AmpTools User Guide
v0.11.0

Matthew Shepherd
Indiana University

January 19, 2021

Preface

Important note to the reader

The complete AmpTools user guide is under development. It will continue to be updated in future releases of AmpTools. The sections that exist in this guide at present are fairly complete and detailed. Any questions or requests for clarification should be sent to *mashephe@indiana.edu*. In addition to reading the documentation here, one may find the tutorial provided in `Tutorials/Dalitz` of the main software package to be helpful for illustrating a use-case for the framework.

Contents

Chapter 1

Introduction and Overview

AmpTools is a set of libraries that are intended to enable conducting unbinned maximum likelihood fits to experimental data. The package was developed to enable an organized approach to "amplitude analysis" or "partial wave analysis," but the core tools can be extended to Monte Carlo generation, visualization of fit results, and can, in principle, be adapted to other types of unbinned maximum likelihood fits, like sums of moments. By design, the core AmpTools libraries contain no information about the format of experimental data or any physics models that are fit to these data. This provides an experiment independent framework that can be used for a variety of applications. It also means that to the end user, the AmpTools library itself cannot be utilized without writing a substantial amount of code in order to read in data, compute amplitudes, and perform fits. It is recommend that the starting user first study the fully functional Dalitz tutorial package that is distributed with AmpTools in the `Tutorials` directory.

Chapter 2

Likelihood Construction

AmpTools utilizes the extended unbinned maximum likelihood method to estimate the values of model parameters used to describe data. For the discussion that follows, let's assume that $\mathbf{x}$ is a vector of dimension n that describes the kinematics, or "phase space," of the reaction[1]. For example, for a conventional Dalitz decay, $n = 2$ and $\mathbf{x}$ is a location on the two-dimensional Dalitz plot. The physics model that is being fit to the data will be a function of $\mathbf{x}$ and will contain m parameters $\boldsymbol{\theta}$. We seek to utilize the data to obtain the best estimates for the values of the parameters $\boldsymbol{\theta}$, and do so by searching the m-dimensional space for the value that maximizes the likelihood.

Given a set of N independent observations $\mathbf{x}_i$ we can write the extended maximum likelihood as a function of the parameters $\boldsymbol{\theta}$ as

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{e^{-\mu}\mu^N}{N!} \prod_{i=1}^{N} \mathcal{P}(\mathbf{x}_i; \boldsymbol{\theta}), \tag{2.1}$$

were $\mathcal{P}$ is the n-dimensional probability density. The function $\mathcal{P}$ can be written in terms of the "intensity" $\mathcal{I}(\mathbf{x}; \boldsymbol{\theta})$, which we will define as the model-predicted number of signal events per unit phase space. We can write the total number of experimentally observed events in terms of the intensity and the efficiency $\eta(\mathbf{x})$. Here the term "efficiency" is the probability that an event with kinematics $\mathbf{x}$ will be detected by the detector and pass all subsequent event selection criteria to make it into the final sample N events that used as an input to the fit. Therefore, we can write the model-predicted number of observed events for some set of parameters $\boldsymbol{\theta}$ as

$$\mu = \int \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \; d\mathbf{x}, \tag{2.2}$$

and the probability density function as

$$\mathcal{P}(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{\mu} \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}). \tag{2.3}$$

In practice, rather than maximizing the likelihood, it is useful to minimize $-2 \ln \mathcal{L}$. Since the goal is to find the minimum of $-2 \ln \mathcal{L}$ in the m-dimensional parameter space that contains $\boldsymbol{\theta}$ then terms that contribute to $-2 \ln \mathcal{L}$ that do not depend on the parameters $\boldsymbol{\theta}$ can be neglected when constructing the function to minimize. Plugging the above expressions into Equation 2.1 and computing, one obtains

$$-2 \ln \mathcal{L}(\boldsymbol{\theta}) = -2 \left(\sum_{i=1}^{N} \ln \mathcal{I}(\mathbf{x}_i; \boldsymbol{\theta}) - \int \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \; d\mathbf{x} \right) + c_1, \tag{2.4}$$

[1] In this entire discussion we are describing the construction of a likelihood function for a single reaction. AmpTools also supports simultaneous fitting of multiple reactions. In such a fit the products of the likelihoods for all reactions is maximized.

where c_1 is a constant that does not depend on $\boldsymbol{\theta}$. The right hand side of the above equation, neglecting c, is what is computed by the AmpTools framework and supplied to the minimizing algorithm (currently Minuit) for minimization[2]. In what follows we discuss some details of this computation.

2.1 The Intensity

Conceptually the "intensity" is a function that describes the density of events in phase space in the final data sample. Usually the goal is to develop a model with parameters to describe the intensity. Then use the data to obtain best estimates for the parameters in the model from which one can draw quantitative conclusions in the context of the model.

In an abstract sense, the likelihood computation that is implemented in AmpTools only requires that the intensity be defined. Within the framework there is the notion of an `IntensityManager` that is able to provide the intensity for any event. This layer of abstraction allows for different implementations of the computation of the intensity, *e.g.*, one based on interfering amplitudes, sums of moments, or simple functions. In practice, the only computation of the intensity that is implemented in AmpTools now is one based on interfering amplitudes, and this computation is handled by the `AmplitudeManager` object, which inherits from the `IntensityManager`. For each reaction of interest in the fit, there is a single instance of the `AmplitudeManager` that computes the intensity according to the following formula.

$$\mathcal{I}(\mathbf{x}) = \sum_{\sigma} \left| \sum_{\alpha} s_{\sigma,\alpha} V_{\sigma,\alpha} A_{\sigma,\alpha}(\mathbf{x}) \right|^2 . \tag{2.5}$$

Here σ indexes the coherent sum. Multiple coherent sums are needed to manage cases where there are a variety of quantum mechanically distinguishable processes, typically sums over the spin states of external particles. The index α runs over the individual amplitudes, usually associated with indistinguishable processes, that are in each coherent sum.

In the expression above, $A_{\sigma,\alpha}(\mathbf{x})$ is a complex-valued function that describes a particular amplitude, *e.g.,* a decay through a particular resonance. This expression may or may not have free parameters, *e.g.,* the mass of a resonance, and the existence or absence of parameters affects the computational complexity of the fit, as described below. In practice the expressions $A_{\sigma,\alpha}(\mathbf{x})$ often factorize for example into kinematic and dynamical terms. To enhance modularity and portability of code, this factorization is supported in the framework. That is, the framework assumes mathematically that

$$A_{\sigma,\alpha}(\mathbf{x}) = \prod_{\gamma=1}^{n_{\sigma,\alpha}} a_{\sigma,\alpha,\gamma}(\mathbf{x}), \tag{2.6}$$

where the user writes code to define the function $a_{\sigma,\alpha,\gamma}(\mathbf{x})$ in an `Amplitude` class and the configuration file specifies the $n_{\sigma,\alpha}$ factors that are multiplied together to construct the complex-valued function $A_{\sigma,\alpha}(\mathbf{x})$.

The numbers $V_{\sigma,\alpha}$ are referred to in the code in places as "production factors" or "production coefficients," and they are complex-valued numbers that represent the magnitude and phase of the production of a particular amplitude $A_{\sigma,\alpha}(\mathbf{x})$. These are typically free parameters in the fit. Finally the factors $s_{\sigma,\alpha}$ are real-valued "scale factors" that are useful for imposing constraints on different contributions to the amplitude based on isospin relations or Clebsch-Gordan coefficients. The scale factors permit one to constrain two different $V_{\sigma,\alpha}$ coefficients to be the same up to a fixed (or floating) scale. In the simplest instances the fit parameters $\boldsymbol{\theta}$ are the set of $V_{\sigma,\alpha}$, but they may include scale factors or other parameters embedded in the definition of $A_{\sigma,\alpha}(\mathbf{x})$.

[2] Almost! There is a provision for event-by-event weights that the weight the terms in the sums. This is omitted for simplicity at this point in the discussion but expanded on in detail in the sections that follow.

2.1.1 Managing Permutations of Identical Particles

The ability to factorize amplitudes as written in Equation 2.6 enhances the overall modularity of user amplitudes as often the same dynamical factors, *e.g.*, Breit-Wigner functions, appear in many different amplitudes. However, it introduces a complication in the case that amplitudes must be symmetrized for identical particles. Let's assume that the final state of a reaction results in the production of two identical pions that we label sequentially as π_1 and π_2. The four-vectors of these detected pions are used to derive the kinematical variables $\mathbf{x}$ that are arguments to the amplitudes, and, since both pions are identical there are two choices for the values of $\mathbf{x}$: $\mathbf{x}(\pi_1, \pi_2)$ and $\mathbf{x}(\pi_2, \pi_1)$. In this case, the LHS of Equation 2.6 must be symmetrized according to

$$A_{\sigma,\alpha}(\mathbf{x}) \rightarrow \frac{1}{\sqrt{2}} \left[A_{\sigma,\alpha}\left(\mathbf{x}(\pi_1, \pi_2)\right) + A_{\sigma,\alpha}\left(\mathbf{x}(\pi_2, \pi_1)\right) \right]. \tag{2.7}$$

In order to preserve the modularity of the code, it is important that this symmetrization happen in the framework and not in the user-written definitions of the amplitude factors that appear on the RHS of Equation 2.6. AmpTools will automatically symmetrize amplitude if it notices two particles as specified by the `reaction` keyword in the configuration file have identical names. Additionally in the user can force symmetrization of particles using the `permute` command in the configuration file.

2.2 Monte-Carlo Integration of the Intensity

In order to normalize the function $\mathcal{P}(\mathbf{x}_i; \boldsymbol{\theta})$ it is necessary to compute $\int \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \, d\mathbf{x}$. This is impossible to analytically compute as the acceptance function $\eta(\mathbf{x})$ in general has an unknown analytic form. The integral is instead performed using Monte-Carlo integration. It is possible to write the integral of a function $f(x)$ in terms of its average value as

$$\int_R f(x) \, dx = R \langle f(x) \rangle. \tag{2.8}$$

In the expression above R is the size of the region of integration and $\langle f(x) \rangle$ denotes the average value of the function. The average value can be obtained by random sampling of the function over the region, and the precision of the integral depends on the smoothness of the function over the region and how well the random sampling covers the region.

Using this strategy we can integrate the intensity by generating a Monte-Carlo sample that uniformly populates the multi-dimensional space that spans the domain of the intensity function. For a sample of M_g generated Monte-Carlo events, we can compute the average value

$$\langle \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \rangle = \frac{1}{M_g} \sum_{i=1}^{M_a} \mathcal{I}(\mathbf{x}_i; \boldsymbol{\theta}), \tag{2.9}$$

where the sum on the right hand side runs over the sample of M_a events that pass all of the analysis criteria[3].

For minimizing $-2 \ln \mathcal{L}$ the RHS of the equation above is a suitable substitute for $\int \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \, d\mathbf{x}$, but strictly speaking it is only the value of the integral up to a factor that is the size of the region of integration. Mathematically one can rescale all of the $V_{\sigma,\alpha}$ parameters by the square root of this factor. This results in a modification of the first term on the RHS of Equation 2.4: a multiplication of the intensity in that expression by a constant. Since a log of products is a sum of logs, this rescaling results in a constant offset $-2 \ln \mathcal{L}$ (equal to $2N$ times the log of the size of the region of integration) and does not affect the minimization procedure.

In order to optimize the computation of $\int \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \, d\mathbf{x}$ it is useful to factorize the production coefficients from the sum. One can write

$$\langle \mathcal{I}(\mathbf{x}; \boldsymbol{\theta}) \eta(\mathbf{x}) \rangle \quad = \quad \sum_{\sigma} \left| \sum_{\alpha} s_{\sigma,\alpha} V_{\sigma,\alpha} \left\{ \frac{1}{M_g} \sum_{i=1}^{M_a} A_{\sigma,\alpha}(\mathbf{x}_i) \right\} \right|^2$$

[3] This is equivalent to setting the value of the function $\eta(\mathbf{x})$ to 1 for accepted events and 0 for others.

$$= \sum_{\sigma} \sum_{\alpha,\alpha'} s_{\sigma,\alpha} s_{\sigma,\alpha'} V_{\sigma,\alpha} V^*_{\sigma,\alpha'} \left\{ \frac{1}{M_g} \sum_{i=1}^{M_a} A_{\sigma,\alpha}(\mathbf{x}_i) A^*_{\sigma,\alpha'}(\mathbf{x}_i) \right\}, \tag{2.10}$$

and notice that the term on the RHS in curly braces, a Hermitian matrix for each sum σ with dimension equal to the number of amplitudes in the sum, does not depend on the parameters of the fit as long as the functions $A_{\sigma,\alpha}$ do not contain any free parameters. These matricies are referred to as "normalization integrals" and can be computed and cached in advance of the fit to reduce computational burden. If, however, the $A_{\sigma,\alpha}$ contain fit parameters then these integrals can, in principle, change with each fit iteration and they must be recomputed, which involves an expensive sum of accepted Monte-Carlo events, with every fit iteration.

The AmpTools framework will detect the presence of free parameters in the amplitudes and respond appropriately. In the case that there are parameters in the amplitude, and one of them changes with an iteration of Minuit then all of the "normalization integrals" that depend on that amplitude will be recomputed at the next computation of the likelihood.

For the purpose of interpreting the results of a fit, it is useful to have integrals of the amplitudes that do not include detector acceptance. These integrals can then be multiplied by the fit parameters to predict the number of true (acceptance corrected) events associated with some amplitude that were produced prior to being accepted by the detector and analysis criteria. These numbers are often tied to the physical observables one is trying to extract. To compute these "amplitude integrals" one repeats the sum in curly braces in Equation 2.10 but this time summing over the generated MC events instead of the accepted MC events. Since these numbers are only useful in interpreting the results and not needed for construction of the likelihood they are only computed at the end of the fit. Throughout the AmpTools code there has been an attempt to keep naming convention such that variables with the name `normInt` refer to integrals computed over accepted MC events, and the name `ampInt` refers to these latter integrals that do not include detector acceptance. Computation and caching of both sets of integrals is managed by the `NormIntInterface` in AmpTools.

2.2.1 Comments and Assumptions about Precision

For the sake of constructing the likelihood, the expression of the average value in Equation 2.10 is assumed to be exact. Recall that it translates into a normalization factor in generating the probability density function that describes the distribution of events. In reality the normalization integrals have a statistical uncertainty that arises from the finite size Monte-Carlo set that used to compute them. This statistical uncertainty can be evaluated numerically by computing not only the average value but the variance of the function. In principle one can then make a quantitative decision as to whether such statistical uncertainties can be safely neglected. In practice, one tends to test this assumption by reducing or increasing the size of the Monte-Carlo sample and ensuring that the results do not change appreciably.

In some instances it will be very difficult to obtain the desired statistical precision for the normalization integrals. For example, suppose that one is examining a reaction with two ϕ mesons in the final state, both of which decay to K^+K^-. If the parent Monte-Carlo sample that is used to perform the numerical integration is generated according to four-body phase space and contains $2(K^+K^-)$ then the $\phi\phi$ amplitude will only be significantly non-zero in a very small amount of the phase space, and many MC events will need to be generated to populate this region sufficiently such that the integral has the desired precision. In general this is a problem that arises with narrow resonances amongst a relatively large number of final state particles. The problem can be mitigated by using a technique known as "importance sampling." In the case above one would generate MC events according a $\phi\phi$ distribution but then apply a weight to each event such that when considering the sum of the weights the sample is distributed according to four-body phase space. AmpTools supports such implementation (use of weights is discussed in detail below) but it relies on the user to develop a technique for providing the weighted MC sample. These importance sampling techniques have the benefit that, when used properly, can increase the numerical precision of the normalization integrals while decreasing the CPU burden both in generating signal MC and performing the fit.

2.3 Using Weighted Events

AmpTools supports the ability to weight events, which has utility for dealing with a variety of issues in addition to the importance sampling technique for doing MC integration discussed above. The goal of this section is to describe where weights may enter the various calculations and provide some general guidance as to when one might choose to use weights.

By default every event that is read into the AmpTools framework has a weight of unity. The framework relies on the user to write a data reader that inherits from the templated class provided by the framework called `UserDataReader`. In doing so the user must write a function that retrieves events from the source of the users choosing (file, database, etc.) and creates objects of type `Kinematics`, which contain the relevant four-momenta of the particles in an event. At the time the `Kinematics` object is created its weight can be specified. It is the responsibility of the user to decide and define (outside of the AmpTools framework) what these weights should be and propagate the choice of weight into the data reader class and use it when `Kinematics` objects are created. Use of non-uniform weights is a powerful tool, but should be done with caution.

The AmpTools framework requires three different classes of events be provided in order to perform a fit: data, generated MC, and accepted MC. An optional fourth class of events, a background or sideband sample, can be provided and is discussed in the next section. All events enter through the a user-defined data reader class and, in principle, all could contain weighted events, but it is up to the user to determine if any *should* contain weighted events.

In the context of fitting data, the weights appear in two places in the construction of the likelihood. In Equation 2.4 the presence of non-unity weights results in the following substitution in the computation.

$$\sum_{i=1}^{N} \ln \mathcal{I}(\mathbf{x}_i; \boldsymbol{\theta}) \rightarrow \sum_{i=1}^{N} w_i \ln \mathcal{I}(\mathbf{x}_i; \boldsymbol{\theta}), \tag{2.11}$$

where w_i is the weight of the i^{th} event. This sum is calculated over data events, so only modification of data event weights affects this sum. Such weighting in the sum is one mechanism for treating background, as is discussed in detail below.

The other place that weights enter the likelihood is in the MC integration procedures for the intensity. The presence of non-unity weights in either the accepted or generated MC samples will result in a modification of the so-called normalization or amplitude integrals, respectively, by the following substitution in Equation 2.10.

$$\frac{1}{M_g} \sum_{i=1}^{M_a} A_{\sigma,\alpha}(\mathbf{x}_i) A^*_{\sigma,\alpha'}(\mathbf{x}_i) \rightarrow \frac{1}{M_g} \sum_{i=1}^{M_a} w_i A_{\sigma,\alpha}(\mathbf{x}_i) A^*_{\sigma,\alpha'}(\mathbf{x}_i), \tag{2.12}$$

where again w_i is the the weight applied to the i^{th} accepted or generated, depending on the integral in question, MC event. The ability to reweight MC events in the likelihood computation is valuable for exploring sensitivity to systematic effects due to the acceptance of the MC. For example suppose, one might expect the acceptance in an angular region of the detector may be less than predicted by MC, then the MC can be reweighted with a weight that depends on angles of the particles and the fit can be repeated to examine the change in the result. Such reweighting and repeating of the fit is often more efficient than generating a new MC sample with a different model for the acceptance, and it removes statistical fluctuations that may be present in the two independent MC samples. The reweighting of MC must be done by the user outside of the AmpTools framework and passed in through the data reader. The AmpTools framework only provides a mechanism to use the weight in the subsequent fit.

In principle, reweighting the data affects the intrinsic assumptions about Poisson statistics that are in the core formulation of the extended maximum likelihood. It is important, especially when using weights, that the user validate that the fit procedure is producing unbiased estimates for parameters with appropriate uncertainties. This can be done by fitting an ensemble of mock data samples where the true values of parameters are known. Reweighting MC events in the computation of the normalization integrals is less

worrisome as this procedure simply modifies the computation of a single number, the average value of the intensity, that is assumed to be exactly know in the formulation of the likelihood.

2.4 Techniques for Managing Background

Extracting estimates for the values of model parameters in the presence of background can be challenging. There are several types of backgrounds and likely just as many ways for dealing with them. The most important advice is that one test the procedure that is used (using mock data with varying levels of background) to ensure that the procedure coupled with the presence of background does not result in biased estimates of parameters or their uncertainties.

2.4.1 Incorporating Background into the Intensity

When the form of the background and its dependence on the kinematic variables $\mathbf{x}$ is known then the most straightforward way to accommodate for background in the fit is to include either an amplitude or a sum that describes this shape in the fit. The choice of amplitude or sum depends on the assumptions that one wants to make. An additional coherent sum could be created with a single amplitude that describes the background. This background sum will then be added incoherently with the remainder of the amplitudes. Similarly one could add an amplitude to an existing sum that is added coherently with other amplitudes. This may be suitable for example if one is trying to describe a non-resonant background process that has identical initial and final state particles to the "signal" process that the other amplitudes in the sum describe.

The remainder of this section is dedicated to discussing how to manage background in the sense of an unbinned fit, where one doesn't have an analytic description of the background, but has the ability to simulate background or use a separate class of data events (like sideband or out of time events) as an accurate representation of the level of background. To simplify the terminology in the discussion that follows, we will assume that anything that can be described with the intensity function is "signal" independent of whether it corresponds to actual signal terms in the intensity or other terms used to accommodate backgrounds. Likewise, consistent with above, the variable μ will always represent the model-predicted number of observed signal events as defined in Equation 2.2.

2.4.2 Subtracting Background with Negative Weights

In the derivation of Equation 2.4 we assumed that the N events were all described by the signal intensity distribution $\mathcal{I}(\mathbf{x};\boldsymbol{\theta})$. That is, unless the intensity is constructed such that it can accommodate the background, as described in the previous section, then we've assumed that our data sample is free of background. If this is not the case, then we need to attempt to apply a correction to the first sum on the RHS of Equation 2.4 in order to subtract the contribution to the sum from background events.

Typically, in the signal region one has no ability to distinguish the signal events from the background events. (If there were such an ability, one would simply remove the background events!) Usually one estimates the contribution to the background using a separate sample of events, *e.g.*, from the "sidebands" of some discriminating variable or perhaps from a simulation. If the signal region contains $N = N_S + N_B$ events where N_S (N_B) is the number of signal (background) events, then only N is known exactly and N_B must be estimated. Let's assume the sample used for background estimation has $\widetilde{N}_B$ independent events and, based on the scaling arguments, *e.g.* size of sidebands or size of generated samples, predicts that there are β background events in the signal region[4]. That is, β is a prediction for the size of N_B based on $\widetilde{N}_B$ events that are not part of the N events in the signal region. Now, in order proceed under the assumption in Equation 2.4 that the first term on the RHS is a sum over just signal events, one replaces this sum with

[4]To try to keep notation consistent let β be the predicted level of background in the signal region and μ be the integral of the intensity, or the predicted level of signal in the signal region. The value of μ can be computed by integrating the intensity and β typically comes from sidebands or simulation.

a sum over weighted events that approximates the signal contribution.

$$\sum_{i=1}^{N} \ln \mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}) \rightarrow \sum_{i=1}^{N_S} \ln \mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}) \approx \sum_{i=1}^{N=N_S+N_B} \ln \mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}) + \sum_{i=1}^{\widetilde{N}_B} w_i \ln \mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}), \tag{2.13}$$

where the weights w_i of the $\widetilde{N}_B$ events in the second term are given by

$$w_i = -\frac{\beta}{\widetilde{N}_B}. \tag{2.14}$$

The AmpTools package supports the above approach to fitting with background. The user needs to only determine the appropriate value for the weights and be sure that the weights are passed in via the user-defined data reader. In the context of AmpTools the only thing that is relevant is that the two samples that contain N and $\widetilde{N}_B$ events, the signal-rich and background only samples, be passed to the AmpTools framework merged as `data`.

There are variations of the above procedure that can be implemented in a similar fashion. For example, instead of having two distinct weights for the signal region one may set the weight on an event-by-event basis corresponding to the probability that the event is a true signal event (as determined from some other independent piece of information). In doing so, one needs to maintain the constraint that the sum of the weights in the data file is a predictor of N_S the number of signal events, as this is what the fit is trying to match to μ which is derived from integrating the intensity.

The subtraction in Equation 2.13 is not exact because the $\widetilde{N}_B$ events are not an exact replication of the N_B background events in the sample coming from the signal region. This approximation is likely good in the limit the N_B is small with respect to N_S or that $\widetilde{N}_B$ is very large, but nevertheless the effect, particularly on the errors associated with the fit parameters, cannot be quantified without some additional work, and this inexact cancellation has other potential side-effects as discussed in Section 2.4.4.

2.4.3 Specifying a Background Sample

The approach, discussed in the previous section, of correcting the sum of the logs of the intensity such that it reflects only a signal contribution seems to be relatively common. It turns out that if one retraces the derivation of Equation 2.4 (rather than trying to correct a single term in this equation), one actually finds additional terms remain in the expression for $-2\ln\mathcal{L}$ that depend (likely very mildly) on the fit parameters. In this section we describe this derivation and how to include the extra terms. Practically speaking, to exercise this method one must specify a separate background data file to AmpTools. This can be advantageous from the point of view of organizing one's analysis workflow or for subsequent plotting of background contributions various distributions. This is the recommended way of managing backgrounds in AmpTools but it is also not without assumption or potential pitfall.

First, following the notation in the previous section, let's assume that one can use a separate sample of $\widetilde{N}_B$ events to predict that the signal region of a the data sample that one would like to fit contains β background events. Let's assume this prediction is exact. In reality the uncertainty of the prediction probably needs to be negligible on the scale of the counting error on the true number of background events in the signal region. This can typically be achieved by large MC samples (if the background comes from MC simulation) or sidebands that are sufficiently larger than the signal region (if the background comes from data)[5]. With this prediction β of the number of background events and μ being the predicted number of signal events we can then write the likelihood as

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{e^{-(\mu+\beta)}(\mu+\beta)^N}{N!} \left(\prod_{i=1}^{N} \mathcal{P}(\mathbf{x}_i;\boldsymbol{\theta}) \right) / \left(\prod_{i=1}^{\widetilde{N}_B} \mathcal{P}(\mathbf{x}_i;\boldsymbol{\theta}) \right)^{\frac{\beta}{\widetilde{N}_B}}, \tag{2.15}$$

[5]The words "large" and "sufficiently larger" are vague, and the appropriate values likely depend on the particular analysis. The user should always validate the procedure with ensembles of mock data.

where the last term corrects the contribution to the likelihood from the background events in the signal region by dividing out the geometric mean of the probability of all the known background events raised to the power of β. The function $\mathcal{P}(\mathbf{x}_i;\boldsymbol{\theta})$, by construction, only describes the probability of observing a signal event – it is of no use to tell us the probability of finding a background event with particular kinematics. The only thing we can attempt to do is use a known background sample to cancel contributions to the likelihood from the N_B events in the signal region. As noted previously, this cancellation is not exact as the sample of $\widetilde{N}_B$ true background events is statistically independent from the N_B background events in the signal region whose contribution to the likelihood one is attempting remove.

Taking the log of Equation 2.15, recalling that $\mu = \int \mathcal{I}(\mathbf{x};\boldsymbol{\theta})\eta(\mathbf{x})d\mathbf{x}$ and β is a constant determined from the independent sample of $\widetilde{N}_B$ events, one obtains

$$-2\ln\mathcal{L}(\boldsymbol{\theta}) = -2\left(\sum_{i=1}^{N}\ln\mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}) - \sum_{i=1}^{\widetilde{N}_B}\frac{\beta}{\widetilde{N}_B}\ln\mathcal{I}(\mathbf{x}_i;\boldsymbol{\theta}) - \mu + \{N\ln(\mu+\beta) - (N-\beta)\ln\mu\}\right) + c_2, \quad (2.16)$$

where c_2 is a constant that does not depend on $\boldsymbol{\theta}$ and is irrelevant for the minimization procedure. The first three terms on the RHS of Equation 2.16 are equivalent to the expression for $-2\ln\mathcal{L}$ under the substitution given in Equation 2.13. The remaining term in curly braces goes to zero as $\beta \to 0$ and no such term appears in the expression in Equation 2.4. In the case that a separate background sample is provided to the AmpTools framework, the function that is actually minimized is given by the non-constant portion of the RHS of Equation 2.16. The difference between minimizing this function and minimizing Equation 2.4 under the substitution in Equation 2.13 (incorporating negative weights into the data sample) is expected to be small. In practice, under variation of the fit parameters $\boldsymbol{\theta}$ the minimization procedure tends to keep the value of μ fixed to $N-\beta$, and hence the term in curly braces in Equation 2.16 tends to be virtually independent of the fit parameters[6] In principle the inclusion of this term may make small improvements in the estimation of the parameter uncertainties.

In summary, the recommended method of incorporating backgrounds in a fit with AmpTools is to prepare a separate background sample such that the sum of the weights of this sample is equal to the predicted background in the signal region, which can be achieved by giving every event in the sample a weight of $\beta/\widetilde{N}_B$. This sample should be passed into AmpTools using the `bkgnd` keyword in the config file. The weighted sum of the sample should represent the distribution of background across the phase space of the fit. The sum of the weights should be equal to the number of background events in the signal region that are present in the file specified by the `data` keyword.[7] Note that this sum must be positive; therefore, if one is transitioning from a method of background subtraction that uses negative weights as discussed in Section 2.4.2, the signs of the background events need to be inverted after removing them the `data` file. The strategies outlined in Sections 2.4.2 and 2.4.3, cannot be mixed. If a background file is provided to the fit, then the weight of every event in the `data` file must be unity or else the fit result will have the incorrect scale.

2.4.4 Remaining Background Challenges

The methods outlined above for dealing with backgrounds are not complete and still, can, in principle fail dramatically. One can envision a recipe for failure by looking at Equation 2.15. Suppose that the intensity, which describes only signal events, is such that it (and consequently $\mathcal{P}$) has a zero to some particular value of $\mathbf{x} = \mathbf{x}_0$. Such zero could arise for example from node in an angular distribution or from some other physically motivated scenario. If the intensity is a valid description of the signal then it will be impossible to ever have a signal event with kinematics such that it lands in the location $\mathbf{x}_0$. However, the background is

[6] One example of where μ may deviate from $N-\beta$ is in a simultaneous fit to multiple reactions where fit parameters are constrained between the reactions. In such a fit there is no freedom to fix μ to $N-\beta$ for all reactions.

[7] In AmpTools v0.10.x and prior, the weights in this sample were set by the framework to be the same sign to avoid confusion about the sign. This limited the ability to have mixed-sign background weights which is necessary for sideband subtraction in more than one dimension. This limitation was removed in v0.11.0 of AmpTools but this change results in a fundamentally different change in behavior of how the background file is used in the fit.

not typically distributed like the signal and it is possible that there is considerable background in the vicinity of $\mathbf{x}_0$. If background events from either the signal region or the separate background sample populate some region close to $\mathbf{x}_0$ then the expression for $-2\ln\mathcal{L}$ becomes very sensitive to these events and the fact that the cancellation of contributions to $-2\ln\mathcal{L}$ from backgrounds in the signal and background samples is not exact could lead to a sensitivity to statistical fluctuations in the background that is not captured in the formulation of the likelihood. This effect is discussed in detail in Ref. [**?**]. However, the proposed solution in this reference requires the tracking variance in background contributions the likelihood in a region of $\mathbf{x}$ by binning $\mathbf{x}$. It has yet to be demonstrated that the technique can be extended to an unbinned fit where the dimension of $\mathbf{x}$ is necessarily larger than one or two.

Fortunately, there are techniques for exploring whether a user's fit, with background included, is teetering on the verge of calculating $\ln 0$. The recommended approach is to employ the bootstrap technique to evaluate the stability of the fit under statistical fluctuation in the signal and background samples. While the AmpTools framework does not have a built in bootstrap evaluation procedure, the user can easily deploy this technique in the context of a user defined data reader. The idea behind bootstrap is to generate many (of order 100) samples of N signal events or $\widetilde{N}_B$ background events from the initial sample but with random oversampling. In each of these samples some initial events will be omitted and others will be repeated. Each of these samples is then fit independently and one examines the variation in the output fit parameters. This variance (and covariance between parameters) is a measure of the statistical uncertainty in the fit and encodes effects like statistical fluctuations in the background that are not explicitly capture in the likelihood. It is expected that bootstrap, while slightly more CPU intensive, will result in a more accurate estimate of statistical uncertainties than those reported by the fitter, which are computed from examining changes in the likelihood function. To summarize, the strategy for managing background in the likelihood fit in AmpTools is to cancel out the contributions to the likelihood function from background events in the signal region using an independent sample of pure background. Since the goal of that exercise is to make the likelihood calculation insensitive to background and hence statistical variations in distribution of background, one needs to a technique to evaluate parameter uncertainties due to statistical fluctuations of both background in the signal region and the independent background sample. The bootstrap technique does just that.

Chapter 3

The Configuration File

The configuration file specifies all data files that are input to the fit, how to construct the intensity from amplitudes, and how to initialize the fit. It is a text file. Any line that is blank or begins with a hash (#) is ignored. All other lines must begin with one of the keywords documented below. There is no mechanism for continuing the commands on one line to the next. The complete syntax for any keyword must exist on a single line. The ordering of the lines in the configuration file is not relevant. There is no need to declare sums or reactions before using them as the parser will parse individual lines in an order determined by the keyword on each line. A functionally equivalent copy of the configuration file is written to the file that logs the output of the fit. *In the documentation that follows, any argument to a keyword that appears in angle brackets is a required argument while arguments appearing in parentheses are optional.*

At present, the only formulation of the intensity that is supported in AmpTools is that constructed from interfering amplitudes. Therefore, the structure and keywords in the configuration file are such that they correspond to a formulation of the intensity like that in Equation 2.5.

All information about the configuration of the fit is contained in an object call `ConfigurationInfo`. The class `ConfigFileParser` is able to generate a `ConfigurationInfo` object from a text file. This approach preserves the possibility in the future to devise alternate mechanisms or sources for configuring the fit. The only requirement is that a compute `ConfigurationInfo` object be produced as part of the configuring process.

3.1 Processing Directives and Special Characters

In order to facilitate organization, functionality, and accuracy of configuration files several keywords can be utilized.

3.1.1 `include`

```
include <file>
```

The `include` keyword will insert the contents of the file named `file` at the present location in the configuration file. The string `file` must be defined with respect to the current directory at execution time. The `include` directive is particularly useful if there are whole sets of amplitudes that the user would like to optionally switch off or on. These can be moved to a separate file and the leading character of the single `include` line can be # or not to toggle off and on the set of amplitudes.

3.1.2 `define`

```
define <word> <defn1> (defn2) (defn3) (...)
```

The `define` directive replaces the isolated string `word` with the one or more strings specified by the subsequent arguments. This is particularly useful for defining numeric parameters for amplitudes. For example, assume that multiple amplitudes need, as an argument, the mass and width of the $f_2(1270)$. It may be advantageous to have a line stating `define f2_1270 1.270 0.187`. Then, for each amplitude that needs these arguments in succession the string `f2_1270` can be used.

3.1.3 keyword

```
keyword <userkeyword> <min arguments> <max arguments>
```

The `keyword` directive lets the user define a new keyword given by `userkeyword` and the minimum and maximum number of arguments that should be provided to the command is given by the final two arguments. The user-defined keyword and arguments can then be utilized throughout the configuration file. The user keywords do not change the functionality of the core AmpTools framework, but they provide an opportunity for the user to pass commands through the configuration file to user-written code that can retrieve the keyword arguments and utilize them at runtime. Retrieval of keywords and arguments is done through the `ConfigurationInfo` class that is provided by the `AmpToolsInterface`.

3.1.4 loop

```
loop <loopname> (val1) (val2) (val3) (...)
```

The `loop` keyword defines a loop, which is a useful to configuration files that have many repetitive lines. If the name `loopname` is detected in the line of a configuration file, that line will be repeated n times, where n is the number of values provided after the `loopname` in the `loop` command. With each repetition `loopname` is replaced with each of the values in sequence. If two (or more) `loopnames` are used in the same line, they should be of the same length and the loop will step through the arguments of each in sync, still resulting in a total of n lines to the configuration file. (Note: for debugging purposes it may be valuable to use the `ConfigurationInfo::display()` function to verify the fit is configured as intended. In addition the `FitResults` object should contain a configuration file that is absent of any preprocessing directives.)

3.1.5

The hash is the comment character. When it is placed as the first character of a line, the line will be ignored. It is not possible to comment out part of a line. Only the entire line can be commented out.

3.1.6 []

Square brackets are used to denote that the string inside references a parameter in the fit. They are typically used when passing arguments to amplitudes. For example a Breit-Wigner amplitude may take, as an argument, the width of the Briet-Wigner in units of GeV. In this case the user may specify the width to be `0.120` or `[bwWidth]`. In the first case, the width is fixed to 120 MeV while the second means the the width is obtained by the parameter `bwWidth`, which may be either fixed or floating but must be declared using the `parameter` keyword.

3.1.7 ::

The double-colon is parsed as if it is a space. This construct appears because arguments to many keywords often reference an amplitude using its reaction name, sum name, and amplitude name. The clarity of the configuration file can be improved by writing these three specifiers, which always come together, as `reaction::sum::amplitude`.

3.2 Fits, Reactions, and Sums

Each global configuration is referred to as a fit and can be named. Within a fit one can have one or more reactions. A reaction typically refers to a unique set of initial and final state particles. In some instances multiple reactions could refer to multiple decay modes of the same set of final state particles. If one is doing an amplitude analysis, then within each reaction there may be one or more coherent sums. All amplitudes within a given sum are added coherently, *i.e.,* they added as complex numbers such that it is possible to generate interferences between amplitudes. All of the sums associated with a reaction are added incoherently, as real numbers, in the construction of the intensity. Typically there are different sums for each quantum-mechanically distinguishable reaction, *e.g.*, for each unique configuration of external spins.

3.2.1 `fit`

```
fit <fitname>
```

The `fit` command names the fit. The only significant practical consequence of this command for the user is that the output of the fit will be written to a file with the name `fitname.fit`.

3.2.2 `reaction`

```
reaction <reaction> <particle1> (particle2) (particle3) (...)
```

The `reaction` keyword declares a reaction that is named by the first argument. The subsequent arguments are the names of the particles in the reaction. The number of particles and ordering of particles should correspond to the number and ordering of the four-vectors that are provided by the user-defined data reader. The names of the particles are for the user to decide and, with one critical exception, have no functional significance in performing the fit. The exception is for cases when AmpTools notices that two particles have exactly the same name. In this case it will assume the particles should be treated as identical particles in the construction of the intensity and all particles with identical names will be permuted to symmetrize the amplitude. (See the discussion in Section 2.1.1.) The user can disable permutation of particles kinematics in construction of the amplitude by choosing particle names that are all unique.

3.2.3 `sum`

```
sum <sum1> (sum2) (sum3) (...)
```

The `sum` keyword declares one or more coherent sums. The option of specifying multiple sums is only to make the configuration file more efficient and is functionally equivalent to specifying each sum with individual `sum` keyword lines.

3.3 Configuring Amplitudes and Parameters

The amplitudes that are added coherently to construct the intensity that describes the data (as in Equation 2.5) can be constructed from multiple factors that are multiplied together, as noted in Equation 2.6. Each of these factors $a(\mathbf{x})$ is defined by user-supplied class that inherits from the `Amplitude` base class (see Chapter **??** for a detailed discussion). Each amplitude factor is declared by one line on the configuration file. In the absence of the use of the `constrain` command, there will be one complex production coefficient that is a free complex parameter in the fit (the $V_{\sigma,\alpha}$ in Equation 2.5) created for each amplitude. The following commands are useful for configuring the amplitudes and initializing the parameters of the fit.

3.3.1 amplitude

```
amplitude <reaction> <sum> <amp> <class> (arg1) (arg2) (...)
```

The `amplitude` directive adds an amplitude factor named `amp` to the reaction and sum specified by the first two arguments. All factors with common values for the string `reaction`, `sum`, and `amp` will be multiplied together. The factor that is added to the amplitude is derived by executing the relevant user-defined functions in the class named `class`. The `class` must be registered with the `AmplitudeManager` in order to be used. This registration typically happens through execution of static function calls in the `AmpToolsInterface` that are made by the user code that executes the fit. The remaining arguments are passed to the constructor of the user-defined amplitude class and be used to configure the amplitude calculation, *e.g.*, if the amplitude calculates a Breit-Wigner function then the arguments may be mass and width of the of the function.

3.3.2 permute

```
permute <reaction> <sum> <amp> <index1> (index2) (...)
```

For the particular reaction, sum, and amplitude specified, the permute command forms the amplitude by adding additional permutations of particles specified by the index arguments and rescaling the entire sum by $1/\sqrt{N}$ where, N is the number of permutations. For example, if a user specifies a reaction that contains three particles: `reaction 3pi piPlus piMinus pi0` then the "default permutation" of the three particles is given by the index code `012`. Suppose now that the user desires to write an amplitude called `rhoPMpi0` (belonging to a sum called `sum`) that should be symmetric under the interchange of particle index 0 and 1 (the `piPlus` and `piMinus`). This symmetrization can be achieved with the command `permute 3pi sum rhoPMpi0 102`.

The AmpTools framework will automatically permute particles that have identical names in the `reaction` declaration. So if, for example, all amplitudes of the fit to the three pion final state described above are symmetric under the interchange of particle index 0 and 1 then the same functionality can be achieved by omitting the `permute` commands and declaring the reaction with the command `reaction 3pi piCharged piCharged pi0`. In this case, every amplitude will automatically be calculated with permutation index codes `012` (the default) and `102` and the results will be added and the sum dived by $\sqrt{2}$ to form the amplitude.

3.3.3 scale

```
scale <reaction> <sum> <amp> <value>
```

The `scale` keyword will set the scale factor for the amplitude, denoted by $s_{\sigma,\alpha}$ in Equation 2.5. The scale factor is a real number and it should be emphasize that it scales the amplitude not the amplitude squared. Therefore, one must take care when setting scale factors for various applications. The value may also be replaced by a free parameter.

3.3.4 constrain

```
constrain <reaction1> <sum1> <amp1> <reaction2> <sum2> <amp2> (<reaction3> <sum3> <amp3> ...)
```

The `constrain` keyword forces the corresponding $V_{\sigma,\alpha}$ (see Equation 2.5) to be equal. Imposing such constraints reduces the number of parameters in the fit. Multiple `constrain` lines are permitted, for example, amplitude A can be constrained to be equal B, and, in a separate line, amplitude B be constrained to be equal to C. This will result in one free parameter for all three amplitudes.

3.3.5 initialize

```
initialize <reaction> <sum> <amp> <"polar"/"cartesian"> <value1> <value2> ("fixed") ("real")
```

The `initialize` command is used to set the initial value for the production coefficient ($V_{\sigma,\alpha}$ in Equation 2.5) for a particular amplitude. In the case that the fourth argument is the word `cartesian` then the `value1` and `value2` are interpreted as the real and imaginary parts of the production coefficient. If the fourth argument is `polar` then `value1` is interpreted as the magnitude and `value2` is interpreted as the phase (in radians) of the production coefficient. Two final optional arguments can be used to either fix the value of the coefficient in the fit and/or, with the use of `real`, set the imaginary part of the of the production coefficient to zero.

To avoid an under-constrained fit, it is necessary that one amplitude in each sum be constrained to be real (or constrained to another amplitude in a different sum whose phase is constrained). The user may experiment with fitting using cartesian or polar coordinates. Polar coordinates have an obvious issue that one is guaranteed to have multiple minima spaced by 2π radians in phase. On the other hand, for amplitudes where the magnitude of the amplitude is highly constrained but the phase is not, polar coordinates tend to reduce the high degree of correlation between the two fit parameters that is typically observed when cartesian coordinates are used.

3.3.6 parameter

```
parameter <par> <value> ("fixed"/"bounded"/"gaussian") (lower/central) (upper/error)
```

The `parameter` command is used to declare a real-valued parameter that can be used as an argument to an amplitude class or as a scale factor. The name of the parameter is given by the first argument and the initial value is given by the second argument. When the user desired to use this parameter as a scale factor or as an amplitude class argument, the name of the parameter should be enclosed in square brackets (`[ ]`) to indicate that the string inside should be interpreted as the name of a parameter. A third optional argument indicates whether the parameter should be fixed, bounded, or constrained using a Gaussian constraint. In the latter case, the value of $-2\ln\mathcal{L}$ is incremented by an amount equal to $(x-\mu)^2/\sigma^2$, where x is the value of the parameter and μ and σ are given by the `central` and `error` arguments, respectively. This type of constraint is useful for incorporating external data. For example, one may want to all the mass of a resonance to float, but impose a constraint based on an independent external measurement of the mass of the resonance. If one chooses `bounded` the bound is enforced by the minimization package, which is currently MINUIT.

3.4 Specifying Input Files

In order to perform a fit, a user must provide a mechanism for reading in several types of input. All of the various inputs must be provided by a class that inherits from the `DataReader` class. See the detailed discussion in Chapter **??**. In the configuration file one needs to only list the names of the data readers and the arguments to be passed to the data reader. In the simplest form the user-provided class reads from a file and the arguments passed to the data reader include the name of the file. The AMPTOOLS framework is agnostic when it comes to format and storage mechanism of data. It only requires that the user provide a class that can supply the lists of four-vectors that represent the events. In all cases the data reader classes are specific to the type of data and specific reaction in the fit, which allows reading data from multiple forms for different data types and different reactions.

3.4.1 data

```
data <reaction> <class> (arg1) (arg2) (...)
```

The `data` keyword tells AmpTools to read what is called the data, *i.e.,* the events to which one is typically trying to fit intensity, for a given reaction from the user-provided class named `class`. The remaining arguments are passed in as a vector of strings to the user-defined class and they can be used to control the class behavior.

3.4.2 genmc

```
genmc <reaction> <class> (arg1) (arg2) (...)
```

The `genmc` keyword tells AmpTools to read the generated Monte-Carlo sample for a given reaction from the user-provided data reader called `class`. The remaining arguments are passed in as a vector of strings to the user-defined class and they can be used to control the class behavior. The generated MC sample is used to compute the value of M_g that appears in Equation 2.10 and also to compute the "amplitude integrals" discussed briefly in the text that follows that equation. Specifying the generated Monte-Carlo sample is optional if the user provides a complete normalization integral file using the `normint` command that is discussed below *and* the only free parameters in the fit are production coefficients of the amplitudes.

3.4.3 accmc

```
accmc <reaction> <class> (arg1) (arg2) (...)
```

The `accmc` keyword tells AmpTools to read the accepted Monte-Carlo sample for a given reaction from the user-provided data reader called `class`. The remaining arguments are passed in as a vector of strings to the user-defined class and they can be used to control the class behavior. The events provided by this class should be a subset of the generated sample that pass all analysis requirements. The sample is used to compute the normalization integrals in Equation 2.10. Specifying the accepted Monte-Carlo sample is optional if the user provides a complete normalization integral file using the `normint` command that is discussed below *and* the only free parameters in the fit are production coefficients of the amplitudes.

3.4.4 normint

```
normint <filename> ("input")
```

The `normint` keyword is used to control the input or output of the calculation of both the "normalization integrals" and the "amplitude integrals" that are discussed in Equation 2.10. In the case that the optional `input` flag is omitted, the integrals will be written the file named `filename`. The user may then read the integrals back from the file using the `input` flag. The computation of integrals using the generated and accepted MC samples can be a resource-intensive activity that consumes large amounts of memory and CPU. If a user is doing a fit in which none of the amplitudes have free parameters, then, as long as the analysis criteria that define the accepted sample do not change the normalization integrals will not change. It can be advantageous in this case to store the integrals after the first fit and utilize them for subsequent fits. Normalization integrals are indexed by amplitude in the file. One needs to be sure that all amplitudes in the fit have a corresponding integral present in the file, but the file can contain more amplitudes than are used in the fit. This makes it possible to have a master file of integrals that can be used for multiple fits, each of which may use different subsets of amplitudes.

3.4.5 bkgnd

```
bkgnd <reaction> <class> (arg1) (arg2) (...)
```

The `bkgnd` keyword specifies the user-defined data reader `class` that will be provide a sample of events that is a representation of the background that is in the set of events provided by the `data` keyword for the

given `reaction`. This sample should be weighted such that the weighted sum represents the distribution of background in the signal region, and the sum of the weights of the sample is equal to the expected background in the sample specified by the `data` keyword. The method for using this sample in the in the likelihood fit is discussed in detail in Section 2.4.3.

3.5 Design Comments

The role of the configuration file is to enable the `ConfigFileParser` class to configure the fit by creating an object called `ConfigurationInfo.` While we extensively document the syntax of the configuration file in the manual, it is possible for a developer to work with the `ConfigurationInfo` object directly. In this case one would use the member functions of `ConfigurationInfo` to create amplitudes, sums, and reactions, constrain parameters, *etc.* Consult the header file `ConfigurationInfo.h` for a comprehensive list of member functions and nested classes that provide and store information for the various components. For a developer this may have the advantage of working in the context of a programming language that permits complex looping structures and other constructions without having to write out a text file. The `ConfigurationInfo` object supports the `operator<<` so it can be displayed as text as needed.

Chapter 4

User Programming Interface

In order to facilitate use of the AmpTools package several classes have been written with the goal of simplifying the programming interface for the user. Ultimately, the user will need to write executable code to perform a fit or do other tasks. One should be reminded that fully-functional examples of these pieces of code exist in the `Dalitz` tutorial that is distributed inside of the `Tutorials` directory with AmpTools. Two key interface classes are provided: the `AmpToolsInterface` and the `FitResults` class. The first manages the objects necessary to execute a fit, make projections of results, or generate Monte Carlo. The second is purely an interface to the result of a fit and provides manipulations of parameters, error matrices, and other useful outputs.

4.1 AmpToolsInterface

It is suggested the user review the header file `AmpToolsInterface.h` or the associated doxygen-based documentation in order to examine the detailed function calls available in the class. Here we provide a brief overview of the how the class functions.

4.1.1 Registering User Classes

Before the user can construct an interface object, the data readers and amplitudes that user intends to use must be registered by the user. This is done through a set of static member functions

- `AmpToolsInterface::registerAmplitude( const Amplitude& defaultAmplitude )`, and
- `AmpToolsInterface::registerDataReader( const DataReader& defaultDataReader )`.

The user calls these functions once for every data reader and amplitude factor that the user intends to use in the application. The arguments to these functions are typically the user-defined classes constructed with the default constructor. These default classes are guaranteed to have two essential features that the interface utilizes: they can return their name in the form of a string and they can create new specialized instances of themselves using arguments. The `AmpToolsInterface` can then find the registered amplitude by name provided in the `ConfigurationInfo` and construct as many amplitude as necessary with the arguments provided in the `ConfigurationInfo`. A similar model is employed for the data reader classes, which also have the ability to return their name and construct new specialized instances of `DataReader` classes. It is important that any data readers or amplitudes that the user desires be registered first otherwise the construction of an `AmpToolsInterface` object from a `ConfigurationInfo` object will likely fail because the `AmpToolsInterface` is unable to create to the specialized amplitudes or data readers that user desires. There is no problem with registering more data readers or amplitudes than are used, so in complex environments one may want to register a whole list of objects. Note that registration of the object will require that the library or code that defines the object be linked at the time the executable is compiled.

4.1.2 Constructing an Interface Object

Typically the `AmpToolsInterface` should be constructed by passing in a pointer to a `ConfigurationInfo` object and an optional functionality flag. The `ConfigurationInfo` object stores the configuration of the fit or the recipe for constructing the intensity in terms of amplitudes, all of which should have been previously registered. The functionality flag must be one of the three arguments specified by the `enum FunctionalityFlag` in the header file. The three arguments and their corresponding functions are described below.

- `kMCGeneration`: This is intended to create an interface suitable for generating Monte Carlo or doing low level debugging of data reader or amplitude classes. The intensity is constructed according to the specifications in the `ConfigurationInfo` object. The functionality of the resulting interface is limited to being able to calculate intensities for individual events. The user can load events into the interface and process intensity calculations for the events. The interface ignores specification of data files in the `ConfigurationInfo`.

- `kPlotGeneration`: Constructing the interface with this option provides the functionality above, plus the interface initializes the data readers specified in the `ConfigurationInfo`. The `PlotGenerator` base class, from which the user can derive specialized plotting classes, constructs and utilizes the `AmpToolsInterface` in this mode. It is typically not required for the user to utilize this mode of operation, but instead just write a specialized plot generator class that can be used to create plots.

- `kFull`: This results in fully functional interface. In particular it creates class to manage calculation of the likelihood and the interface to the minimizer. *This is the default flag in the case that one is not is specified.*

Once the object is constructed it can be reconfigured using the method `resetConfigurationInfo( ConfigurationInfo* )`, which if functionally equivalent to destroying the object and recreating it with the same functionality flag, but using a different configuration.

4.1.3 Performing a fit with the `MinuitMinimizationManager`

After registering amplitudes and data readers, then constructing an `AmpToolsInterface` object, the user can the use the interface to perform a fit. Access to fitting is done through the `minuitMinimizationManager()` function, which returns a pointer to a `MinuitMinimizationManager` object. The user can use this object to tailor the behavior of the fit. Below are a few of the key functions that are provided by the `MinuitMinimizationManager`.

- `setPrecision( double )`: This sets the expected machine precision using the `SET EPS` command in MINUIT. Absence of this call will result in MINUIT trying to determine machine precision, which is typically satisfactory. However, in GPU accelerated fits or in parallel environments, sometimes the precision that is maintained in the parallel computation of the likelihood is worse than machine precision determined by MINUIT. This may result in unstable behavior as fluctuations in the likelihood calculation due to precision effects become interpreted by MINUIT as real variations in the likelihood with changing parameters. The user may want to start with automatically determined precision and only experiment with this quantity only if instability is observed. Note that changes in $-2\ln\mathcal{L}$ of order unity are significant when evaluating uncertainties on parameters. Naively one expects that he precision needs to be several orders of magnitude less than unity for the fit to function well. One may compare this scale with the typical value of the likelihood to understand if issues with precision are likely to emerge.

- `setStrategy( int )`: This allows one to change the MINUIT strategy. What follows is directly quoted from the MINUIT manual. "In the current release, this parameter can take on three integer values (0, 1, 2), and the default value is 1. Value 0 indicates to Minuit that it should economize function calls; it is intended for cases where there are many variable parameters and/or the function takes a long time

to calculate and/or the user is not interested in very precise values for parameter errors. On the other hand, the value 2 indicates that Minuit is allowed to waste function calls in order to be sure that all values are precise; it is intended for cases where the function is evaluated in a very short time and/or where the parameter errors must be calculated reliably."

- `setMaxIterations( int )`: This changes the maximum number of iterations for MINUIT minimization operations. The default value is 5000.

After configuring the behavior of the minimization routine, then the typically the user wants to run the minimizer. The `MinuitMinimizationManager` provides two methods for this, `migradMinimization()` and `minosMinimization`, where these invoke the standard MINUIT routines known as MIGRAD and MINOS. After running one of the two previous minimization algorithms, the user may like to know the status of the fit. The functions noted above have no return value, but the status of the function call can be evaluated by using the `status()` member function of the `MinuitMinimizationManager`. A complete documentation of the `status()` return values is in `MinuitMinimizationManager.h`, but the two values that the user is most likely to see are a value of 0, which means the minimization terminated normally, or a value of 4, which indicates that the minimization did not converge. In addition one can check the status of the error matrix using the function `eMatrixStatus()`, which will return one of the following values.

0 : the error matrix has not been calculated;

1 : the error matrix has been approximated and is not accurate;

2 : the full error matrix has been calculated but has been forced to be positive definite; or

3 : the full accurate covariance matrix has been calculated.

Normally the user will desire that fitting terminate such that the `status()` function returns a value 0 and the `eMatrixStatus()` function returns a value of 3. For the current configuration of the fit parameters, the Hessian matrix (the matrix of second derivates and also the inverse of the error matrix) can be obtained by using the `hesseEvaluation()` method, which returns a vector of vectors of doubles, which can be indexed like a two-dimensional matrix. Once the user is pleased with the exit status of the minimization algorithm, then the user should call the `finalizeFit()` function that is a member of the `AmpToolsInterface` object. This method will write the output of the fit to a log file specified by the name of the fit as listed in the `ConfigurationInfo` object.

4.1.4 Using low-level `AmpToolsInterface` functions

In order to manage the various data sets associated with a fit, the `AmpToolsInterface` must have the ability to load and store data and calculate intensities for individual events in the data sample. It makes sense to make this functionality available to users of the interface for a variety of operations. The most common use case is for Monte-Carlo generation, but that methods are also useful for debugging operations such as verifying that events are properly loaded by a data or that individual amplitudes are properly calculated.

The exact method and their arguments are best referenced by examining the documentation in the `AmpToolsInterface.h` file, but here we summarize the functionality. The interface supports the notion of multiple data sets and many low level calls will take an optional last argument that defines the data set index. Events associated with a data set can be cleared from the interface. They can also be loaded either from a `DataReader` class or through an array of `Kinematics` objects.

Once a particular data set is loaded into the interface the user can calculate the amplitudes associated with a particular reaction using the data set as input. This is done using the `processEvents( string reactionName )` method. Here `reactionName` corresponds to the particular reaction in `ConfigurationInfo` object with which the interface was constructed that contains the amplitudes of interest. The `processEvents` function takes an optional data set index. The method loops over the loaded data and calculates the various amplitudes within the reaction of interest that have been specified. After the loaded events have been

processed, one may obtain the intensity on an event-by-event basis, which is a useful operation when writing an accept/reject Monte-Carlo generation algorithm.

Finally, for low-level debugging, there are a number of methods to print amplitudes, intensity, and kinematics for individual events, which can be passed to the method using a `Kinematics` pointer. Again, see the header file for call structure. These functions are particularly useful for debugging calculations of amplitudes and intensities as it tests the full framework in the way that a fit does and uses many of the same objects used to calculate the likelihood.