April 28, 2017
version 2.0

GammaCombo User Manual

Till Moritz Karbach[1], Matthew William Kenzie[1], Maximilian Schlupp[2], Konstantin Schubert[2]

[1] *European Organization for Nuclear Research (CERN), Geneva, Switzerland*
[2] *Technische Universität Dortmund, Dortmund, Germany*

GammaCombo is a framework to combine measurements in order to compute confidence intervals on parameters of interest. A global likelihood function is constructed from the probability densitiy functions of the input observables, which is used to derive likelihood-based intervals and frequentist intervals based on pseudoexperiments following the Plugin method to handle nuisance parameters.

Contents

1 Introduction

The interpretation of observables in terms of parameters of the underlying theory is a common problem in physics. Well established methods exist on how to do that, but in practice, their application is often found to be cumbersome. GammaCombo is a framework that hopes to turn this process into a more pleasurable experience.

The basic principle is to assume a probablility density function (PDF) for the observables, and form a likelihood function. From these, best-fit values as well as confidence intervals or upper limits can be computed, following two staticical methods: the profile likelihood construction and the Plugin method. Both methods are located at the more frequentist-like side of the spectrum of statistical schools.

The strength of GammaCombo is to provide tools helping in the following areas:

- provide a framework to construct the likelihood function
- implement minimization strategies to aid the profiling of nuisance parameters
- provide debug tools helping to identify problems during profiling
- make publication quality plots, in both one and two dimensions
- provide a way to run large scale toy experiments on a batch system
- provide debug tools helping to judge the quality of frequentist toys
- combine likelihood functions corresponding to different measurements

While being able to save the user a lot of tedious work, one of the design choices is to not invent a black box that is just too dark. After all, applying statistical methods is, in most cases, not trivial and requires a detailed understanding of what is going on. On the downside, this requires the user to code in `C++`.

GammaCombo is able to run state-of-the art sized combinations of measurements. An example of such combinations is LHCb's combination of measurements that are sensitive to the CKM angle γ [1].

2 Statistical methods

2.1 Basic Principle

The strategy to combine the input measurements described in the previous sections is to build a combined likelihood function. This combined likelihood is the product of the individual likelihoods. Denoting the observables by capital Latin letters, and parameters by Greek letters, the combined likelihood $\mathcal{L}$ is

$$\mathcal{L}(\vec{\alpha}) = \prod_i \mathcal{L}_i(\vec{\beta}_i) \ , \tag{1}$$

where the product runs over all input measurements, the vectors $\vec{\beta}_i$ hold the parameters of the input measurements, the vector $\vec{\alpha}$ holds all parameters of the $\vec{\beta}_i$, and the $\mathcal{L}_i$ are the input likelihoods. In this context, the likelihood and the probability density function (PDF) are very similar. The PDF, say f, is a function not only of the parameters $\vec{\beta}$, but also of the observables $\vec{A}$:

$$f = f(\vec{A}|\vec{\beta}) \ . \tag{2}$$

The likelihood is obtained by fixing the observables to the measured values. If multiple measurements of the same kind were performed, the likelihood is the product of as many PDF terms. But in the context of the combination, each measurement was performed exactly once, thus the distinction is subtle,

$$\mathcal{L}(\vec{\beta}) \equiv f(\vec{A}|\vec{\beta})\Big|_{\vec{A}=\vec{A}_{\rm obs}} \ . \tag{3}$$

To give an example, lets consider the combination of two measurements. The first is a cartesian measurement, which measures four cartesian observables that are related to polar coordinates through

$$x_\pm = r_B^{DK} \cos(\delta_B^{DK} \pm \gamma) \tag{4}$$
$$y_\pm = r_B^{DK} \sin(\delta_B^{DK} \pm \gamma) \, . \tag{5}$$

The second measurement is a direct measurement of the parameter r_B^{DK},

$$(r_B^{DK})_{\rm obs} = r_B^{DK} \ . \tag{6}$$

Therefore the vector of parameters $\vec{\alpha}$ has the following components:

$$\vec{\alpha} = (\gamma, r_B^{DK}, \delta_B^{DK})^T \ , \tag{7}$$

where T means transpose. The purpose of the combination is to find the best values of the components of $\vec{\alpha}$, defined as those that maximize the combined likelihood, and to compute well-defined error intervals. The vector of observables, $\vec{A}$, has the following form:

$$\vec{A} = (x_-, y_-, x_+, y_+, (r_B^{DK})_{\rm obs})^T \ . \tag{8}$$

The components of $\vec{A}$ are related to those of $\vec{\alpha}$ by the truth relations Eqns. 4 and 6. We can now construct the combined likelihood as

$$\mathcal{L}(\vec{\alpha}) = f(\vec{A}_{\rm obs}|\vec{A}(\vec{\alpha})) \ , \tag{9}$$

where $\vec{A}_{\rm obs}$ holds the measured results, and $\vec{A}(\vec{\alpha})$ are the truth relations. In case of uncorrelated observables, this factors into

$$\mathcal{L}(\vec{\alpha}) = \prod_i f_i(\vec{A}_{i,\rm obs}|\vec{A}_i(\vec{\alpha}_i)) \ . \tag{10}$$

To obtain confidence intervals, a first and simple approach is to inspect the profile likelihood curve, which is described in Section 2.2. This has the advantage of being computationally cheap, but in many examples comes at the price of bad frequentist coverage. A better method is to rely on a method based on pseudo experiments, the PLUGIN method, which is described in Section 2.3. This method is much more computing expensive. It builds upon the profile likelihood method, so in any case it does make sense to start with the easier PROB method.

2.2 The profile likelihood method (PROB)

We define a χ^2-function as

$$\chi^2(\vec{\alpha}) = -2\ln\mathcal{L}(\vec{\alpha}). \tag{11}$$

In general, there are many equivalent global minima, reflecting the fact that there are multiple solutions possible. At each of these minima, the χ^2 has a value $\chi^2_{\rm min}$. To evaluate the confidence level of a certain truth parameter (for example γ) at a certain value (γ_0) we consider the value of the χ^2-function at the new minimum, $\chi^2_{\rm min}(\vec{\alpha}')$, where the γ component of $\vec{\alpha}'$ is fixed to $\gamma = \gamma_0$. The new minimum satisfies $\Delta\chi^2 = \chi^2_{\rm min}(\vec{\alpha}') - \chi^2_{\rm min} \geq 0$. In a purely Gaussian situation for the truth parameters, the p value $p \equiv 1 - \mathrm{CL}$ is given by the probability that $\Delta\chi^2$ is exceeded for a χ^2-distribution with one degree of freedom:

$$1 - \mathrm{CL} = \frac{1}{\sqrt{2}\,\Gamma(1/2)} \int_{\Delta\chi^2}^{\infty} e^{-t/2} t^{-1/2}\, \mathrm{d}t. \tag{12}$$

Because Eq. (12) can be evaluated using the `TMath::Prob()` function,

$$1 - \mathrm{CL} = \mathrm{Prob}(\Delta\chi^2, n_{\rm dof} = 1) \ , \tag{13}$$

this method is sometimes called the “PROB method”, for example described in the PDG booklet [2]. It is equivalent to the profile likelihood method, implemented e.g. in the MINOS method of MINUIT. The confidence intervals are obtained from the $1 - \mathrm{CL}$ curves by finding the point where it intersects the $1 - \mathrm{CL} = 1 - 0.68$ and $1 - 0.95$ levels. These points are found by interpolating between the four nearest scan points. The PROB method is known to undercover in many situations, that is, the reported intervals contain the true value in a lesser number of times than claimed. However it can also overcover, in which case the reported intervals are too large.

2.3 The pseudo-experiment based frequentist method (Plugin)

A more accurate approach is to not rely on the linearity of the truth relations, that relate the observables to the parameters, for example Eqns. 4 and 6. Instead on assuming that the test statistics $\Delta\chi^2$ is distributed as a χ^2, one should compute its distribution using pseudo experiments. This method is based on the approach by Feldman and Cousins [3], and extends it by introducing the concept of nuisance parameters. In the "Plugin" method [4] (sometimes also "$\hat{\mu}$ method"), the nuisance parameters are kept, at all times, at their best fit values determined by the observed data, that is they are kept at their profile likelihood point. The algorithm of the method is equivalent to the algorithm propsed by Feldman and Cousins in the absence of nuisance parameters [5]. It is briefly described in the following. For a certain value of interest (γ_0), we:

1. calculate $\Delta\chi^2 = \chi^2_{\rm min}(\vec{\alpha}') - \chi^2_{\rm min}$ as before;
2. generate a "toy" result $\vec{A}_{\rm toy}$, using Eq. 3 with parameters set to $\vec{\alpha}'$ as the PDF;
3. calculate $\Delta\chi^{2\prime}$ of the toy result as in the first step by replacing $\vec{A}_{\rm obs} \to \vec{A}_{\rm toy}$, *i.e.* minimize again with respect to $\vec{\alpha}$, once with the scan parameter floating and once with the scan parameter fixed;
4. calculate $1-\mathrm{CL}$ as the fraction of toy results which perform worse than the measured data, *i.e.*
$$1 - \mathrm{CL} = N(\Delta\chi^2 < \Delta\chi^{2\prime})/N_{\rm toy}\ . \tag{14}$$

It has better coverage properties than the Prob method, but also not necessarily a perfect coverage. The reason is, that at each scan step the nuisance parameters (components of $\vec{\alpha}$ other than the scan parameter) are plugged in at their best fit values for this step, as opposed to computing an n-dimensional Neyman confidence belt. In our simple example we have already $n = 3$, but situations with many more parameters are very common. This would render a fine grained scan of the parameter space computationally very expensive.

In our experience the intervals obtained using the Plugin method are usually within 10%–20% of those obtained using the Prob method. This means there are only few cases where qualitative statements obtained using the Prob method become obsolete when considering the Plugin results.

3 Installation

The installation of GammaCombo requires the version management system `git` and the build system `cmake` to be installed on your system. It also requires an installation of `Root`, compiled with the `RooFit` library, and a C++ compiler recent enough to support C++11. See a full list of dependencies in Tab. 1. The source code is hosted on `github.com`. The following steps install GammaCombo on your system.

```
# get and build gammacombo
git clone https://github.com/gammacombo/gammacombo.git
cd gammacombo
# if you want to work with the most recent version of the code,
# switch to the development branch:
#git checkout development
mkdir build
cd build
cmake ..
make # can use -j4 to use 4 CPU cores
make install
```

In addition to the GammaCombo core library, this also builds an example combiner called tutorial. It is also possible to compile with debug symbols:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

To run a first combination, try the following:

```
# run the tutorial
cd ../tutorial
bin/tutorial -u # usage
bin/tutorial -c 1 --var a_gaus -i # run combination 1
ls -l plots/pdf/tutorial_tutorial1_a_gaus.pdf # the plot that was created
```

More on the tutorial is described later in Sec. 4. The GammaCombo source code is documented with `doxygen`. You can build the `doxygen` web page documentation, if you have `doxygen` installed, and probably also `graphviz`'s `dot` command. The documentation is created in the `doc` subdirectory of the top level directory.

```
# build the doxygen documentation
# still in the build directory
make doc
# open the created webpages on Mac:
cd ..; open doc/html/index.html
```

Table 1: Dependencies of GammaCombo on libraries and software packages. It is not excluded that it can run with older versions as well, however these are the versions GammaCombo has been tested with.

dependency	minimum version	details
`ROOT`	5.34.23	GammaCombo is able to run also with previous versions, but it has been thorougly tested with this version. `ROOT` needs to be compiled to include `RooFit`, for example using the `--enable-roofit` option in the configure step
`RooFit`	3.60	see `ROOT`
`boost`	1.57.0	
`Doxygen`	1.8.2	optional

3.1 Installation of subpackages

Each separate combination requires a new subpackage which exists in its own directory. The example given in this manual is a subpackage called "tutorial".

There are several combinations implemented by the GammaCombo code which each have their own subpackage. These are listed in Table 2. Some of these contain private data which has not yet been published or is not intended to be public. Consequently each subpackage is held in a private location on a `hepforge` server and only certain users are allowed access to them.

If you require access to the subpackages then you will need to follow these steps:

1. Inform us you require access by emailing us (gammacombo-owner@projects.hepforge.org) with your name and a justification.
2. Register as a hepforge user citing your reasons for joing as a GammaCombo user / developer. You can do so at this page. Please also sign up to our mailing list here so we can keep you informed of updates, downtime, changes etc.
3. You should be able to browse subpackages on the web in the `git-private` area of GammaCombo on `hepforge` which can be found here.

To install a subpackage on your system you can do the following:

```
cd gammacombo # the location you installed the main package above
# set useful environment variable
HEPFORGE=<hepforge_user_name>@login.hepforge.org
# clone the package
git clone \$HEPFORGE:/hepforge/git/gammacombo/private/<sub_package_name>
# edit cmake/combiners.cmake file to include the sub_package_name
cd build
```

```
cmake ..
make # can use -j4 for 4 CPU cores
make install
```

To run the subpackage follow steps as per the tutorial subpackage in the rest of this document. For example to print the usage:

```
# to check that it runs ok
cd ../<sub_package_name>
bin/<sub_package_name> -u # print the usage
```

Table 2: A list of GammaCombo subpackages and their uses

Subpackage name	Details
`gammacombo`	Combination of LHCb analyses which measure or constrain the CKM angle γ. *Note that this has the same name as the main package but only for historical reasons*
`biggammacombo`	Combination of analyses which measure or constrain B_s mixing parameters, ϕ_s, Γ_s, $\Delta\Gamma_s$.
`hfag`	Combination of charm mixing averages
`belle2_Vub`	Combination of constraints on CKM parameter V_{ub}
`Vub_Vcb`	Combination of constraints on CKM parameters V_{ub} and V_{cb}

4 Tutorial

The tutorial consists of three parts: a simple Gaussian, a two-dimensional Gaussian, and a circular PDF that puts a constraint on a radius parameter. Lets run the tutorial executable (`main/tutorial.cpp`) to get an overview of what measurements and combinations are defined in the tutorial. (The other exectuables in the `main/` folder are not part of this tutorial. They can be ignored.)

```
$ cd tutorial
$ bin/tutorial -u # -u always prints the usage and exits

AVAILABLE MEASUREMENTS

   (1) 1D Gaussian (a_{obs} = -0.5)
   (2) 1D Gaussian (a_{obs} = 1.5)
   (3) 2D Gaussian (a_{obs}, b_{obs})
   (4) circle (a_{obs}, b_{obs})

AVAILABLE COMBINATIONS

   (0) empty
   (1) Gaus 1
   (2) Gaus 2
   (3) Gaus 1 & Gaus 2
   (4) 2D Gaus
   (5) 2D Gaus & Gaus 1
   (6) Circle
   (7) 2D Gaus & Circle
   (8) Gaus 1 & Gaus 2 & 2D Gaus
```

4.1 Simple Gaussian measurement

We start with the probably most simple case of all: a theory parameter being constrained through a single measurement, that follows a simple Gaussian PDF. To find the best estimate for the theory parameter a_{th} the following likelihood function is maximized:

$$\mathcal{L}(a_{\mathrm{th}}) = G(a_{\mathrm{obs}}, \sigma_{a_{\mathrm{obs}}}, a_{\mathrm{th}}) = \frac{1}{\sigma_{a_{\mathrm{obs}}}\sqrt{2\pi}} \exp\left(\frac{-(a_{\mathrm{obs}} - a_{\mathrm{th}})^2}{2\sigma^2_{a_{\mathrm{obs}}}}\right) . \tag{15}$$

Here, the result of the measurement is denoted as

$$a_{\mathrm{obs}} \pm \sigma_{a_{\mathrm{obs}}} , \tag{16}$$

and the numerical values used in the tutorial's measurement 1 are

$$a_{\mathrm{obs}} = -0.5 \pm 1.0 . \tag{17}$$

The resulting confidence intervals for a_{th} are computed with

```
bin/tutorial -c 1 -i --var a_gaus --ps 1
```

where the options mean

`-c` combination ID. This refers to pre-defined combinations (in this case nothing is combined, really, it is just the simple Gaussian). A list of available IDs can be obtained by running with the usage flag, `bin/tutorial -u`.

`-i` interactive mode. Any plot will be shown directly in the familiar ROOT canvas. Exit with `Ctrl+c`.

`--var` the variable name of the theory parameter to be scanned.

`--ps 1` print solution on the plot. The given value configures the position of the numerical value on the plot. See the help `bin/tutorial -h` for options.

The resulting plot is shown in Fig. 1.

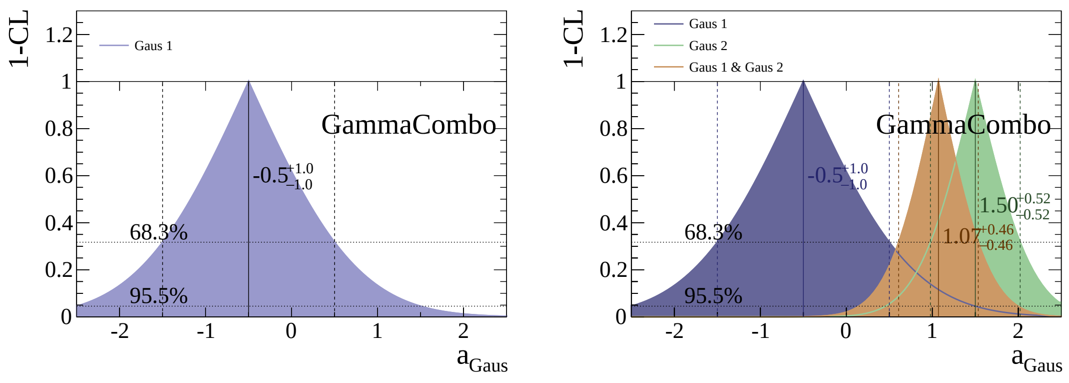

Figure 1: Left: $1 - \text{CL}$ plot resulting from tutorial measurement 1, a simple Gaussian. Right: Curves from tutorial measurements 1 and 2, together with the result of combining both (Sec. 4.2).

4.2 Combining two Gaussian measurmeents

In the tutorial executable, there is also a second Gaussian defined, measurement 2, which constrains a_{obs} to a different value,

$$a_{\text{obs}} = 1.0 \pm 0.5\,(\text{stat}) \pm 0.15\,(\text{syst})\,. \tag{18}$$

The combination of Eq. 17 and 18 should reproduce the usual weighted average,

$$a_{\rm th} = \frac{w_1 a_1 + w_2 a_2}{w_1 + w_2} \tag{19}$$

$$\sigma_{\rm th} = \frac{1}{\sqrt{w_1 + w_2}} \tag{20}$$

$$w_i = \frac{1}{\sigma^2_{a_i}}\,, \tag{21}$$

where the a_i refer to Eq. 17 and 18, and the statistical and sytematic uncertainty of Eq. 18 was combined in quadrature. The numerical result of Eq. 19 is

$$a_{\rm th} = 1.07 \pm 0.46\,. \tag{22}$$

Lets put them both into the same plot, and then also plot the combination of both. The resulting plot is shown in Fig. 1.

```
# multiple instances of -c will be added to the same plot
bin/tutorial -c 1 -c 2 -i --var a_gaus
# combination number 3 combines both Gaussians
bin/tutorial -c 1 -c 2 -c 3 -i --var a_gaus --ps 1
```

Options used:

`-c 1 -c 2 -c 3` When the combination argument is given multiple times, all combinations will be computed and added to the same plot. The order controls the order in the plot, where the last one is in the foreground.

4.3 Two-dimensional Gaussian measurement

The next measurement of the tutorial, measurement 3, contains a two-dimensional Gaussian PDF in the previous variable, a, and another one, b. Now the likelihood function is given by

$$\begin{aligned}\mathcal{L}(a_{\rm th}, b_{\rm th}) &= G(a_{\rm obs}, \sigma_{a_{\rm obs}}, a_{\rm th}, b_{\rm obs}, \sigma_{b_{\rm obs}}, b_{\rm th}) \\ &= \frac{1}{\sqrt{\det V}(\sqrt{2\pi})^{n/2}} \exp\left(-\frac{1}{2}(\vec{x}_{\rm obs} - \vec{x}_{\rm th})^T V^{-1} (\vec{x}_{\rm obs} - \vec{x}_{\rm th})\right)\,,\end{aligned} \tag{23}$$

where $\vec{x} = (a_{\rm obs}, b_{\rm obs})$ is the vector of observables, n is the number of observables, and V is the covariance matrix. Combination number 4 contains only this two-dimensional measurement, the numerical values of the observations are

$$a_{\rm obs} = 0.1 \pm 1.0\,, \tag{24}$$

$$b_{\rm obs} = 1.5 \pm 1.0\,, \tag{25}$$

$$\rho(a, b) = 0.6\,. \tag{26}$$

The resulting plots are shown in Fig. 2.

```
# the 2D scan is triggered by giving a second --var argument
bin/tutorial -c 4 --var a_gaus --var b_gaus -i
# redo the scan to print 2D confidence contours and a
# marker at the best fit value
bin/tutorial -c 4 --var a_gaus --var b_gaus -i --npoints 70 --2dcl --ps 1
# just remake the plot (without redoing the scan), and
# activate magnetic plot boundaries
bin/tutorial -c 4 --var a_gaus --var b_gaus -i --2dcl --ps 1 --magnetic -a
    plot
```

The command line options mean:

`--var a_gaus --var b_gaus` specify the two variables to perform the scan for. The variable given first will be plotted on the horizontal axis, the second one on the vertical axis.

`--2dcl` plot contours corresponding to “two-dimensional” confidence level. The default is to plot contours such that their projection to the axes coincides with the one-dimensional scans. In this case the resulting innermost contour corresponds to 39% CL. When the option is given, the innermost contour is enlarged to hold 68% CL.

`--npoints 70` use 70 scan points in each direction, so $70^2 = 4900$ in total.

`--ps 1` print a marker at all local minima. If `--ps 2` is given, only the best fit value is indicated.

`--magnetic` switches on magnetic plot boundaries, that will drag the contours towards them. In many cases, this results in nicer contours, but not always.

`-a plot` activates the plot action: Whenever a scan is done, the result is saved into a file in `plots/scanner`. This can be read in to remake the plot, without having to rerun the entire scan. With more complex combinations, this saves time.

While a two-dimensional scan is running in interactive mode (`-i`), a window opens illustrating the scan progress. The scan is done in a trajectory spiralling away from the start points. In this plot, the current $\Delta\chi^2$ is plotted. Any subsequent scan opens one more of these plots. If in a subsequent scan lower $\Delta\chi^2$ values are encountered, this will show up. At the end of the scan, one more window is opened showing the final $\Delta\chi^2$, from which the confidence contours are computed. Updating these control plots can consume a significant computing time for simple combinations, like the tutorial. In these cases, the program runs much faster non-interactively (without `-i`).

For more complex combinations, two-dimensional scans can take a long time, especially with a high granularity. For these cases, the `-a plot` option is very useful, which allows the plot to be redone without having to rerun the scan. This way, the look and feel can be changed: plot markers at best fit points, change titles, change colors, change the number of contours, activate magnetic boundaries, etc. When running with the `-a plot` option, the only other important options are `-c 6 --var a_gaus --var b_gaus`, all other ones are not needed to identify the correct file and can be omitted.

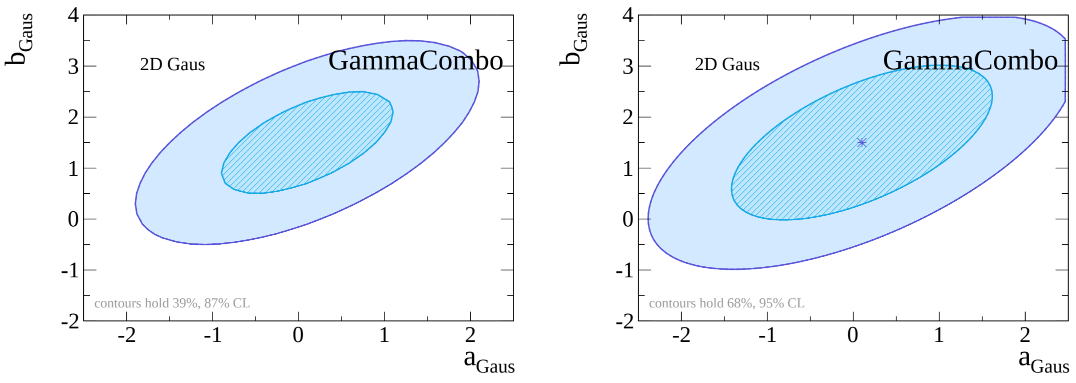

Figure 2: Result of tutorial 4, a two-dimensional Gaussian ellipse (Sec. 4.3). Left: with "1D" confidence regions. Right: with "2D" confidence regions and a marker at the best fit value.

4.4 A more advanced example: circular constraint

The next measurement of the tutorial is a circular constraint for the parameters a and b. This happens if we have a measurement of a radius-like parameter. In measurement 4 of the tutorial, a measurement of r is implemented,

$$r = 2.0 \pm 0.25\,, \tag{27}$$

which is related to the parameters a and b through the truth relation

$$r = \sqrt{a^2 + b^2}\,. \tag{28}$$

The constraint is shown in Fig. 3 (left). Note that when computing it (code below), it is likely that several best fit points are found. This is because all points on the circle have the same likelihood, and due to the numerical accuracy some get picked up. In the default configuration, a rescan is done for each local maximum of the likelihood is found—this might take a while in this case. Let's combine the circular constraint with the two-dimensional Gaussian measurement from the previous example. At first, both could be added to the same plot Fig. 3 (right). Then, we could combine the circular constraint with the two-dimensional Gaussian measurement and plot all three combinations together. This is shown in Fig. 4.

```
# scan the circular constraint of the tutorial
bin/tutorial -c 6 --var a_gaus --var b_gaus -i
# add the previous 2D Gaussian to the same plot
```

```
bin/tutorial -c 6 --var a_gaus --var b_gaus -a plot -i -c 4
# combine the circular constraint with the two-dimensional Gaussian
bin/tutorial -c 7 --var a_gaus --var b_gaus
# plot all three into the same plot
bin/tutorial -c 6 --var a_gaus --var b_gaus -a plot -i -c 4 -c 7
```

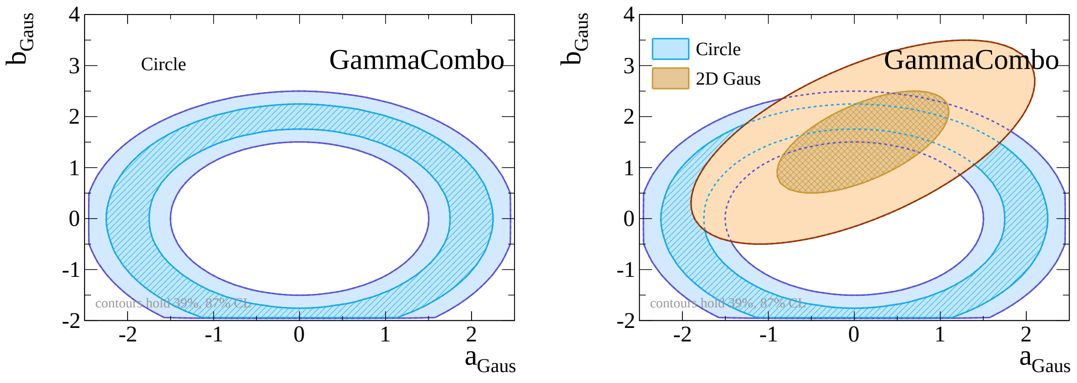

Figure 3: Result of the tutorial 6, a circular constraint (Sec. 4.4). Left: the circular constraint alone. Right: added the two-dimensional Gaussian from the previous example.

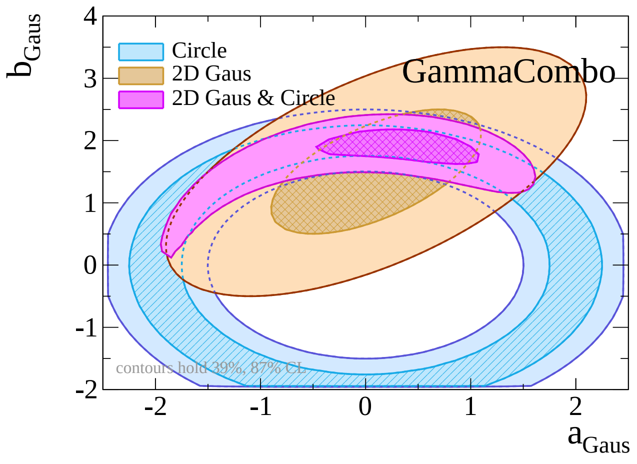

Figure 4: Result of the tutorial 6, a circular constraint (Sec. 4.4). Combining the circular constraint with the two-dimensional Gaussian measurement.

4.5 Changing combinations on the fly

It is possible to change existing combinations on the fly from the command line by adding and/or removing measurements. The measurements are refered to by their internal numbers, which can be obtained from the usage printout (`-u`).

Example 1: add a measurement Lets add a measurement on the fly to one of the tutorial combinations. The resulting plot is shown in Fig. 5.

```
# Lets start by running the unmodified combination 1,
# just consisting of measurement 1:
$ bin/tutorial -i --var a_gaus -c 1
# Now print out the usage message to find a measurement we can add:
$ bin/tutorial -u

AVAILABLE MEASUREMENTS

   (1) 1D Gaussian (a_{obs} = -0.5)
   (2) 1D Gaussian (a_{obs} = 1.5)
   (3) 2D Gaussian (a_{obs}, b_{obs})
   (4) circle (a_{obs}, b_{obs})

# Lets add number 2 to the combination:
$ bin/tutorial -i --var a_gaus -c 1:+2
# For comparison, lets add the unmodified combination to the same plot:
$ bin/tutorial -i --var a_gaus -c 1:+2 -c 1
```

Example 2: delete a measurement Lets delete a measurement on the fly from one of the tutorial combinations. The resulting plot is shown in Fig. 5.

```
# This is the unmodified tutorial combination 8,
# consisting of three measurements:
$ bin/tutorial -i --var a_gaus -c 8

Combiner Configuration: Gaus 1 & Gaus 2 & 2D Gaus
=======================

 1. [measurement 1] 1D Gaussian (A)
 2. [measurement 2] 1D Gaussian (B)
 3. [measurement 3] 2D Gaussian (A,B)

# Lets delete measurement 1 from the combination.
$ bin/tutorial -i --var a_gaus -c 8:-1 -c 8
```

Example 3: a new combiner from scratch An easy trick to make completely new combinations from the command line is to define an empty combination in the main file. In the tutorial, this combination has index 0. Then one can add whatever one wants.

```
# Adding several measurements to the empty combination
# to reproduce combination 8:
$ bin/tutorial -i --var a_gaus -c 0:+1,+2,+3
```

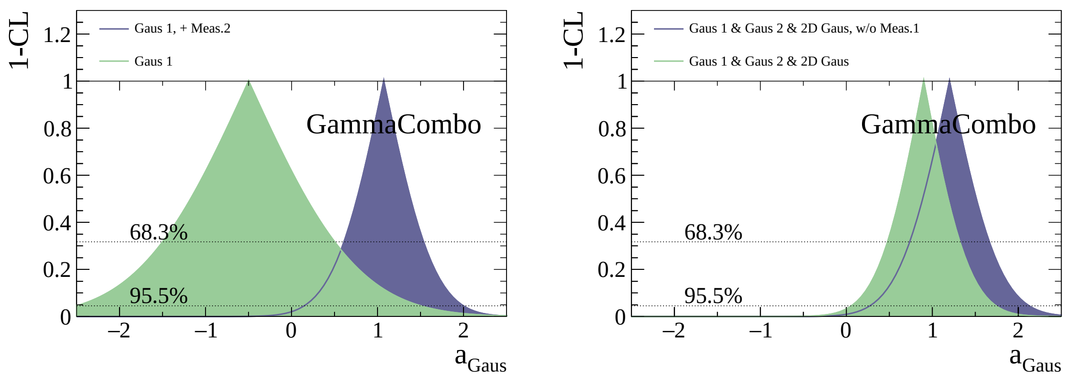

Figure 5: Result of Examples 1 and 2. Left: Add a measurement to an existing combination from the command line. Right: Delete a measurement from an existing combination.

4.6 Running the Plugin method

Running the pseudo-experiment based Plugin method for a particular combination consists of two steps:

1. generate and fit the pseudo experiments, possibly on a batch farm;
2. read in the fit results and compute the CL intervals.

Running the Plugin method requires that a profile-likelihood scan is already set up, because each batch job will recompute the profile likelihood. In practice this means one should have successfully made the Prob plot before, whithout running into problems. For the tutorial this is the case! Lets take as an example combination number 7 ("2D Gaus & Circle") and scan for parameter a. The Plugin result is shown in Fig. 6.

```
# Run three jobs to generate and fit toy experiments, possibly on
# different cores or batch nodes:
bin/tutorial -c 7 --var a_gaus -a pluginbatch --ntoys 100 --nrun 1
bin/tutorial -c 7 --var a_gaus -a pluginbatch --ntoys 200 --nrun 2
bin/tutorial -c 7 --var a_gaus -a pluginbatch --ntoys 400 --nrun 3

# Compute the plugin intervals from all toys:
bin/tutorial -c 7 --var a_gaus -a plugin -j 1-3 -i
# Only plot the plugin curve:
bin/tutorial -c 7 --var a_gaus -a plugin -j 1-3 -i --po
```

The arguments mean:

`-a pluginbatch` batch mode. This will run a number of Plugin toys and save them into an output file. Many of these jobs can be run in parallel for example on a batch farm.

`--ntoys 100` produce 100 PLUGIN toys per scan point.

`--nrun 1` run job number 1. This number will be added to the filenames of the produced PLUGIN toy files. Files will be overwritten if the same number is given.

`-j 1-3` read in PLUGIN toy files number 1 through 3.

`--po` make a PLUGIN-only plot.

The results in Fig. 6 show that, first of all, the PLUGIN method agrees quite nicely with the profile likelihood method. Probably it will yield slightly larger intervals, but in order to be conclusive, one would have to run more toys. Also, the result is visibly non-Gaussian, as it develops an asymmetric tail on the left. This was already apparent from the two-dimensional Fig. 4. In this case, however, the non-Gaussianity likely doesn't affect the statistical coverage, as both methods return quite similar results.

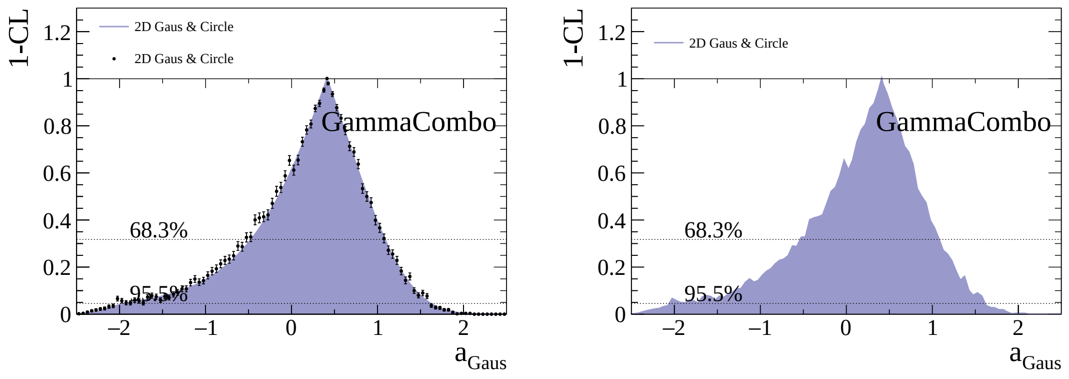

Figure 6: Result of running the PLUGIN method with combination number 7: a 2D Gaussian with a circular constraint. Left: the points are the result from the toys, while the filled curve corresponds to the PROB method. In this case, there is good agreement between both. Right: Plotting the result of the PLUGIN as a filled curve, without the PROB curve.

4.7 The standard Feldman-Cousins example

The presence of a physically forbidden region for a parameter complicates the statistical treatment. In particular, the profile likelihood construction is easily spoiled, when the boundary is close to the region of interest. In this situation, the method by Feldman and Cousins [3] provides an accepted frequentist alternative. In the absence of nuisance parameters, the PLUGIN method is equivalent to the method by Feldman and Cousins.

Lets take the example of the simple Gaussian measurement given in Sec. 4.1. The parameter a has a physical limit configured (see `src/ParametersTutorial.cpp`),

$$a > -1.5\,, \tag{29}$$

which has not been enforced so far. It can be activated through the physical range option `--pr`. The resulting plot is shown in Fig. 7 (left). Note that we are trying to scan in the non-physical region, which is currently not well supported: The drop at $a = -1.5$ is not vertical due to the finite binning, and as a result, the interpolated confidence interval is not accurate. This can be overcome by restricting the scan range to the physical region (Fig. 7 right).

```
# activate physical parameter limits
$ bin/tutorial -c 1 -i --var a_gaus --ps 1 --pr
# restrict scan range to the physically allowed region
$ bin/tutorial -c 1 -i --var a_gaus --ps 1 --pr --scanrange -1.5:2.5
```

The arguments mean:

`--pr` physical range. This will enforce the physical parameter ranges defined in `src/ParametersTutorial.cpp`.

`--scanrange -1.5:2.5` Adjust the scan range to the given range. The default range is defined in `src/ParametersTutorial.cpp`.

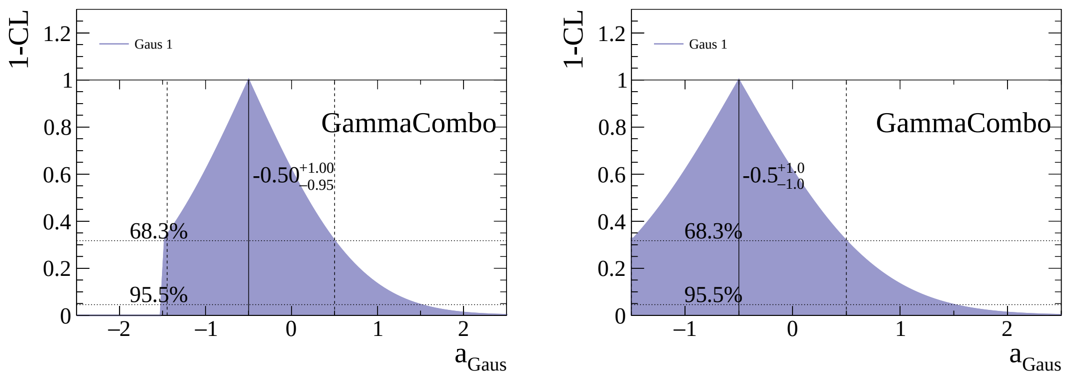

Figure 7: Applying a physical range to combination 1. Left: We tried to scan in the now unphysical region, which caues some inaccurracies (see text). Right: We restricted the scan range to the physical region, resulting in accurately deterimined (but statistically questionable) intervals.

Lets now run the Plugin toys for this example. The result is shown in Fig. 8. The difference to the profile likelihood method is most apparent close to the physical boundary: the boundary seems to "push away" the lower boundary of the confidence interval. The observed interval boundaries reproduce the nominal ones taken from the original Feldman-Cousins paper [3]. The values to compare to are given in Tab. X of Ref. [3] at $x_0 = 1.0$ because our Gaussian is shifted with respect to theirs by 1.5 units.

```
# run plugin toys for combination 1 (simple Gaussian), with physical boundary
# (try to run mulitple jobs to generate more toys)
$ bin/tutorial -c 1 --var a_gaus -a pluginbatch --ntoys 400 --pr --scanrange
    -1.5:2.5 --nrun 1
# compute the plugin intervals from all toys:
$ bin/tutorial -c 1 --var a_gaus -a plugin -j 1 --pr --scanrange -1.5:2.5 -i

a_gaus = [ -1.24,   0.51] ( -0.50 -  0.74 +  1.01) @0.68CL, Plugin
a_gaus = [  -1.5,    1.4] (  -0.5 -   1.0 +   1.9) @0.95CL, Plugin
```

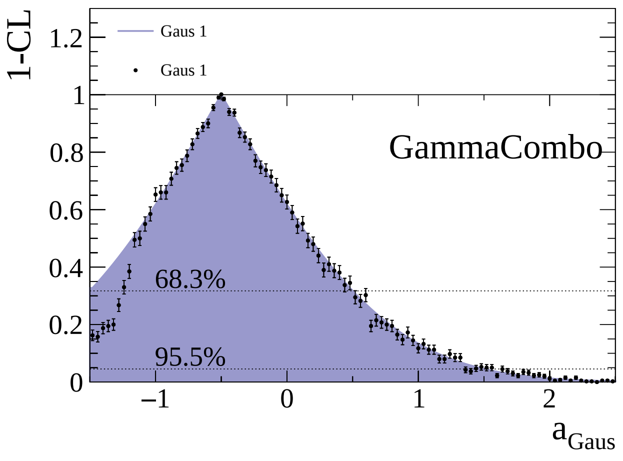

Figure 8: Applying a physical range to combination 1 and running the Plugin method. In this case the Plugin method is equivalent to the method by Feldman-Cousins. The difference to the profile likelihood method is most apparent close to the physical boundary at $a = -1.5$.

4.8 Running the Plugin method in 2D

It is also possible to run the Plugin method in two dimensions. The commands follow the previos lines, however a few additional complications arise. Mostly, the 2D scan becomes computationally demanding quickly, due to the vast number of points needed for smooth 2D contours. Lets take as an example combination 7 (2D Gaus & Circle), which is the pink contour in Fig. 4. We will restrict the 2D grid to a size of 30×30 bins. The result is shown in Fig. 9.

A first attempt just plots the Plugin contours along with the Prob contours, without specifying the contours should be true "2D confidence contours", and should therefore cover the true value in, e.g., 68% of the cases *in two dimensions*. Since one would need additional assumptions on how to translate the p-value, which is computed directly by the PLUGIN method, into an "1D equivalent", this is not done in GAMMACOMBO. Therefore the contours don't match until one give the `--2dcl` option.

Both the PLUGIN and PROB contours seem to match quite well in this example, too. This is not trivial, but perhaps not too surprising given that they already matched in the one-dimensional case of Fig. 3.

To make the best out of scarce toy statistics in the 2D case, a smoothing option was implemented for 2D contours. However, it needs to be used with care: in this example (Fig. 9), it seems to artificially widen the contour over what appears to be reasonable.

The 2D PLUGIN method also produces a p-value histogram, that allows onen to directly judge the status of toy statistics. This is also shown in Fig. 9.

```
# run 2D plugin toys for combination 7 (2D Gaus & Circle)
bin/tutorial -c 7 --var a_gaus --var b_gaus -a pluginbatch --ntoys 100
    --npoints 30 --npointstoy 30 --nrun 1
bin/tutorial -c 7 --var a_gaus --var b_gaus -a pluginbatch --ntoys 100
    --npoints 30 --npointstoy 30 --nrun 2
bin/tutorial -c 7 --var a_gaus --var b_gaus -a pluginbatch --ntoys 100
    --npoints 30 --npointstoy 30 --nrun 3

# compute the plugin contours from all toys. This will produce
# non-matching contours!
bin/tutorial -c 7 --var a_gaus --var b_gaus -a plugin -j 1-3 -i --group off
    --legsize 0.38:0.10 --leg 0.17:0.80

# with the --2dcl option, the contours match
bin/tutorial -c 7 --var a_gaus --var b_gaus -a plugin -j 1-3 -i --group off
    --legsize 0.38:0.10 --leg 0.17:0.80 --2dcl

# lets try to use the smoothing option for the contours
# (in this case it doesn't seem to do any good)
bin/tutorial -c 7 --var a_gaus --var b_gaus -a plugin -j 1-3 -i --group off
    --legsize 0.38:0.10 --leg 0.17:0.80 --2dcl --smooth2d
```

The arguments mean:

`--npoints 30` use 30×30 bins for the 2D PROB scan which is done prior to the PLUGIN toys.

`--npointstoy 30` use 30×30 bins also for the 3D PLUGIN grid. In principle they don't have to use the same granularity. You will receive warnings if, as a consequence of unequal bin sizes, the PLUGIN scan point and the PROB point where the values of the nuisance parameters is taken from, differs too much.

`--group off` turns off the group label—else, in this case, the large GAMMACOMBO string would cover the legend

`--legsize 0.38:0.10` legend size, we use it to reduce the size of the legend

`--leg 0.17:0.80` legend position, we move the legend up a little to not cover

`--2dcl` use two-dimensional confidence contours for the Prob method

`--smooth2d` apply a smoothing algorithm to the 2D contours—to be used with care, see text

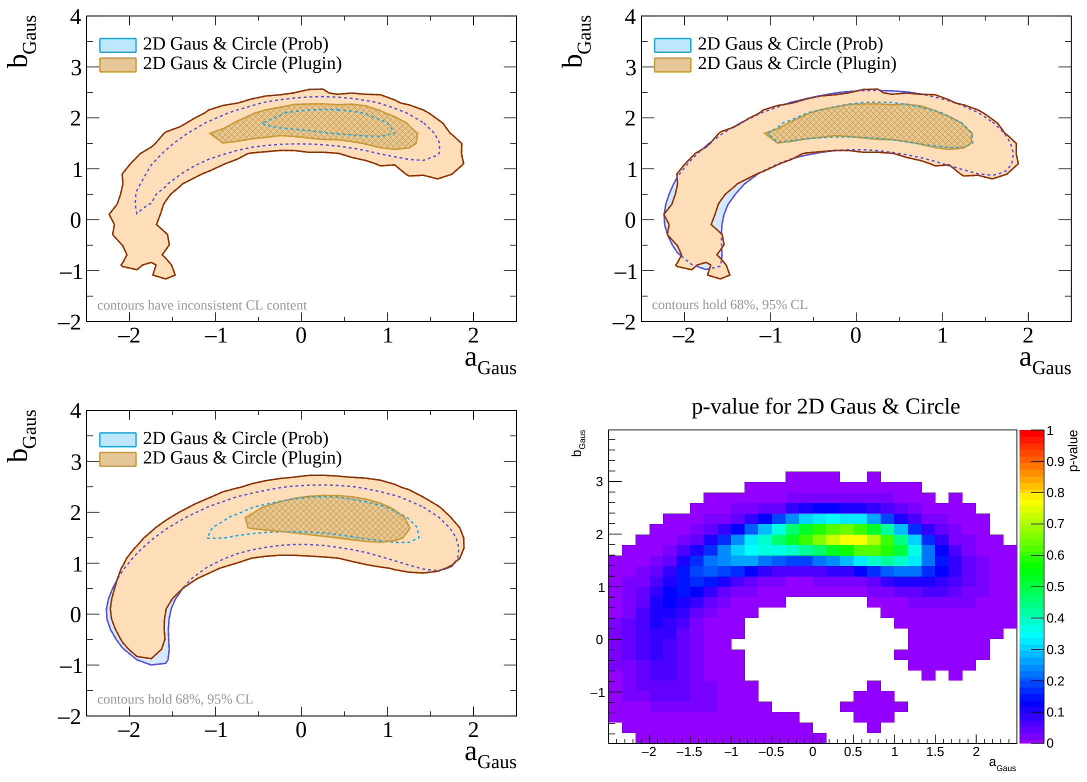

Figure 9: Result of running the Plugin method in two dimensions on combination 7. Top left: confidence contours for the Prob and Plugin methods, when not using the `--2dcl` option, causing inconsistent CL contents in the plot (see text). Top right: Using the `--2dcl` option. Bottom left: Using the 2D smoothing option `--smooth2d`. Bottom right: The raw *p*-value histogram obtained from the Plugin method, with a toy statistics of 300 toys per grid point.

5 Create a new combiner module

In this section we describe how to add a new combiner module to GammaCombo, that will contain everything that is needed for a new combination project: all measurement classes (containing the observed central values, uncertainties, correlations, as well as the PDF), parameter definitions, and a set of predefined combinations.

We will take a real-life example based on a measurement of the CKM angle γ, as reported in Ref. [6]. The observables are cartesian coordinates $x_\pm$ and $y_\pm$, which are linked to the parameter of interest, γ, and two nuisance parameters δ_B^{DK} and r_B^{DK} through the relations

$$x_\pm = r_B^{DK}\cos(\delta_B^{DK} \pm \gamma) \tag{30}$$

$$y_\pm = r_B^{DK}\sin(\delta_B^{DK} \pm \gamma)\,. \tag{31}$$

The observables are measured to be

$$x_- = (\;\;2.5 \pm 2.5 \pm 1.1) \times 10^{-2}\;, \tag{32}$$

$$y_- = (\;\;7.5 \pm 2.9 \pm 1.5) \times 10^{-2}\;, \tag{33}$$

$$x_+ = (-7.7 \pm 2.4 \pm 1.1) \times 10^{-2}\;, \tag{34}$$

$$y_+ = (-2.2 \pm 2.5 \pm 1.1) \times 10^{-2}\;, \tag{35}$$

where the first uncertainty is statistical, and the second is systematic. The statistical and systematic correlations are given in Tables 3 and 4.

Table 3: Statistical cartesian correlations.

	x_-	y_-	x_+	y_+
x_-	1	-0.247	0.038	-0.003
y_-		1	-0.011	0.012
x_+			1	0.002
y_+				1

Table 4: Systematic cartesian correlations.

	x_-	y_-	x_+	y_+
x_-	1	0.005	-0.025	0.070
y_-		1	0.009	-0.141
x_+			1	0.008
y_+				1

In addition to this real-life cartesian example we will add a mock-up measurement of the nuisance parameter r_B^{DK}, which we can then combine with the cartesian example:

$$r_B^{DK} = 0.10 \pm 0.01\,. \tag{36}$$

1. The following steps describe how to create the source files needed for a new combiner. The complete solution is contained in the directory `tutorial/cartesian`, so the quick way to get it run is to just copy this directory one level up to where the other combiner directories live.

   ```
   cp -r tutorial/cartesian .
   ```

   For a more step-by-step like experience, we will copy the existing tutorial directory and modify the source files to make the new cartesian combiner from scratch.

   ```
   cp -r tutorial cartesian
   rm -r cartesian/cartesian # delete the solution!
   ```

2. Tell the GammaCombo build system about the new combiner:

   ```
   vi cmake/combiners.cmake
   # edit the file to look like
   SET( COMBINER_MODULES
      tutorial
      cartesian
   )
   vi cartesian/CMakeLists.txt
   # change the name of the project to look like this
   SET(COMBINER_NAME
      cartesian
   )
   ```

3. Define the project parameters. These will be γ, r_B^{DK}, and δ_B^{DK}. All the parameters are defined in one single class. Here their names are defined, their titles, unit, the default start value for the fit, the default scan range, and the physically allowed region.

   ```
   # rename the parameter class header file to match the new project
   cd cartesian/include
   mv ParametersTutorial.h ParametersCartesian.h
   vi ParametersCartesian.h # edit to change Tutorial for Cartesian
   # also rename the implementation file
   cd cartesian/src
   mv ParametersTutorial.cpp ParametersCartesian.cpp
   vi ParametersCartesian.cpp
   ```

   Add the parameters to the `ParametersCartesian.cpp` file:

   ```
   void ParametersCartesian::defineParameters()
   {
      Parameter *p = 0;

      p = newParameter("g");
   ```

```
   p->title = "#gamma";
   p->startvalue = DegToRad(70);
   p->unit = "Rad";
   p->scan = range(DegToRad(0), DegToRad(180));
   p->phys = range(-7, 7);

   p = newParameter("d_dk");
   p->title = "#delta_{B}^{DK}";
   p->startvalue = DegToRad(127);
   p->unit = "Rad";
   p->scan = range(DegToRad(0), DegToRad(180));
   p->phys = range(-7, 7);

   p = newParameter("r_dk");
   p->title = "r_{B}^{DK}";
   p->startvalue = 0.09;
   p->unit = "";
   p->scan = range(0.02, 0.2);
   p->phys = range(0, 1e4);
}
```

The name is being used on the command line, while the title of a parameter will be used in the plots. The unit of a parameter can be an arbitrary string, however, setting it to `Rad` will tell GammaCombo that this parameter is an angular parameter. During minimization, angular parameters will be kept within the range $[0, 2\pi]$. The start value is being used in the first minimization of the χ^2 function. Some care needs to be taken here, because it is very easy to pick starting values so far away from the sensible region that the χ^2 function reaches too large values that prevent the fit from converging. The default scan range applies to both one- and two-dimensional scans, if not being overwritten from the command line using the options `--scanrange` (horizontal axis) and `--scanrangey` (vertical axis). The physically allowed range can be used to impose hard limits on the value of a parameter, which get activated through the `--pr` option.

4. Create the measurement class holding the measurement of the cartesian observables, as well as the definition of their Gaussian PDF. At first, create the the header file in `include/PDF_Cartesian.h`.

```
#ifndef PDF_Cartesian_h
#define PDF_Cartesian_h

#include "PDF_Abs.h"
#include "ParametersCartesian.h"

using namespace RooFit;
using namespace std;
using namespace Utils;

class PDF_Cartesian : public PDF_Abs
```

```
{
  public:
    PDF_Cartesian(TString cObs, TString cErr, TString cCor);
    ~PDF_Cartesian();
    void         buildPdf();
    void         initObservables();
    virtual void initParameters();
    virtual void initRelations();
    void         setCorrelations(TString c);
    void         setObservables(TString c);
    void         setUncertainties(TString c);
};

#endif
```

5. Constructor implementation.

```
  PDF_Cartesian::PDF_Cartesian(TString cObs, TString cErr, TString cCor)
: PDF_Abs(4) // <-- configure the number of observables
{
  name = "cartesian"; // <-- configure the PDF name, should be unique
  initParameters();
  initRelations();
  initObservables();
  setObservables(cObs);
  setUncertainties(cErr);
  setCorrelations(cCor);
  buildCov();
  buildPdf();
}

PDF_Cartesian::~PDF_Cartesian(){}
```

6. Implement the parameter initialization and the truth relations. The truth relations are implemented through `RooForlumaVar` objects. It is also possible to implement them through customized objects, that inherit from `RooAbsReal`. This way, relations of arbitrary complexity can be added. This will be discussed in Sec. 5.1. The names of the predicted observables need to end on `_th`.

```
void PDF_Cartesian::initParameters()
{
  ParametersCartesian p; // <-- use the project's parameter class
  parameters = new RooArgList("parameters");
  parameters->add(*(p.get("r_dk")));
  parameters->add(*(p.get("d_dk")));
  parameters->add(*(p.get("g")));
}

void PDF_Cartesian::initRelations()
{
   theory = new RooArgList("theory");
   RooArgSet *p = (RooArgSet*)parameters;
   // The order of this list must match that of the COR matrix!
   theory->add(*(new RooFormulaVar("xm_dk_th", "xm_dk_th", "r_dk*cos(d_dk-g)",
       *p)));
   theory->add(*(new RooFormulaVar("ym_dk_th", "ym_dk_th", "r_dk*sin(d_dk-g)",
       *p)));
   theory->add(*(new RooFormulaVar("xp_dk_th", "xp_dk_th", "r_dk*cos(d_dk+g)",
       *p)));
   theory->add(*(new RooFormulaVar("yp_dk_th", "yp_dk_th", "r_dk*sin(d_dk+g)",
       *p)));
}
```

7. Implement the definition of the observables. The names of the observables need to match those of the predicted ones, with the only difference that they end on `_obs`. Their titles will show up in pull plots. The configured value is not important, but the ranges should be far away from any boundary there might be, as they will affect the toy generation within the Plugin method.

```
void PDF_Cartesian::initObservables()
{
   observables = new RooArgList("observables");
   // The order of this list must match that of the COR matrix!
   observables->add(*(new RooRealVar("xm_dk_obs", "x-", 0, -1, 1)));
   observables->add(*(new RooRealVar("ym_dk_obs", "y-", 0, -1, 1)));
   observables->add(*(new RooRealVar("xp_dk_obs", "x+", 0, -1, 1)));
   observables->add(*(new RooRealVar("yp_dk_obs", "y+", 0, -1, 1)));
}
```

8. Implement the measured values of the observables. Based on a config string, different values can be loaded. This is very useful if different versions of a given measurement exist, for example a first measurement from the year 2010, and an update from 2012. These config strings are being used inside the main program file, `main/cartesian.cpp` (see below). The string stored in `obsValSource` is meant for information only, it will be printed in the verbose output (`-v`). The “truth” and “toy” entries should not be touched.

```
void PDF_Cartesian::setObservables(TString c)
{
   if ( c.EqualTo("truth") ){
      setObservablesTruth();
   }
   else if ( c.EqualTo("toy") ){
      setObservablesToy();
   }
   else if ( c.EqualTo("year2014") ){
      obsValSource = "arxiv:1408.2748";
      setObservable("xm_dk_obs", 2.5e-2);
      setObservable("ym_dk_obs", 7.5e-2);
      setObservable("xp_dk_obs",-7.7e-2);
      setObservable("yp_dk_obs",-2.2e-2);
   }
   else {
      cout << "PDF_Cartesian::setObservables() : ERROR : config not found: "
          << c << endl;
      exit(1);
   }
}
```

9. Implement the measured uncertainties of the observables. It is important to match the order that of the observables. The systematic error can be set to zero without problems, but the statistical error should always be finite. The `obsErrSource` string is again solely for informational purposes.

```
void PDF_Cartesian::setUncertainties(TString c)
{
   if ( c.EqualTo("year2014") ){
      obsErrSource = "arxiv:1408.2748";
      StatErr[0] = 0.025; // xm
      StatErr[1] = 0.029; // ym
      StatErr[2] = 0.024; // xp
      StatErr[3] = 0.025; // yp
      SystErr[0] = 0.011; // xm
      SystErr[1] = 0.015; // ym
      SystErr[2] = 0.011; // xp
      SystErr[3] = 0.011; // yp
   }
   else {
      cout << "PDF_Cartesian::setUncertainties() : ERROR : config not found: "
          << c << endl;
      exit(1);
   }
}
```

10. Implement the measured correlations of the observables. Here the order needs to match that of the observables. The function `resetCorrelations()` sets all correlations to zero.

```
void PDF_Cartesian::setCorrelations(TString c)
{
   resetCorrelations();
   if ( c.EqualTo("year2014") ){
      corSource = "arxiv:1408.2748";
      double dataStat[] = {
         // xm     ym     xp     yp
          1.   , -0.247, 0.038, -0.003, // xm
         -0.247, 1.   , -0.011, 0.012, // ym
          0.038, -0.011, 1.   , 0.002, // xp
         -0.003, 0.012, 0.002, 1.    // yp
      };
      corStatMatrix = TMatrixDSym(nObs,dataStat);
      double dataSyst[] = {
         // xm      ym      xp      yp
          1.   ,  0.005, -0.025, 0.070, // xm
          0.005,  1.   ,  0.009, -0.141, // ym
         -0.025,  0.009,  1.   ,  0.008, // xp
          0.070, -0.141,  0.008, 1.    // yp
      };
      corSystMatrix = TMatrixDSym(nObs,dataSyst);
   }
   else {
      cout << "PDF_Cartesian::setCorrelations() : ERROR : config not found: "
          << c << endl;
      exit(1);
   }
}
```

11. Implement the Gaussian PDF. It is possible to use any other functional form for the PDF, as will be described in Sec. 5.2, but the Gaussian is probably the most common choice. Don't change the name of this PDF object.

```
void PDF_Cartesian::buildPdf()
{
   pdf = new RooMultiVarGaussian("pdf_"+name, "pdf_"+name,
       *(RooArgSet*)observables, *(RooArgSet*)theory, covMatrix);
}
```

12. Implement the second PDF class for the measurement of r_B^{DK}. We will just adapt the existing class for a simple-Gaussian measurement from the tutorial, `PDF_Gaus`.

   - copy the header file `include/PDF_Gaus.h` into `include/PDF_rb.h`
   - edit the new header file to adjust the class name (`PDF_rb`), and to include the header file of the parameter class of the cartesian project (`ParametersCartesian.h`).
   - copy the implementation file `src/PDF_Gaus.cpp` into `src/PDF_rb.cpp`
   - edit the class name, change to the parameter class of the cartesian project
   - include the paramter `r_dk` rather than `a_gaus`
   - change the name of the r_B^{DK} observable to e.g. `r_dk_obs`; similar for the theory value
   - set the observed central values for the identifier string "year2013" to 0.1 and the statistical error to 0.01

13. If you created the folder for the cartesian project as suggested by copying the tutorial folder, you should remove now the non-needed classes `PDF_Circle.h`, `PDF_Gaus.h`, `PDF_Gaus2d.h`, to not confuse the build system. Also remove the corresponding implementation files from the `src` directory.

14. Now we need to create the new main file. Rename the one from the tutorial to `main/cartesian.cpp` and change it to look like this:

```
#include <stdlib.h>
#include "GammaComboEngine.h"
#include "PDF_rb.h"
#include "PDF_Cartesian.h"

using namespace std;
using namespace RooFit;
using namespace Utils;

int main(int argc, char* argv[])
{
   GammaComboEngine gc("cartesian", argc, argv);

   // define PDFs
   gc.addPdf(1, new PDF_Cartesian("year2014","year2014","year2014"),
       "Cartesian");
   gc.addPdf(2, new PDF_rb("year2013","year2013","year2013"), "rb");

   // Define combinations
   gc.newCombiner(0, "empty", "empty");
   gc.newCombiner(1, "cartesian1", "Cartesian",  1);
   gc.newCombiner(2, "cartesian2", "rb",         2);
   gc.newCombiner(3, "cartesian3", "Cartesian & rb", 1,2);

   // Run
   gc.run();
}
```

15. Change to the build directory and build the new project

```
cd build
make install -j4
```

16. Run the new combiner. The results are shown in Fig. 10.

```
cd cartesian
bin/cartesian -u
bin/cartesian -c 1 -c 3 --var g -i
bin/cartesian -c 1 -c 2 -c 3 --var r_dk -i
```

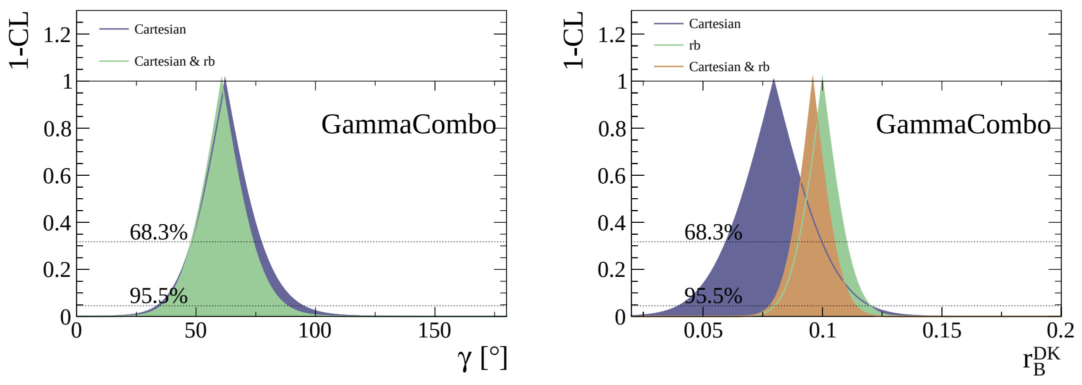

Figure 10: Results of the new combiner module "cartesian". Left: curves for parameter γ. Right: curves for (nuisance) parameter r_B^{DK}.

5.1 Advanced: `C++` truth relations

To be filled.

5.2 Advanced: Non-Gaussian observables

To be filled.

6 Limits with Datasets

Originally, Gammacombo was designed to combine a handful of data points into a measurement. But its functionality has since been extended to set limits on physical parameters by modelling big sets of data with probability density functions. For example, GammaCombo can be used to obtain a confidence interval on the branching ratio of a particle decay by analysing the distribution of detector events in the mass variable.

Despite the fact that this is theoretically the same problem as the combination of measurements, it comes with its own challenges. Therefore, GammaCombo offers a separate interface for these kind of problems. It is described in this chapter.

6.1 Terminology

A useful piece of terminology associated with limit setting on datasets is the distinction of *global observables.* As opposed to the "normal" observabeles such as the mass or the lifetime of particles, which are measured many times, which are identically and independently distributed and which are stored in datasets, the *global observables* are those that are measured only once. Typical exampels are external inputs such as the world average of the branching ratio of a control channel. In a way, one can think of a a normal GammaCombo combination as a dataset combination that contains only global observables.

The function that models the probability distribution for a global observables is called its *constraint PDF.* GammaCombo assumes that the complete probability density function describing your experiment factorizes into the PDF(s) describing the independent and identically distributed events in the dataset(s) and a coupole of constraint PDFs.

6.2 Usage

Other than for the simple combinations, the command line is not very useful when calculating limits on datasets. Instead, the combination is almost fully specified in the main cpp file of the combiner subpackage. Also, it was decided that instead of defining the PDFs as Gammacombo classes, they should instead be loaded from a https://root.cern.ch/roofit /emphRooWorkspace. The dataset(s), the global observables, the probability density function - all these must be present in the workspace. In fact, Gammacombo even expects that the fit model has been fitted to the data once. (This tight dependence on a RooWorkspace has been a controversial decision among the developers and might be removed in a future iteration.)

The components that are required to be present in the workspace are:

- A `RooAbsPdf` that models the dataset - without the constraints on global observables

- A RooDataSet that contains the data
- The RooFitResult of the initial fit to data. This MUST be named data_fit_result in the workspace.
- A RooArgSet containing PDFs for the constraints.
- A RooArgSet that lists the global observables.
- A RooArgSet that lists the non-global observables whose measurements are stored in the dataset (for example the mass).
- A RooArgSet that lists all parameters.

Critically, GammaCombo also assumes that all RooRealVars have the values that correspond to the fit result of the free fit to data.

Each object in the RooFit workspace has a name, be it the RooAbsPdf or a RooArgSet. These names also identify the objects in the workspace and need to be passed to GammaCombo. This way they can be found by the framework. The function calls which pass these names to the gammacombo engine can be derived from the example in the following section.

6.3 Example

The example for setting limits with datasets is part of the tutorial combiner and is included in the default installation. As mentioned before, we require a workspace which holds the data, the different observables and parameters, grouped into sets, as well as the probability density function and the result of an initial fit to the dataset. This workspace is construed by the function tutorial_dataset_build_workspace.cpp which can be found in the tutorial/main folder. This function also generates the dataset that is taking the place of the dataset to be analyzed. Obviously, this is only for the sake of example. Have a look into the file to see what it does. Then execute it by calling

```
bin/tutorial_dataset_build_workspace
```

A workspace should be created and saved into the bin directory.

The main file for the dataset tutorial is called tutorial_dataset.cpp and is located in the tutorial/main folder as well. It is faily simple and will look very similar for other analyses:

```
#include "GammaComboEngine.h"
#include "TFile.h"
#include "RooGaussian.h"
```

```
#include "RooExponential.h"
#include "RooWorkspace.h"

int main(int argc, char* argv[])
{

  // Load the workspace from its file
  TFile f("workspace.root");
  RooWorkspace* workspace = (RooWorkspace*)f.Get("dataset_workspace");
  if (workspace==NULL){
    std::cout<<"No workspace found:"<<std::endl;
  }

  // Construct the PDF and pass the workspace to it
  PDF_Datasets* pdf = new PDF_Datasets(workspace);
  pdf->initData("data"); // this is the name of the dataset in the workspace
  pdf->initPDF("mass_model"); // this the name of the pdf in the workspace
      (without the constraints)
  pdf->initObservables("datasetObservables"); // non-global observables whose
      measurements are stored in the dataset (for example the mass).
  pdf->initGlobalObservables("global_observables_set"); // global observables
  pdf->initParameters("parameters"); // all parameters
  pdf->initConstraints("constraint_set"); // RooArgSet containing the
      "constraint" PDF's

  // Start the Gammacombo Engine
  GammaComboEngine gc("tutorial_dataset", argc, argv);

  // add the PDF
  gc.addPdf(0, pdf);
  // there is just a single pdf, and its id is 0

  // Combiners are not supported when working with datsets.
  // The statistical model is fully defined with the PDF
  gc.run(true); // The boolean parameter is a flag that tells Gammacombo that
      we are analysing a dataset
}
```

The most notable difference to the "normal" gammacombo is that no `combiner` objects are used. This means that there is (currently) no way to add or remove measurements when datasets are involved. Also make sure to pass `true` to the `run()` method when calling. This tells GammaCombo that we are analyzing a dataset and that it shouldn't expect a combiner. Another important difference: You do not need to define a PDF in GammaCombo. The `PDF_Datasets_Abs` object is simply a wrapper around the `RooFit` probability density function that is contained in the workspace. It provides an interface to GammaCombo and handles the generation of simulated events.

To run the example, call

```
  bin/tutorial_dataset --var branchingRatio --npoints 10 --scanrange
      1e-7:2e-6 -i
```

This performs a simple prob scan on the example data. The p-value is determined at 10 points in the given scan range. The name of the scanned parameter is `branchingRatio`. This name is originally defined when building the workspace. A plot of the p-value curve will be saved to `plots/bin`.

After completing the prob scan, you can also run a plugin scan:

```
bin/tutorial_dataset -a pluginbatch --var branchingRatio --ntoys 50
    --npoints 10 --scanrange 1e-7:2e-6
```

This will produce the necessary toy distributions of the test statistic, but I won't calculate any p-values yet. Make sure that scan range and the number of points does exactly match the argument values used in the prob scan. Finally, to calculate and plot the p-values from the plugin scan:

```
bin/tutorial_dataset -a plugin --var branchingRatio -i
```

Again, a plot of the p-value curve will be saved to `plots/bin`.

6.4 Custom implementations of `PDF_Datasets_Abs`

You may find that the default fitting and event generation algorithms in the `PDF_Datasets_Abs` class are not sufficient for your purpose.

In these cases, you can substitute your own implementation of these methods. This is most easily obtained by writing a new class that inherits from `PDF_Datasets_Abs` and implements the corresponding virtual methods. Here is how the header file of such a class could look like:

```
#ifndef PDF_DatasetTutorial_h
#define PDF_DatasetTutorial_h

#include "PDF_Datasets_Abs.h"

class PDF_DatasetTutorial : public PDF_Datasets_Abs
{
public:
  PDF_DatasetTutorial(RooWorkspace* w);
  ~PDF_DatasetTutorial();
  virtual RooFitResult* fit(bool fitToys = kTRUE) override;
  virtual void        generateToys(int SeedShift = 0) override;
  virtual void        generateToysGlobalObservables(int SeedShift = 0)
      override;
};
```

```
#endif
```

This would allow to re-implement the methods `fit` and `generateToys` and `generateToysGlobalObservables`. Do not forget to call the constructor of the parent class:

```
#include "PDF_DatasetTutorial.h"

PDF_DatasetTutorial::PDF_DatasetTutorial(RooWorkspace* w):
    PDF_Datasets_Abs(w){}
PDF_DatasetTutorial::~PDF_DatasetTutorial();

RooFitResult* PDF_DatasetTutorial::fit(bool fitToys){
   ... your implementation here ...
};

void  PDF_DatasetTutorial::generateToys(int SeedShift) {
   ... your implementation here ...
}
void  PDF_DatasetTutorial::generateToysGlobalObservables(int SeedShift) {
   ... your implementation here ...
}
```

As an example, the class `PDF_DatasetTutorial` can be found in the `tutorial/includes` and `tutorial/src` folders. It re-implements the method for generating the toy dataset and the call to the `RooFit` fitter. In order to use it for the tutorial, you can must include the `"PDF_DatasetTutorial.h"` header at the top of the `tutorial_datasets.cpp` file and replace the line where the PDF object is instantiated from

```
  PDF_Datasets* pdf = new PDF_Datasets(workspace);
```

to

```
  PDF_Datasets* pdf = new PDF_DatasetTutorial(workspace);
```

After compilation, you are now using the custom implementation.

7 Advanced Topics

Most of this section is yet to be filled.

7.1 Numerical results

The printout of GammaCombo contains the parameter values of each local minimum of the χ^2 function that was found during the scan. These solutions are ordered by their χ^2 value, so the global minimum is displayed first. An example from tutorial combination 7 (see Sec. 4.4) is:

```
SOLUTION 0:

  combination: tutorial7
  title:      2D Gaus & Circle
  date:       Sun Feb 22 23:01:27 2015
  FCN: 0.20063, EDM: 1.58985e-07
  COV quality: 3, status: 0, confirmed: yes

    Parameter                    FinalValue +/- Error (HESSE)
  ------------------------------ ------------------------------------
     0              a_gaus      0.411559 +/-  0.758865
     1              b_gaus       1.93224 +/-  0.265757
```

The several components have the following meaning:

SOLUTION 0 is the running index of the displayed solution

combination the name given to the combination in the main file

title the title given to the combination in the main file

FCN the value of the χ^2 function

EDM estimated distance to minimum, as provided by MINUIT

COV quality result of `RooFitResult::covQual()`

status result of `RooFitResult::status()`

confirmed solutions are first found as maxima of the p-value curve, which is obtained while the scan parameter is fixed to the scanned point. Before printout, the χ^2 function gets reminimized with the scan parameter floating, too. Only if the reminimization agrees reasonably with the result from the scan, the solution is regarded as confirmed.

Error (HESSE) the parabolic error obtained by HESSE

All solutions get stored to disk into parameter files into the `plots/par` directory. They have the following format:

```
cat plots/par/tutorial_tutorial7_a_gaus.dat

##### auto-generated by ParameterCache #######
##### printed on Sun Feb 22 23:01:27 2015 ######
# ParameterName                value      errLow     errHigh

----- SOLUTION 0 -----
### FCN: 0.20063, EDM: 1.58985e-07
### COV quality: 3, status: 0, confirmed: yes
a_gaus                        0.411559   -0.758865    0.758865
b_gaus                        1.932237   -0.265757    0.265757
```

Here, `errLow` and `errHigh` are obtained from the parabolic HESSE errors, so will always be the same.

It is possible to print out the correlation matrices of each solution. This is triggered by the command line option `--printcor`.

7.2 Defining a custom scan strategy and starting points

In more complex combinations, it can happen that the χ^2 function has multiple local minima, that are well separated from each other. As the scan progresses, different values of nuisance parameters can shrink the χ^2 hill separating the minima, and it may happen, that the fit scanner jumps into the second mininum and stays there. This will usually be visible as a jump in the p-value curve. One can be, however, unlucky and completely miss a second mininum.

A good way to alleviate this problem is to redo the scan using another starting point, and to accept points with a lower χ^2 value (a higher p-value) into the final curve, if any are found. In GammaCombo, the default scans are done in "drag mode", where the fit at each scan step uses as start parameters the fit result from the previous scan step. Therefore it matters what the start parameters of the first scan were. The default is to use the parameter values defined in the parameter class (`ParametersTutorial.cpp` in the tutorial).

It is possible to override this default to explicitly start at a certain point. For this, the parameter files (see Sec. 7.1) are used: In order to make a start parameter file, one simply copies one parameter file, adding `_start` to the name. For example:

```
cp plots/par/tutorial_tutorial7_a_gaus.dat
    plots/par/tutorial_tutorial7_a_gaus_start.dat
```

Start parameter files can contain any number of parameter points (labelled "SOLUTION" inside the file). For each point found, one rescan will be performed.

Parameter files can also be used from other combinations. If it contains values for parameters that are not in the current combination, they will simply be ignored.

Instead of creating a start parameter file, one can also just provide a `.dat` file through the command line option `--parfile`.

A good strategy to obtain flawless p-value curves is to rescan once per local minimum of the χ^2 function. This requires that one knows already more or less where the minima are—more about strategies how to find them in Sec. 8. For this reason the default scan strategy in GammaCombo is to first run an initial scan, and then to rescan for each local minimum encountered.

7.3 Plotting an overview of a combination's structure

GammaCombo produces a graphical representation of the content of a combination in the `.dot` format, that can be interpreted by the `dot` command of the `graphviz` package.

7.4 Control Plots

- Giving the option `--pulls` will produce a pull plot showing which observables deviate the most from the prediction. It will also show the value of the χ^2 function at the best fit value, and the corresponding (naive) fit probability.
- Giving the option `-e` will produce a plot of the parameter evolution of a Prob scan.
- Many control plots exist to judge the quality of the Plugin toys.

7.5 Run an Asimov toy

7.6 Predict observables

- Observable names are unique in the Combiner so multiple PDFs with observables of the same name can be combined. For this purpose each observable has a unique ID appended, for example "_UID2" for ID 2. The number corresponds to the number that the PDF which provides this observable has in the Combiner.
- One can make predictions for observables by simply scanning for them. When doing this, the corresponding χ^2 constraint usually needs to be tightened. This effectively replaces the inversion of the truth equation by the minimization. A plot is being produced showing the deviation of the actual value of the observable and the predicted one, as a function of the scan step. This should be a flat line at zero, and allows to judge if the constrained was tightened enough.

7.7 Attempt a coverage correction

8 Use cases, hints, and advice

to be filled

9 Command line options

Here is a list of available command line options. The full list can always be obtained by running

```
bin/tutorial -h
```

`--2dcl` plot contours corresponding to "two-dimensional" confidence level. The default is to plot contours such that their projection to the axes coincides with the one-dimensional scans. In this case the resulting innermost contour corresponds to 39% CL. When the option is given, the innermost contour is enlarged to hold 68% CL. Sect. 4.3.

`--asimov` run an Asimov toy. Sect. 7.5.

`--color` changes the plot color of the plotted combination. There are six different colors for one-dimensional plots defined in `GammaComboEngine::defineColors()`, that can be specified through numbers 0–5. For two-dimensional plots, there are four different colors defined in `OneMinusClPlot2d::OneMinusClPlot2d()`, that can be specified through the numbers 0–3.

`--controlplots` Make control plot for the Plugin toys. Sect. 9.

`--covCorrectPoint` Define the point for the coverage correction (experimental). Sect. 7.7.

`--covCorrect` Activate a coverage correction (experimental). Sect. 7.7.

`--digits, -s` Set the number of printed digits right of the decimal point. Default is automatic.

`--evol, -e` Plot the profile likelihood parameter evolution of a Prob scan. Sect. 7.4.

`--fix 'g=1.7,r_dk=-0.09'` Fix given scan parameters to the given values.

`--group LHCb` Change the GammaCombo logo to LHCb.

`--id 57` When making controlplots (`--controlplots`), only plot toys generated at a specific scan point, e.g. 57. The id number refers to the scan step and therefor to a specific value of the parameter of interest.

`--importance` Activate importance sampling for Plugin toys. This reduces the number of toys generated in a region with large expected p value, therefore saving computing cycles, that can better be invested to get a better p value accuracy in the tails.

`--intprob` Use the internal (Prob) χ^2 histogram instead of the χ^2 from the toy files to evaluate $1 - \mathrm{CL}$ of the Plugin method. Sect. 7.4.

`--largest` Report largest CL interval: from the lowest boundary of all intervals to the highest boundary of all intervals. Useful if two intervals are very close together.

`--leg` Give print options for the legend such as its position, or to turn it off.

`--lightfiles` Produce only light weight Plugin toy files. They cannot be used for control plots but save disk space.

`--log` make logarithmic one-dimensional 1-CL plots

`--magnetic` switches on magnetic plot boundaries, that will drag the contours towards them. In many cases, this results in nicer contours, but not always. Sect. 4.3.

`--ncontours` Plot this many sigma contours in 2D plots (max 5).

`--ndivy` Set the number of axis divisions (y axis in 1D and 2D plots).

`--ndiv` Set the number of axis divisions (x axis in 1D and 2D plots).

`--nosyst` Ignore systematic uncertainties.

`--npoints2dx` Configure the number of scan points on x-axis in 2D scans.

`--npoints2dy` Configure the number of scan points on y-axis in 2D scans.

`--npointstoy` Number of scan points used by the plugin method.

`--npoints` Configure the number of scan points used in 1D and 2D scans (for 2D, use same number of points for each axis).

`--nrun 1` run job number 1. This number will be added to the filenames of the produced Plugin toy files. Files will be overwritten if the same number is given.

`--ntoys 100` produce 100 Plugin toys per scan point.

`--parfile` Give a specific parameter file to provide, for example, start parameters. Sect. 7.2.

`--plotid` Make the control plot with given ID to save time (`--controlplots`). Sect. 7.4.

`--pluginplotrange` Restrict the Plugin plot to a given range to rejcet low-statistics outliers.

`--po` make a Plugin-only plot.

`--prelim` Plot “Preliminiary” into the plots. See also `--unoff`.

`--printcor` Print correlation matrix of each solution found.

`--probforce` Use a stronger minimum finding algorithm for the Prob method. Sect. 7.

`--pr` physical range. This will enforce the physical parameter ranges defined in `src/ParametersTutorial.cpp`. Sect. 4.7.

`--ps 1` print a marker at all local minima. If `--ps 2` is given, only the best fit value is indicated.

`--ps 1` print solution on the plot. The given value configures the position of the numerical value on the plot. See the help `bin/tutorial -h` for options.

`--pulls` Make a pull plot illustrating the consistency of the best solution with the observables. Sect. 7.4.

`--qh` A list of quick hacks that can be enabled through the command line, that modify very specific areas of the code. These areas can be found by searching for calls to the `isQuickhack()` function.

`--scanforce, -f` Use a stronger minimum finding method for the PLUGIN method.

`--scanrange -1.5:2.5` Adjust the scan range to the given range. The default range is defined in `src/ParametersTutorial.cpp`. Sect. 4.7.

`--scanrangey -1.5:2.5` For 2D plots: Adjust the scan range of the vertical axis to the given range. The default range is defined in `src/ParametersTutorial.cpp`. Sect. 4.7.

`--sn2d` 2D version of `--sn`.

`--sn` Save nuisances to a parameter cache file at certain scan point. Sect. 7.2.

`--title` Override the title of a combination.

`--unoff` Plot "Unofficial" into the plots. See also `--prelim`.

`--var a_gaus --var b_gauss` specify the two variables to perform the scan for. The variable given first will be plotted on the horizontal axis, the second one on the vertical axis.

`--var` the variable name of the theory parameter to be scanned.

`-a plot` activates the plot action: Whenever a scan is done, the result is saved into a file in `plots/scanner`. This can be read in to remake the plot, without having to rerun the entire scan. With more complex combinations, this saves time.

`-a pluginbatch` activates the PLUGIN batch mode. This will run a number of PLUGIN toys and save them into an output file. Many of these jobs can be run in parallel for example on a batch farm.

`-a plugin` activates the PLUGIN mode. This will read in a number of PLUGIN toy files that where previously produced. These files need to be specified using the `-j` option.

`-c 1:+2` modifying combination number 1 by adding PDF number 2. Sect. 4.5.

`-c 5` combination number 5. This refers to pre-defined combinations. A list of available IDs can be obtained by running with the usage flag, `bin/tutorial -u`. When the combination argument is given multiple times, all combinations will be computed and added to the same plot. The order controls the order in the plot, where the last one is in the foreground. Sect. 4.1.

`-d` Activate debug level output.

`-i` interactive mode. Any plot will be shown directly in the familiar ROOT canvas. Exit with `Ctrl+c`.

`-j 1-3` read in Plugin toy files number 1 through 3.

`-u` Prints usage information and exits.

`-v` Activate verbose output.

10 Acknowledgements

The author would like to thank Max Schlupp, Marco Gersabeck, Omer Tzuk, Matt Kenzie, and Florian Bernlochner for valuable discussions and contributions to GammaCombo.

References

[1] LHCb collaboration, R. Aaij *et al.*, *Measurement of the CKM angle γ from a combination of $B^\pm \to Dh^\pm$ analyses*, Phys. Lett. **B726** (2013) 151, `arXiv:1305.2050`.

[2] Particle Data Group, J. Beringer *et al.*, *Review of Particle Physics (RPP)*, Phys. Rev. **D86** (2012) 010001.

[3] G. J. Feldman and R. D. Cousins, *A Unified approach to the classical statistical analysis of small signals*, Phys. Rev. **D57** (1998) 3873, `arXiv:physics/9711021`.

[4] B. Sen, M. Walker, and M. Woodroofe, *On the unified method with nuisance parameters*, Statistica Sinica **19** (2009) 301.

[5] T. M. Karbach, *Feldman-Cousins Confidence Levels - Toy MC Method*, `arXiv:1109.0714`.

[6] LHCb collaboration, R. Aaij *et al.*, *Measurement of the CKM angle γ using $B^\pm \to DK^\pm$ with $D \to K^0_S\pi^+\pi^-, K^0_S K^+K^-$ decays*, `arXiv:1408.2748`.