution time decreased by about a factor 7. When the C++/`ROOT`-implementation was run for the same data in those same conditions, the Python-program showed to be a factor 15 faster in total. The execution time of the Python program is thus about one order of magnitude *faster* than the C++/`ROOT`-implementation, despite being written in a interpreted language. However, without `Numba`, execution time increases to several hours for large data sets, which is unfeasible.

For further improvement of the execution speed, Algorithms 1 and 2 are good candidates. Since the structure used is a simple for-loop, and the execution order of the summation does not matter, parallelization could work. In fact, `Numba` offers built-in automatic parallelization by setting optional argument `parallel = True` and replacing the for-loops with `Numba` `prange`-iterables. More information about `Numba` parallelization can be found in the `Numba` documentation [22]. Parallelization was not used in this project simply because it did not support the 32-bit Python used, but should be doable using a standard 64-bit system. This should in theory be able to decrease the execution time by a factor equal to the amount of cores available (typically 2-8), and would likely be worth implementing if very large data-sets are used.

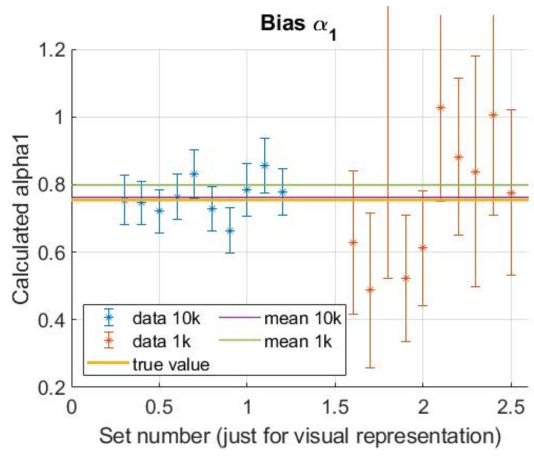
Regarding the statistical properties of the program, it is both consistent and asymptotically unbiased, as concluded in section 6.2. The consistency, presented in Table 2 (more detailed in Appendix B.2), shows standard deviations are pretty much identical between the Python and the C++-versions. Neither the C++ nor the Python implementation shows any bias. Considering these results, the program is considered to be ready to analyze new data in a more efficient and user friendly way.



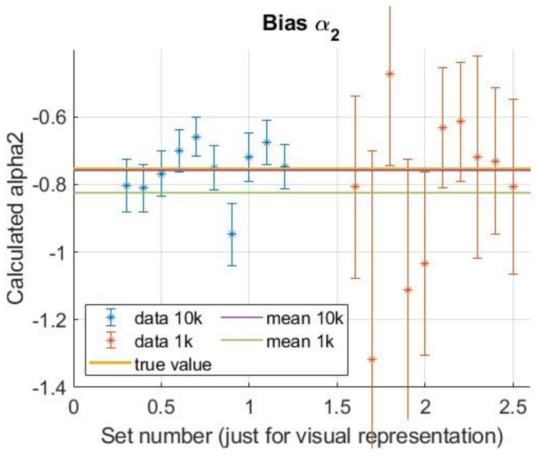
(a) Bias estimate of α



(b) Bias estimate of $\Delta\Phi$



(c) Bias estimate of α_1



(d) Bias estimate of α_2

Figure 11: Parameter and error estimates for each parameter in $\boldsymbol{\beta}$ for data sizes 1k and 10k, ten runs each. The x-axis is just for visual representation and holds no meaning apart from separating runs visually. Error bars represent one standard deviation. The yellow line represents the input value used in the event generator. The purple line shows the average value of the 10k (blue) measurements, and the green line likewise for 1k (red). The purple line is sometimes partially coinciding with the yellow line.

8 Summary

A fitting program in Python has been developed and its statistical properties analyzed. Tying back to the goals of this thesis defined in section 1.3, the program should:

- Be able to make a multidimensional parameter estimation.

The program runs and accepts 5D-data as input, both a signal and normalization data. It returns a fit for the parameters according to the formalism described in section 2. The 1D as well as the 2D formalism implementations were also developed along the way, in varying levels of maturity. For example, the 1D fit did not feature any Monte-Carlo integration and no `Numba`. This is because the 1D case is so simple that the normalisation can be carried out analytically. The 2D-fit still only uses `scipy.optimize.minimize` for optimization, thus not yielding a usable error estimation. However, all versions of the program run smoothly and return parameter estimations.

- Be easy to setup and run.

To run the 5D-implementation requires:

- Python 3.6 or later.
- `Numpy`, a standard library for anyone conducting scientific computing in Python.
- `Scipy`, for optimizer. Not necessary if only using 'good' starting guesses. Also standard.
- `iminuit`, for parameter and error estimates. Common framework within particle physics.
- `Numba`, for speed. Not mandatory, but ignoring it will slow down execution significantly. This framework appears to be growing in popularity but can be a little tricky to satisfy. Runs without modification.
- Access to data and specifying a path to it in the beginning of the program.

All libraries are also easy to install for anyone who has installed Python libraries before, using `pip`. Once done, all that needs to be done is make sure the program has files for the input data to read and it is ready to run.

- Have a similar numerical performance compared to the current C++/`ROOT` program. That means error estimates and a covariance matrix as well as similar errors for similar data sizes.

The fit returns a covariance matrix and error estimation, and the errors are similar to the C++/`ROOT` program. From Table 2 and Figure 10 we can conclude that the fit is consistent and asymptotically unbiased.

- Be sufficiently fast to analyse realistically sized data samples within reasonable time.

Execution time on an average computer for a data set sized in the millions is well below a day, which should be sufficient for any application. Slightly improving the machine running the code will also result in execution-times closer to a few minutes for any realistically sized data-set. Additionally, large data-sets will mostly be run only a few times. For iterative testing and modification with smaller data-sets: around one minute. This is definitely acceptable. It is also about an order of magnitude faster than the C++/`ROOT`-implementation, so either way it is an improvement.

- Be easy to read, maintain and modify.

The entire program is written in pure Python, mostly using the simple syntax of for-loops. While more ”exotic” techniques such as function decorators and `Numba` are used, one typically will not need to edit these in order to modify the code. `Numba` also requires a somewhat stricter syntax, though this only affects certain parts. However, as long as the input data is converted to `Numpy`-arrays, the stricter syntax should not be an issue for any modifications. The worst case scenario is that `Numba` is not able to optimize the code and in this case it will typically fall back on the slower version.

Using the program to estimate parameters of other spin 1/2-hyperons is as simple as changing the input data. It can also be modified to act as a general MLL-estimator by simply changing the function $\mathcal{W}$. All libraries used are in stable versions, though all are continuously being developed further. In the future, they might lose backwards compatibility, but since they will likely only change syntax rather than reduce usability, this should be maintainable without too much effort, and likely none.

Considering this, the specifications of the program are essentially met. While the execution time can still be improved and there likely are more optimizations of the code to be made, it is well within acceptable bounds. Numerical performance is not identical to the current implementation, but it passes essential statistical tests and qualitatively as well as quantitatively performs similarly to the C++/`ROOT`-implementation. In addition to being written in Python, the code is well commented which should further simplify its future use.

Due to covid-19 restrictions, most work was conducted from home on my personal laptop. Only occasionally did I work from the office. Since I did not have much previous experience with C++, did not have or install `ROOT` on my laptop and the C++/`ROOT`-implementation was somewhat hard to read for me, the Python program was in most ways developed independently from it. Obviously, methods to replace `ROOT` had to be found, but even the rest of the code was written mostly independently from the previous implementation. Only when it came to testing performance began serious comparison between the two implementations to ensure no quality was lost. Due to the maximum log-likelihood being a well established method for parameter estimation, the general structure of the code ended up fairly similar anyway.

9 Conclusion and Outlook

This project has demonstrated clear benefits over the existing C++ and `ROOT` frameworks in being more user friendly while performing similarly regarding statistical properties. We benefit from several existing Python packages, similar to those in the previous C++

implementation. Some of the main hurdles of scientific computing in Python, such as execution time, are being overcome by projects such as `Numba`. As the trend of Python becoming more and more common in scientific data analysis continues, these libraries continue to grow in number and maturity.

The program developed during this project can hopefully be used by other research groups and lay the foundation for further analysis of data using the general spin 1/2 hyperon decay formalism. While the example of $\Lambda\overline{\Lambda}$-decays has been covered here, the program can just as well be used to study the decays of other hyperons such as Σ^+ and $\Lambda_c^+ \to pK_s$, as well as the first decay of Ξ^- and Ξ^0 hyperons. In addition, with the extended formalism in [15] and [3] it can be used for complete studies of Σ^0, Ξ^0/Ξ^- and Ω^- hyperons. Another possible future use is for hyperon production from proton-antiproton annihilations, such as at the PANDA-experiment being built in Germany.

MLL being a more generally powerful and significant tool in parameter estimation in all fields of science combined with the ease-of-use and the fact that many universities now teach Python might also lay the foundation for further student use. Furthermore, considering that a user previously was required to have a deeper understanding of the formalism, C++ and computers in general just to be able to run this parameter estimation, this new tool opens up for a stronger physics projects in future thesis projects. This will allow students and teachers with less specific training to make contributions and learn by doing.

In the long term, the hope is that this program and the science it is based on will facilitate further research in this field. Whether these studies will explain the origins of our universe's very existence or conclude that this is probably not the path which will lead to scientific breakthroughs, it will map previously uncharted territory.

Populärvetenskaplig sammanfattning

Frågan om universums existens gäckar fysiker såväl som icke-fysiker än idag. Ur en fysikalisk synpunkt finns det grundläggande principer som antyder att universum borde ha förgåtts redan strax efter Big Bang; Uppenbarligen är så inte fallet. Ett fundamentalt antagande är att materia och antimateria skapades i lika mängder i Big Bang, och om en viss typ av symmetri funnits mellan dessa två borde vi observera lika mycket materia som antimateria i universum. De borde då förinta varandra (annihilera) vid kontakt, vilket som vi vet inte fallet. Därför söker nu forskare efter tecken på att denna fundamentala symmetri kan brytas. Det är mätdata från denna typ av asymmetrijakt som detta projekt ska anpassa för att få fram indikativa parametrar som kan ge svar på om denna symmetri uppvisas eller inte. Syftet med projektet är att detta anpassningsprogram ska vara enkelt att använda och modifiera, så att sökandet ska bli enklare och i framtiden användas till att kunna ge svar på dessa fundamentala frågor.

References

[1] G. Cowan, *Statistical Data Analysis* (Oxford University Press, New York, 1998), ISBN 0-19-850155-2.

[2] G. Fäldt, A. Kupsc, *Phys. Let.* **B772**, p. 16-20 (2017)

[3] E. Perotti, G. Fäldt, A. Kupsc, S. Leupold, & J. J. Song, *Phys. Rev.* **D99**, 056008 (2019)

[4] M. Gell-Mann, *Phys. Let.* **Vol 8, issue 3**, p. 214-215 (1964)

[5] N. Cabibbo, *Phys. Rev. Lett.* **10**, 531 (1963); M. Kobayashi, T. Maskawa, *Prog. Th. Phys.* **49**, 652 (1973)

[6] A. D. Sakharov, *Pisma Zh. Eksp. Teor. Phys. Fiz.* **5**, 32 (1967)

[7] A. Heikkilä, *Studies of Light Hyperon Decay Parameters*, BSc Thesis, Uppsala University (2019).

[8] minutephysics YouTube-channel, *How to Tell Matter From Antimatter* (2020), [Online, video, viewed 2021-05-04]

[9] G. Aad *et al.*, *Phys. Let.* **B716**, p. 1-29 (2012)

[10] W. Bernreuther, *Lect. Notes Phys.* **591**, 237 (2002); L. Canetti *et al.*, *New J. Phys.* **14**, 095012 (2012)

[11] Wikipedia contributors, Standard Model - Wikipedia, The Free Encyclopedia (2021). [Online, image, accessed 2021-05-14].

[12] P.A. Zyla *et al.* (Particle Data Group), Prog. Theor. Exp. Phys. **2020**, 083C01 (2020)

[13] BESIII Collaboration, *Nature Phys 15*, p. 631–634 (2019)

[14] BESIII Collaboration, *Phys. Rev. Lett.* **123 12**, 122003 (2019).

[15] G. Fäldt, K. Schönning, *Phys Rev* **D 101**, 033002 (2020)]

[16] The SciPy Community, NumPy Documentation (2021). [Online, accessed 2021-05-14].
`https://numpy.org/doc/stable/`

[17] PyPi, numpy 1.20.3 (2021). [Online, accessed 2021-05-14].
`https://pypi.org/project/numpy/#files`

[18] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.

[19] SciPy developers, SciPy documentation (2021). [Online, accessed 2021-05-14].
`https://www.scipy.org/`

[20] Piti Ongmongkolkul and the iminuit team, iminuit documentation (2021). [Online, accessed 2021-05-14].
`https://iminuit.readthedocs.io/en/stable/`

[21] The pandas development team, *Zenodo*, **10.5281**/zenodo.4681666 (2021)

[22] Anaconda, Numba documentation (2021). [Online, accessed 2021-05-14].
`http://numba.pydata.org/`

[23] The PyPy Team, PyPy (2021). [Online, accessed 2021-06-06].
`pypy.org`

[24] J. Pivarski, Princeton University, PyHEP Numba Tutorial or Getting Numba to Do What You Want (2021), presentation.
`https://indico.cern.ch/event/985350/`

A Appendix: The Program

A.1 Code

All code and related project-files can be accessed freely on GitHub in repository ”MLL-python”:
https://github.com/benjaminverbeek/MLL-python

Final 5D program can be found here:
https://github.com/benjaminverbeek/MLL-python/blob/main/5D/MLL_5D_combined.py

The code per 2021-05-25, commit 014a47f97fbe754103c828829f81bf8c89c511ac follows below (wrapped lines indicated by an arrow):

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Program to make 5D fit (Double-tag BESIII)            #
# Benjamin Verbeek, updated 2021-05-14                  #
# Now using iminuit for fit, gives proper variance.     #
# Theory definitions specified in appropriate places.   #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# NOTE: Takes about twice as long as only using minuit/scipy

##### IMPORTS #####
# Imports necessary modules
from math import pi as PI       # for pi, built-in Python
import time                     # for timing, built-in Python
import numpy as np              # requires download, e.g. "$ pip3 install
↪  numpy". For scipy and efficient funcs.
from numpy import sin, cos, arccos
from scipy import optimize      # requires download, e.g. "$ pip3 install
↪  scipy". For optimization of LL.
from numba import jit           # requires download, e.g. "$ pip3 install
↪  numba". For efficient execution.
from iminuit import Minuit      # requires download, e.g. "$ pip3 install
↪  iminuit". For optimization of LL.
##### END IMPORTS #####

print(f'{" RUNNING MAX LOG LIKELIHOOD FIT ":-^60}')
##### FIT PARAMETERS FOR ANALYSIS #####
dataFrom = 0
dataTo   = 100_000 + 1
normFrom = 0
normTo = 1000_000 + 1
```

```
dispIterInfo = False    # if set to True, prints info about each iteration (LL,
↪  time, Norm). False: just a loading bar.
use_scipy_for_initial_guess = False  # set to True if the initial guess is bad.

# Set some parameters for the fit.
angleDistributionData_filename = "mcsig100k_JPsi_LLbar.dat"  # specify path if
↪  not in same folder.
normalizationData_filename = "mcphsp1000k_JPsi_LLbar.dat"
numberedInput = True         # Is one column of the input data just numbering?
↪  Specify that here.

# Do not change:
numberedInput = int(numberedInput)  # True - 1, False - 0. This is how many
↪  colums to skip in indata files. Can be specified manually further down.
nIter=0 # counts iterations
totalTime = [] # for timing per iteration. For analysis.

###### THEORY ######    (can be swapped out for whatever )
# Theory from http://uu.diva-portal.org/smash/get/diva2:1306373/FULLTEXT01.pdf ,
↪  same as ROOT-implementation.
# here alpha = eta = alpha_psi
# C_n,m take theta (no subindex), delta-phi and alpha as input. Only using
↪  nonzero ones.
# C defined as per equation 23.
@jit(nopython=True)
def C00(alpha, dPhi, th): return  2*(1 + alpha * cos(th)**2)

@jit(nopython=True)
def C02(alpha, dPhi, th): return  2*(1-alpha**2)**0.5 *
↪  sin(th)*cos(th)*sin(dPhi)

@jit(nopython=True)
def C11(alpha, dPhi, th): return  2*sin(th)**2

@jit(nopython=True)
def C13(alpha, dPhi, th): return  2*(1-alpha**2)**0.5 *
↪  sin(th)*cos(th)*cos(dPhi)

@jit(nopython=True)
def C20(alpha, dPhi, th): return  -1*C02(alpha,dPhi,th)

```

```python
@jit(nopython=True)
def C22(alpha, dPhi, th): return  alpha*C11(alpha,dPhi,th)

@jit(nopython=True)
def C31(alpha, dPhi, th): return  -1*C13(alpha,dPhi,th)

@jit(nopython=True)
def C33(alpha, dPhi, th): return -2*(alpha + cos(th)**2)
##########
# a-funcs should have alpha1/2, th1/2, ph1/2 index depending on lambda (1) or
↪  lambda-bar (2).
# Only defining used ones. Defined as per equation 50.
@jit(nopython=True)
def a00(alpha, th, ph): return 1

@jit(nopython=True)
def a10(alpha, th, ph): return alpha * cos(ph) * sin(th)

@jit(nopython=True)
def a20(alpha, th, ph): return alpha * sin(th) * sin(ph)

@jit(nopython=True)
def a30(alpha, th, ph): return alpha * cos(th)
##########
# Defined as per equation 54.
@jit(nopython=True)
def WDoubleTag(alpha, dPhi, alpha1, alpha2, th, th1, ph1, th2, ph2):
    return C00(alpha, dPhi, th)/2 * a00(alpha1, th1, ph1) * a00(alpha2, th2,
    ↪  ph2) + \
    C02(alpha, dPhi, th)/2 * a00(alpha1, th1, ph1) * a20(alpha2, th2, ph2) + \
    C11(alpha, dPhi, th)/2 * a10(alpha1, th1, ph1) * a10(alpha2, th2, ph2) + \
    C13(alpha, dPhi, th)/2 * a10(alpha1, th1, ph1) * a30(alpha2, th2, ph2) + \
    C20(alpha, dPhi, th)/2 * a20(alpha1, th1, ph1) * a00(alpha2, th2, ph2) + \
    C22(alpha, dPhi, th)/2 * a20(alpha1, th1, ph1) * a20(alpha2, th2, ph2) + \
    C31(alpha, dPhi, th)/2 * a30(alpha1, th1, ph1) * a10(alpha2, th2, ph2) + \
    C33(alpha, dPhi, th)/2 * a30(alpha1, th1, ph1) * a30(alpha2, th2, ph2)
##### END THEORY #####


##### MC INTEGRATOR #####
# MC-integrator for normalization factors
```

```python
@jit(nopython=True) # numba decorator. Significantly improves performance (~factor 100)
def MCintegral(alpha,dPhi,alpha1,alpha2, uniformAngles, distributionFunc):
    """Monte Carlo integration for normalization, for given parameters, a set of normalization angles and a distributionFunc."""
    s = 0.0   # sum
    n = 0.0   # number of points
    for xi in uniformAngles: # xi is a 5D list here
        th,th1,ph1,th2,ph2 = xi
        s += distributionFunc(alpha,dPhi,alpha1,alpha2 , th,th1,ph1,th2,ph2) # evaluate W at a bunch of random points and sum.
        n += 1  # count number of points. Could also use len(uniformAngles)
    return 1/n * s #* (2**3 * (2*PI)**2)    # MC-integral: average value of function, technically multiplied by area (2**3 * (2*PI)**2)
                    # this does not affect results however, since it just becomes adding a constant to the LL-function.
##### END MC INTEGRATOR #####

##### NEG LOG LIKELIHOOD FUNCTION #####
# Help function, split out so numba can optimize it.
@jit(nopython=True)
def iterativeLL(par, var, pdf):
    s = 0  # sum
    alpha,dPhi,alpha1,alpha2 = par
    for v in var: # iterate over samples
        th,th1,ph1,th2,ph2 = v  # unpack angles. Cannot use *v for numba compatibility.
        s -= np.log(pdf(alpha,dPhi,alpha1,alpha2 , th,th1,ph1,th2,ph2)) # log-sum of pdf gives LL. Negative so we minimize.
    return s

# Note that this is unjitted: It would barely improve performance and would reduce readability/generalizability by a lot.
def negLLMinuit(par):
    """A minuit modified negLL function. Only takes in parameters to optimize.
    It is vital that global variables xi_set and normAngs are defined containing data on the correct format."""
    normSep = True
    global xi_set       # bit ugly. Requiered for Minuit fit (function to optimize only dependent on parameters to optimize for)
    global normAngs     # WARNING: Global vairable
    var = xi_set
    pdf = WDoubleTag
    normAngs = normAngs

    t1 = time.time()    # time iterations

    if dispIterInfo:
        print("--------")
    else:
        global nIter    # WARNING: Global variable
        shapes = ['-', '~', '=', '+', '*', ':']
        maxWidth = 50 + 1
        nIter = nIter + 1
        print(shapes[(nIter//maxWidth)%len(shapes)]*(nIter%maxWidth) + f">
        ↪  {nIter} ", end="\r")    # Loading bar.

    if normSep==True:   # Structured this way, a finished PDF can be used too.
    ↪  Just set normSep=False
        normalization = MCintegral(*par, normAngs, pdf)
        t2 = time.time()
        if dispIterInfo: print(f"One normalization done... took {t2 - t1:.5f}
        ↪  seconds. \t   Norm:  {normalization}")
    else:
        normalization = 1   # nothing happens. log(1) = 0.
    # Calculate LL-sum and add normalization
    r = iterativeLL(par,var, pdf) + len(var)*np.log(normalization) # normalize
    ↪  after; -log(W_i/norm) = -log(W_i) + log(norm)
    t3 = time.time()
    if dispIterInfo:
        print(f"One LL-sum done. Took {t3 - t2:.5f} seconds. \t\t\t neg LL:
        ↪  {r}")    # takes a long time but not AS long as normalization.
        print(f"Total time for one iteration was {t3 - t1:.5f} seconds.")
    totalTime.append(t3 - t1)
    return r
##### END NEG LOG LIKELIHOOD FUNCTION #####
############# END HELP FUNCTIONS #############

########## MAIN: ##########
def main():
    start_time = time.time()
    print("Reading input data... \t (this might take a minute)")
    ########## READ DATA: ##########
    # Read angle distribution data, save in numpy-array (needed for numba compatibility & speed)
    global xi_set   # Global variable for access in negLL
    xi_set = [ list(map(float,i.split()))[numberedInput:] for i in open(angleDistributionData_filename).readlines() ]    # list (of lists)
    # Iterate thru lines of datafile, for each line, split it into list of number contents, map the content of that list from
    # str -> float, convert map object -> list, skip first if it is numbered input, all in list comprehension.
    xi_set = np.asarray(xi_set) # converts to numpy.array. Much faster than numba typed list.
    xi_set = xi_set[dataFrom:dataTo]    # for analysis
    print(f"First row: {xi_set[0]}")    # sanity-check data
    print(f"Size of signal set: {len(xi_set)}")
    print("Finished reading signal data.")
    t2 = time.time()
    print(f'{f" {(t2 - start_time):.3f} seconds ":-^60}')

    # Read normalization data
    print("Reading normalization data...")
    global normAngs     # Global variable for access in negLL
    normAngs = [ list(map(float,i.split()))[numberedInput:] for i in open(normalizationData_filename).readlines() ]    # list (of lists)
    normAngs = np.asarray(normAngs) # needed for numba. Fixed datatype.
    normAngs = normAngs[normFrom:normTo]    # for analysis
    print(f"First row: {normAngs[0]}")      # sanity-check data
    print(f"Number of points for normalization: {len(normAngs)}")
    print(f'{f" {(time.time() - t2):.3f} seconds ":-^60}')
    print(f'{f" {(time.time() - start_time):.3f} seconds total for all input data ":-^60}')
    ########## END READ DATA ##########

    ########## OPTIMIZE WITH MINUIT (and maybe scipy) ##########
    #initGuess = (0.461, 0.740, 0.754, -0.754)  # expected results
    initGuess = (0.46, 0.7, 0.7, -0.7)
    bnds = ((-1,1),(-PI,PI),(-1,1),(-1,1))   # bounds on variables (needed for scipy)
    # Options for the optimizer. Can also fix method. Read more: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html
    ftol = 10**-3
    varNames = ('alpha', 'dPhi', 'alpha_1', 'alpha_2')
    if use_scipy_for_initial_guess == True:
        print(f"\nOptimizing for starting guess with scipy
        ↪ optimize.minimize\ninitial guess: {initGuess}")
        # scipy existing minimizing function.
        res = optimize.minimize(negLLMinuit, initGuess, bounds=bnds, tol=ftol)
        ↪ # Scipy optimization for starting guess for minuit
        print()
        print(f"Optimizing with minuit\ninitial guess: {res.x}")
        m = Minuit(negLLMinuit, res.x, name=varNames)  # define minuit function
        ↪ and initial guess
    else:
        print(f"Optimizing with minuit\ninitial guess: {initGuess}")
        m = Minuit(negLLMinuit, initGuess, name=varNames) # define minuit
        ↪ function and initial guess
    m.errordef = Minuit.LIKELIHOOD      # important
    m.migrad()  # run minuit optimziation
    print()
    print(f"Finding errors with minuit")
    m.hesse()   # run covariance estimator
    print() # offset /r from loading bar.
    print(f"Valid optimization: {m.valid}")  # was the optimization successful?
    print("Parameter estimation:")
    for var, val, err in zip(varNames, m.values, m.errors):
        print(f"{var:>10}: {val:>15.10f} ± {err:.10f}")
    print("Covariance matrix:")
    print(m.covariance)
    print(f'Covariance matrix accurate: {m.accurate}')

    # For easy analysis.
    '''
    print(f"\nFor analysis:          Data: {dataFrom}-{dataTo-1}       Norm:
↪   {normFrom}-{normTo-1}")
    print(f"Initial guess: {initGuess}")
    if use_scipy_for_initial_guess:
        print(f"Minuit initial guess (from scipy): {res.x}")
    resVals = []
    resErrs = []
    for val, err in zip(m.values, m.errors):
        resVals.append(str(val))
        resErrs.append(str(err))
```

```python
    #print(', '.join(resVals))   # easy to copy-paste to excel.
    #print(', '.join(resErrs))
    print(', '.join(resVals + resErrs))
    print(dataFrom-dataTo)
    print(sum(totalTime)/len(totalTime))
    print("Done.")
    '''
    ########## END OPTIMIZE WITH MINUIT ##########
########## END MAIN ##########

if __name__ == "__main__":  # doesn't run if imported.
    t0 = time.time()
    main()
    print(f"--------- TOOK A TOTAL OF {time.time() - t0:.3f} SECONDS ---------")
```

A.2 Sample Output

Sample output using Scipy for initial guess. (run 2021-04-15 version) (computer specifications: Python 3.9.0 32-bit, processor Intel(R) Pentium(R) CPU 4405U @ 2.10GHz 2.11 GHz, 64-bit. Windows 10 Education, 20H2, OS-version 19042.928.)

```
# -------------- RUNNING MAX LOG LIKELIHOOD FIT --------------
# Reading input data...     (this might take a minute)
# First row: [ 0.0321418  2.31065    2.80985    2.02057   -1.68605  ]
# Size of signal set: 100000
# Finished reading signal data.
# ---------------------- 0.793 seconds -----------------------
# Reading normalization data...
# First row: [ 0.0321418  2.31065    2.80985    2.02057   -1.68605  ]
# Number of points for normalization: 1000000
# ---------------------- 8.077 seconds -----------------------
# ---------- 8.871 seconds total for all input data ----------

# Optimizing for starting guess with scipy optimize.minimize
# initial guess: (0.46, 0.7, 0.7, -0.7)
# -----------------------------------> 35
# Optimizing with minuit
# initial guess: [ 0.4568892   0.74740629  0.75525831 -0.74799768]
# ===========> 113 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~> 101
# Finding errors with minuit
# ==================================> 136
# Valid optimization: True
# Parameter estimation:
#      alpha:    0.4567725392 ± 0.0108241175
#       dPhi:    0.7448948983 ± 0.0235477417
#    alpha_1:    0.7659180877 ± 0.0216926385
#    alpha_2:   -0.7392833336 ± 0.0210097171
# Covariance matrix:
# -----------------------------------------------------
# |         |     alpha      dPhi   alpha_1   alpha_2 |
# ----------|-----------------------------------------|
# |   alpha |  0.000117     7e-05 -1.13e-05  1.33e-05 |
# |    dPhi |     7e-05  0.000554    -5e-05  1.82e-05 |
# | alpha_1 | -1.13e-05    -5e-05  0.000471  0.000394 |
# | alpha_2 |  1.33e-05  1.82e-05  0.000394  0.000441 |
# -----------------------------------------------------
# Covariance matrix accurate: True
# ------ TOOK A TOTAL OF 71.218 SECONDS ------
```

B Appendix: Measurements Performance Analysis

B.1 Time

Table 3: Data used to make plots in figure 8 and 9.

N	N (normalize)	time/iter (s), with Numba	time/iter (s), without Numba
1000	10000	0.03469	2.054649962
5000	50000	0.05875	9.79479047
10000	100000	0.0742	20.45531472
25000	250000	0.1529	53.89932299
50000	500000	0.2483353925	88.24089575
100000	1000000	0.4883890447	-

B.2 Consistency

Table 4: Data used for consistency analysis and plot 10.

index		**N**	**alpha**	**dPhi**	**alpha1**	**alpha2**	**error**	**1 sigma**		
from	to	**input:**	**0.461**	**0.74**	**0.754**	**-0.754**				
0	100000	100000	0.4568	0.7449	0.7659	-0.7393	0.0108	0.0235	0.0217	0.0210
25000	100000	75000	0.4596	0.7467	0.7739	-0.7243	0.0125	0.0274	0.0256	0.0240
0	50000	50000	0.4563	0.7721	0.7673	-0.7307	0.0154	0.0343	0.0300	0.0288
50000	85000	35000	0.4652	0.7130	0.7318	-0.7836	0.0183	0.0388	0.0365	0.0388
50000	75000	25000	0.4586	0.7121	0.7157	-0.7912	0.0216	0.0459	0.0426	0.0467
75000	87500	12500	0.4827	0.7216	0.7733	-0.7304	0.0313	0.0676	0.0651	0.0616
87500	92500	5000	0.4403	0.8189	0.8469	-0.6633	0.0492	0.1171	0.0955	0.0767
92500	95000	2500	0.3575	0.4979	0.8286	-0.7517	0.0630	0.1090	0.1881	0.1709
95000	96125	1125	0.5260	0.6311	0.5506	-0.8951	0.1108	0.2332	0.2187	0.3473
96125	96625	500	0.6118	0.6276	0.8891	-0.7088	0.1552	0.3104	0.4965	0.3975

B.3 Bias

Table 5: Calculated estimations for 10 sets of 1k data. Used to plot figure 11.

index		**N**	**alpha**	**dPhi**	**alpha1**	**alpha2**	error	**1 sigma**		
from	**to**	**input:**	**0.461**	**0.74**	**0.754**	**-0.754**	-	-	-	-
0	1000	1000	0.61933	0.81528	0.62765	-0.80677	0.12042	0.33061	0.21239	0.26964
1000	2000	1000	0.32502	0.42563	0.48683	-1.31621	0.09862	0.19709	0.23023	0.61410
2000	3000	1000	0.52867	0.45624	1.20834	-0.47279	0.11233	0.23399	0.68437	0.27240
3000	4000	1000	0.41611	0.59572	0.52342	-1.11116	0.10642	0.21858	0.18697	0.38505
4000	5000	1000	0.38466	0.66967	0.61175	-1.03354	0.10260	0.23149	0.16895	0.27120
5000	6000	1000	0.69284	0.96670	1.02576	-0.63297	0.11910	0.46435	0.27632	0.17795
6000	7000	1000	0.42074	1.06740	0.88184	-0.61398	0.10121	0.22776	0.23122	0.17601
7000	8000	1000	0.49014	0.48419	0.83719	-0.72046	0.10527	0.20628	0.34107	0.29910
8000	9000	1000	0.36715	0.58680	1.00432	-0.73028	0.09027	0.15928	0.29648	0.21770
9000	10000	1000	0.39150	0.56362	0.77614	-0.80617	0.09816	0.18524	0.24580	0.25958
		Average:	0.46362	0.66312	0.79832	-0.82443				

Table 6: Calculated estimations for 10 sets of 10k data. Used to plot figure 11.

index		**N**	**alpha**	**dPhi**	**alpha1**	**alpha2**	**error**	**1 sigma**		
from	**to**	**input:**	**0.461**	**0.74**	**0.754**	**-0.754**	**-**	**-**	**-**	**-**
0	10000	10000	0.45741	0.65430	0.75415	-0.80241	0.03328	0.06570	0.07313	0.07818
10000	20000	10000	0.49630	0.79514	0.74558	-0.81067	0.03361	0.07565	0.06390	0.06909
20000	30000	10000	0.43713	0.75097	0.72100	-0.76780	0.03411	0.07557	0.06418	0.06819
30000	40000	10000	0.46910	0.82024	0.76302	-0.69958	0.03466	0.08051	0.06762	0.06263
40000	50000	10000	0.46384	0.84189	0.83054	-0.65911	0.03422	0.08156	0.07055	0.05728
50000	60000	10000	0.47570	0.81999	0.72723	-0.75080	0.03444	0.08036	0.06478	0.06658
60000	70000	10000	0.45917	0.63980	0.66403	-0.94814	0.03252	0.06525	0.06586	0.09192
70000	80000	10000	0.46958	0.67050	0.78358	-0.72019	0.03440	0.07074	0.07703	0.07102
80000	90000	10000	0.47648	0.70977	0.85593	-0.67477	0.03413	0.07252	0.08139	0.06533
90000	100000	10000	0.44772	0.75083	0.77686	-0.74664	0.03346	0.07089	0.06878	0.06617
		Average:	0.46524	0.74534	0.76219	-0.75801				