BLUE
A software package to combine correlated estimates within ROOT
Program manual
Version 1.9.0

Richard Nisius

January 28, 2014

Max-Planck-Institut für Physik (Werner-Heisenberg-Institut)
Föhringer Ring 6, D-80805 München, Germany,
http://www.mpp.mpg.de/~nisius,
Richard.Nisius@mpp.mpg.de

Abstract

The combination of correlated estimates of a number of observables is a common task in particle physics. This is frequently performed using the BLUE (Best Linear Unbiased Estimate) method.

Given the widespread usage of the ROOT analysis package, a flexible ROOT implementation of the BLUE mathematical framework has been written, and is described in this manual. The software is freely available from the corresponding hepforge project page. Given it is based on ROOT, it is distributed under the GNU Lesser Public License.

1 Introduction

The combination of a number of estimates for a single observable is discussed in Ref. [1]. Here, the term estimate denotes a particular outcome (measurement) of an experiment based on an experimental estimator of the observable, which follows a probability density distribution (pdf). The particular estimate obtained by the experiment may be a likely or unlikely outcome given that distribution. Repeating the measurement numerous times the estimates will follow the underlying pdf of the estimator. The analysis makes use of a χ^2 minimisation to obtain the combined values. In Ref. [1], this minimisation is expressed in the mathematically equivalent BLUE language.

Provided the estimators are unbiased, when applying this formalism the **Best Linear Unbiased Estimate** of the observable is obtained with the following meaning. **Best:** The combined result for the observable obtained this way has the smallest variance; **Linear:** The result is a linear combination of the individual estimates; **Unbiased Estimate:** When the procedure is repeated for a large number of cases consistent with the underlying multidimensional pdf, the mean of all combined results equals the true value of the observable. The extension to more than one observable is described in [2].

For many years, a freely available Fortran based software [3] to perform the combination for a number of estimates and for several observables was widely used. The implementation of the BLUE method described here is targeted at being used within the ROOT analysis framework [4].

The equations to solve the problem for the general case of m estimates of n observables with $m \geq n$ can be found in [2]. They are implemented in the software presented, but are not repeated here. However, the simple case of two correlated estimates of the same observable is discussed in some detail. This is because already for this case the main features of the combination can easily be understood.

Let x_1 and x_2 with variances σ_1^2 and σ_2^2 be two estimates from two unbiased estimators of the true value x_T of the observable, and ρ the total correlation of the two estimators. Let the estimator resulting in x_1 be as least as precise as the estimator of x_T than the estimator yielding x_2, such that $z \equiv \sigma_2/\sigma_1 \geq 1$. Then the BLUE of x_T is:

$$x = (1-\beta)\, x_1 + \beta\, x_2\,,$$

where β is the weight of the less precise estimate. The variable x is the combined result and σ_x^2 is its variance. To investigate the improvement on the precision of x when adding the information of x_2 to the more precise estimate from x_1, i.e. to decide whether it is worth combining, the variable σ_x/σ_1 is investigated. This variable quantifies the uncertainty of the combined value in units of the uncertainty of the more precise estimate, i.e. $1-\sigma_x/\sigma_1$ is the relative improvement achieved by also using x_2 from the less precise estimator.

The two quantities and their derivatives with respect to the parameters ρ and z are given in Eqs. 1–6. They are valid for $-1 \leq \rho \leq 1$ and $z \geq 1$, but for $\rho = z = 1$. The resulting variations of the combined value are given in Eqs. 7–8.

$$\beta = \frac{x - x_1}{x_2 - x_1} = \frac{1-\rho z}{1 - 2\rho z + z^2} = \frac{1-\rho z}{(1-\rho z)^2 + z^2(1-\rho^2)} \tag{1}$$

$$\frac{\sigma_x}{\sigma_1} = \sqrt{\frac{z^2(1-\rho^2)}{1-2\rho z + z^2}} \tag{2}$$

$$\frac{d\,\beta}{d\,\rho} = \frac{z(1-z^2)}{(1-2\rho z + z^2)^2} \tag{3}$$

$$\frac{d\,\frac{\sigma_x}{\sigma_1}}{d\,\rho} = z(z-\rho)(1-\rho z)\sqrt{\frac{1}{(1-\rho^2)(1-2\rho z+z^2)^3}} \tag{4}$$

$$\frac{d\,\beta}{d\,z} = \frac{\rho(1+z^2) - 2z}{(1-2\rho z + z^2)^2} \tag{5}$$

$$\frac{d\,\frac{\sigma_x}{\sigma_1}}{d\,z} = (1-\rho z)\sqrt{\frac{1-\rho^2}{(1-2\rho z+z^2)^3}} \tag{6}$$

$$\frac{d\,x}{d\,\rho} = (x_2 - x_1)\,\frac{d\,\beta}{d\,\rho} \quad (7)$$

$$\frac{d\,x}{d\,z} = (x_2 - x_1)\,\frac{d\,\beta}{d\,z} \quad (8)$$

The resulting β and σ_x/σ_1, as functions of ρ, and for various z values (Eq. 1 and Eq. 2) are shown in Figures 1(a) and 1(b). A few features of the variables β and σ_x/σ_1 are discussed below that are important to understand the results of the combination.

The value of β is smaller or equal to 0.5, because otherwise x_2 would be the more precise estimate. Since the denominator in Eq. 1 is positive for all allowed values of ρ and z, the function for β turns negative for $\rho > 1/z$ as shown in Figure 1(a). As can be seen from the second term in Eq. 1, the value of β can be interpreted as the difference of the combined value from the more precise estimate in units of the difference of the two estimates. When β is negative, the signs of the numerator and denominator are different. This means the value of x lies on the opposite side of x_1 than x_2 does, or in other words, the combined value lies outside the range spanned by the two estimates.

Since the denominator in Eq. 1 and Eq. 2 are identical, and the denominator of Eq. 1 equals the numerator of Eq. 2 plus an additional term that is positive for all values of ρ and z, the value of σ_x/σ_1 is always smaller than 1 as shown in Figure 1(b). Again this is expected, since including the information from the estimate x_2 should improve on the knowledge of x, which means on its precision σ_x. Not surprisingly, the value of σ_x/σ_1 is exactly one for $\rho = 1/z$, i.e. when $\beta = 0$. In this situation, the information from x_2 is ignored in the linear combination, and consequently $x = x_1$ and $\sigma_x = \sigma_1$.

The derivatives of β and σ_x/σ_1 with respect to ρ as functions of ρ, and for various z values (Eq. 3 and Eq. 4) are shown in Figures 1(c) and 1(d). The equations for β and σ_x/σ_1, this time as a function of z and for various ρ values, are shown in Figures 2(a) and 2(b). Finally, the derivatives of β and σ_x/σ_1 with respect to z as functions of z, and for various ρ values (Eq. 5 and Eq. 6) are shown in Figures 2(c) and 2(d). These derivatives can be used to evaluate the sensitivity of the combined result to the imperfect knowledge on both the correlation ρ and the uncertainty ratio z of the individual estimators. With this information the stability of the combined result can be assessed and a decision can be taken on whether to refrain from combining. This decision should only be based on the features of the parameters σ_x/σ_1 and β but not on the outcome for a particular pair of estimates x_1 and x_2. This is because these parameters and their derivatives are features of the underlying probability distributions of the estimators, whereas the two specific values are just a pair of estimates, i.e. a single possible outcome of results.

This manual is organised as follows: The software structure is outlined in Section 2, followed by the description of the user interface given in Section 3. A number of examples provided are discussed in Section 4. The conversion of input files for the Fortran software [3] to functions to be used with this ROOT implementation is explained in Section 5. Some hints on the installation and usage of the software are given in Section 6. Conclusions are drawn in Section 7 followed by an appendix. In Appendix A, the relations for β and σ_x, Eq. 1 and Eq. 2, are derived in the BLUE formalism. Finally, the changes made to the software are documented in the release notes given in Appendix B.

2 Software structure

This section explains the general strategy for the usage of the package. The details of the functions mentioned here are given in Section 3. The functionality is implemented in a ROOT class called `Blue` that derives from `TObject`. No attempt has been made to override the default implementations provided by this, but for what is described below.

The usage of the software is separated in up to three steps.

1. During the first step the constructor is called and the individual estimates and their uncertainties, as well as all correlation matrices of the uncertainty sources are filled. Optionally, also names for estimates, uncertainty sources and observables can be filled. When this has been completed, the input stream is closed automatically and the filling functions are disabled.
2. In the second (optional) step individual estimates and/or uncertainty sources can be disabled, or correlation assumptions can be altered for the combination to follow by calling the corresponding `Set...()` functions. If this step is used, before a further combination can be performed, the input to the combination has to be fixed by the user by calling `FixInp()` indicating the end of the selection. After this call a number of `Print` functions are available for digesting the input and the selections made.
3. In the third step the actual combination is performed by calling (`FixInp()` if step 2 is omitted and) one of the `Solve...()` functions. A number of `Print` functions are provided for digesting the result for the observables.

The second and third steps can be performed as often as wanted. In this case, after any combination, first the input has to be freed for further selections by calling either `ReleaseInp()` or `ResetInp()`. The difference of these two options is discussed below.

3 Details of the interface

This section describes the details of the interface. All arguments passed to member functions are declared as `const`, but for those that are return values as described below. However, this fact is not mentioned in the description of the function prototypes below. This means arguments denoted as `Int_t` in fact are `const Int_t`. In contrast, functions that are `const`, i.e. those that do not alter the state of the object are marked as such.

3.1 Constructor

`Blue(Int_t NumEst, Int_t NumUnc, Int_t NumObs, Int_t* IWhichObs, Int_t* IWhichFac)`:
`Blue(Int_t NumEst, Int_t NumUnc, Int_t NumObs, Int_t* IWhichObs)`:
`Blue(Int_t NumEst, Int_t NumUnc, Int_t* IWhichFac)`:
`Blue(Int_t NumEst, Int_t NumUnc)`: The first constructor instantiates the object for a number of estimates (`NumEst`), uncertainty sources (`NumUnc`) and observables (`NumObs`). The array `IWhichObs` indicates which observable a given estimate is determining. The array `IWhichFac` defines different groups to be considered in systematic variations of the correlation assumptions, when using `SolveScaRho()`, see below. The input for the example of four estimates, ten uncertainty sources for two observables, where the first two estimates determine the first observable,

and the second two estimates determine the second observable is: `NumEst = 4`, `NumUnc = 10`, `NumObs = 2`, and `IWhichObs = {0,0,1,1}`. If these fall into two groups of estimates, e.g. (0, 2) and (1, 3), which e.g. could stem from two experiments, and for which the correlation assumption should be scanned differently for the pairs of estimates from the same experiment (02) and (13), or from different experiments (01), (03), (21) and (23), the following info should be provided:

$$\texttt{WhichFac} \;=\; \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \tag{9}$$

where the array `IWhichFac` should contain this matrix in row wise storage. The values on the diagonal are not relevant, the off-diagonal elements should start from zero and run up to `l = NumFac-1`, where `NumFac` is the number of groups desired.

In the case of a single observable, i.e. if `NumObs = 1`, the information in `IWhichObs` is redundant and ignored. In this case the more simple constructors can be used instead. If also possible scans in `SolveScaRho()` should be performed simultaneously for all pairs of estimates, the last constructor is sufficient.

3.2 Filling functions

`void FillEst(Int_t i, Double_t* x)`: The estimate i with the index in the following range: `i = 0, ..., NumEst-1` is filled. The array x must contain `NumUnc + 1` entries, namely the value of the estimate and the individual uncertainties in the following form: $x = \texttt{Value}, \sigma_0, \sigma_1, \ldots, \sigma_{k_{\max}}$ with $k_{\max}$ `= NumUnc - 1`. The software assumes that σ_0 is the statistical uncertainty and σ_k with $k > 0$ are systematic uncertainties.

If for a source k a negative entry $\sigma_k < 0$ is supplied, this value is considered a percentage uncertainty. During filling this is converted from $\sigma_k \to -\sigma_k \cdot$ `Value / 100`.

`void FillCor(Int_t k, Double_t* x)`: The correlation matrix of the uncertainty k with indices in the range $k = 0, \ldots,$ `NumUnc-1` is filled. For the example of `NumEst = 3` the correlation matrix for any uncertainty source k is:

$$V \;=\; \begin{pmatrix} V_{00} & V_{01} & V_{02} \\ V_{10} & V_{11} & V_{12} \\ V_{20} & V_{21} & V_{22} \end{pmatrix}. \tag{10}$$

The array x must contain the row wise storage of this matrix, i.e. for the above example it should read $x = V_{00}, V_{01}, V_{02}, V_{10}, V_{11}, V_{12}, V_{20}, V_{21}, V_{22}$. The user should ensure the matrix to be a valid correlation matrix, i.e. the elements to be within bounds, the matrix to be symmetric, and that the diagonal elements are unity, i.e. the following conditions should be fulfilled: $V_{ii} = 1$ and $-1 \le V_{ji} = V_{ij} \le 1$ for $i \neq j$, for all $i, j = 0, \ldots$ `NumEst - 1`. If the matrix is not symmetric, or off diagonal elements are outside their range of validity, the input is not consistent. In this case, an error message is issued and the software will refrain from combining. In any case, the diagonal elements will be forced to unity by the software.

Given the above relations, the entire information is contained in one half of the off diagonal elements (e.g. those marked in red in Eq. 10). To account for this, this function can also be called

with k replaced by $-k$ (for $k \neq 0$). In this case the array x should only contain the significant elements again in row-wise storage, i.e. in the above case $x = V_{01}, V_{02}, V_{12}$ is expected by the software. Again, if elements are outside their range of validity, the input is not consistent, an error message is issued and the software will refrain from combining.

`void FillCor(Int_t k, Double_t rho)`: Frequently uncertainty sources are either uncorrelated or fully correlated amongst all estimates. In this case, only a single value, namely the overall correlation obeying $-1 \leq$ `rho` $= \rho_k \leq 1$ is significant. A call to this function will store a correlation matrix with $V_{ii} = 1$ and $V_{ji} = V_{ij} = \rho_k$ for $i \neq j$, for $i, j = 0, \ldots,$ `NumEst - 1` for the source k. If the value of ρ_k is not within bounds, the input is not consistent, an error message is issued and the software will refrain from combining.

The following functions allow to assign names to estimates, uncertainties and observables. They are implemented as `TString` objects. The length of each name is arbitrary, however all printing functions and the display routine are optimised for names with equal length of seven characters. The type of characters can be freely chosen, however those requiring math mode should be avoided when using `LatexResult()` (see below). For all functions it is the responsibility of the user to ensure the correct length of the arrays of names, i.e. names for `NumEst` estimates, `NumUnc` uncertainties and `NumObs` observables should be provided. The functions can only be called before the end of input of estimates and correlations is recognised by the software. Therefore, it is recommended to first fill the names if wanted.

`void FillNamEst(TString* NamEst)`: A call to this function will store the names of the estimates.

`void FillNamUnc(TString* NamUnc)`: A call to this function will store the names of the uncertainties.

`void FillNamObs(TString* NamObs)`: A call to this function will store the names of the observables.

3.3 Fix and free input

`void FixInp()`: The input is fixed for solving and the calculation of several matrices is initiated.

`void ReleaseInp()`: The input is freed for additional selections. Any further selection starts from the situation at the last call to TvarFixInp().

`void ResetInp()`: The input is freed for additional selections. However, in this case any further selection starts from the original user input.

3.4 Solver

The default method for solving the problem is:

`void Solve()`: The BLUE combination for the presently active estimates and uncertainties is performed.

In the following a number of specific `Solve...()` functions are discussed which themselves call `FixInp()` and `Solve()` several times. As a consequence, after calling one of these functions the output of the print functions related to estimates and uncertainties will be different from the one after the last user call to `FixInp()`. In contrast, since these functions use `ReleaseInp()`, the situation in terms of active estimates, uncertainties and correlation assumptions remains unchanged. Exceptions are: `SolvePosWei()`, where estimates resulting in negative weights are disabled at return, and `SolveMaxVar()`, where the uncertainties are scaled, see `SetRhoFacUnc()` for details. For the user to get to a clean situation after using theses exceptions it is recommended to use `ResetInp()` before subsequent calls to `Solve()`.

`void SolveRelUnc(Double_t Dx)`: The BLUE combination is performed for the presently active estimates and uncertainties, of which at least one has to be a Rel-ative Unc-ertainty. Iterations are made until the relative difference of the combined value with respect to the one from the previous iteration falls below `Dx` percent.

The uncertainty sources can be an arbitrary mixture of relative or absolute uncertainties, see `SetRelUnc(...)` for how to steer this. The term absolute uncertainty means that the value of the uncertainty is identical for all possible values of the estimator pdf, i.e. it is independent of the actual value of the estimate. This means it is the same for the actual estimate, any combined value and the true value. Therefore, irrespectively whether it was calculated for the estimate it also applies to the combined value. In contrast, a relative uncertainty (e.g. of some percent) varies across the pdf and depends on the actual value of the estimate. Consequently, in this case after each iteration the uncertainty is replaced by the expected uncertainty of the true value x_{T}, approximated by the one of the combined value x.

The procedure works as follows: First a BLUE combination is performed. Then the uncertainties are adjusted based on the result and the next iteration is performed. This is repeated until convergence is reached. For each estimate i and each relative uncertainty k the dependence of the contribution from this source to the covariance matrix can be defined by the user as a second order polynomial in x. The function reads $\sigma_{ik}^2 = a_0 + a_1\,x + a_2\,x^2$. See `SetRelUnc(...)` for the details of the implementation and Ref. [5] for an example of a more complicate situation.

`void SolveAccImp(Int_t ImpFla, Double_t Dx) const`:
`void SolveAccImp(Double_t Dx) const`: For each observable a combination of the estimates is performed Acc-ording to their Imp-ortance. For the first implementation, three definitions of importance of the estimates j are implemented given the most precise estimate is i. The second uses `ImpFla = 0`.
The options implemented are:
1) `ImpFla = 0` means sorted by Eq. 2 using $12 = ij$
2) `ImpFla = 1` means sorted by the absolute BLUE weights $|\alpha_j|$
3) `ImpFla = 2` means sorted by inverse variance $1/\sigma_j^2$.

The software suggests which estimates to combine until the uncertainty of the combined value is never improved by more than `Dx` percent by adding further estimates. First a BLUE

combination for the presently active estimates and uncertainties is performed. For each active observable the related estimates are sorted by importance. According to this list one estimate at a time is added to the most precise one and the combination is performed, while all less important estimates of this observable are disabled. In contrast, all estimates of other observables are kept active such that the full correlation is preserved. This is repeated for all active observables. The outcome can be digested by a call to `PrintAccImp()`.

`void SolveScaRho(Int_t RhoFla, Double_t* MinRho, Double_t* MaxRho) const`:
`void SolveScaRho(Int_t RhoFla) const`:
xs `void SolveScaRho() const`: This function performs a scan in the correlation assumptions for all active estimates, uncertainty sources k, and observables, while using `NumFac` groups of multiplicative factors r, performing ten steps each in the range defined by `MaxRho` $> r >$ `MinRho`, while decreasing r. Non of the active uncertainties is allowed to be declared as `changed` or `reduced` uncertainty, see below `SetRhoXXXUnc()` for the definitions. While the groups l are always scanned independently, the sources k are scanned either independently for `RhoFla = 0`, or simultaneously for `RhoFla = 1`.

Given that the sources of uncertainty k in general are independent, because otherwise adding them quadratically to calculate the total uncertainty would not be correct, an independent scan, i.e. `RhoFla = 0` is recommended. If this is wanted, and the variation for all sources and groups (k, l) should be done in the range $1 > r > 0$ with respect to the initially provided correlation, the last implementation should be used. Otherwise the boundaries should be given in the following form: `MinRho(k=0 l=0, k=0 l=1, ..., k=NumUnc-1 l=NumFac-1)`.

Manipulations with many groups l that may end up in manipulating single entries of the covariance matrix, can easily lead to instable matrix inversions. The software is protected against this.

The procedure works as follows. First a combination is performed for the active estimates and uncertainties treating all uncertainties as scaled uncertainties, while using any given scale provided by preceding calls to `SetRhoFacUnc()`, and for $r = 1$. Then a scan is performed and the differences of the observables and their uncertainties with respect to the values from the initial result are stored. Inversion failures are indicated by values of -1.00 for both differences. Finally, the outcome can be digested by a call to `PrintScaRho()`.

`void SolveInfWei() const`: This function is only available for a single observable. It yields the same result as a call to `Solve()` but also calculates the information weights defined in [6]. The weights calculated are the BLUE weights α_i, the intrinsic weights, the marginal weights, the weight assigned to the correlation and finally the relative weights. These weights are defined as follows:

$$\begin{aligned} BLUE &= \alpha_i \\ intrinsic &= \frac{\sigma_x^2}{\sigma_i^2} \\ correlation &= 1 - \Sigma_i intrinsic \\ marginal &= 1 - \frac{\sigma_x^2}{\sigma_{xm-i}^2} \end{aligned}$$

$$relative \;=\; \frac{|\alpha_i|}{\Sigma_i|\alpha_i|}$$

Here σ^2_{xm-i} denotes the variance of the combination when using all m estimates, but the estimate i. The outcome can be digested by a call to `PrintInfWei()`. *NOTE:* It is recommended to *NOT* use relative weights when achieving scientific results because of the weakness of the concept, see Ref. [6] for a discussion. Here, they are only implemented to enable comparisons.

The following functions implement two alternative solving methods. *NOTE:* It is recommended to *NOT* use these functions when achieving scientific results because of the weakness of the concepts. Here, they are only implemented to enable comparisons.

`void SolvePosWei() const`: For each observable a combination is performed by including only estimates of this observable that have Pos-itive Wei-ghts and all other estimates of different observables. First a BLUE combination for the presently active estimates and uncertainties is performed. Then, all estimates that determine this observable, and have negative BLUE weights, are disabled and the next combination is performed. This is repeated until no estimates with negative weights remain.

`void SolveMaxVar(Int_t IFuRho) const`: This functions is only available for a single observable. Three methods are implemented to Max-imise the Var-iance of the combined result by changing, i.e. reducing the correlations of the systematic uncertainties in an artificial, but controlled way, see Ref. [6]. This is achieved by multiplying all covariance entries (i.e. the off diagonal elements of the contributions to the covariance matrix for the uncertainty source k) for $k > 0$ by factors f_{ijk}, thereby changing the initially assigned correlations. This procedure is not applied to the source $k = 0$, which is assumed to be the statistical uncertainty, which is either uncorrelated between estimates, or the correlations are exactly known, because they have been determined by the experiments as e.g. in Ref. [7].

The following options are implemented:
1) `IFuRho =` `0` means $f_{ijk} = f$ for all i, j, k,
2) `IFuRho =` $\pm$ `1` means $f_{ijk} = f_k$ for all i, j,
3) `IFuRho =` `2` means $f_{ijk} = f_{ij}$ for all k.

Since for each source k and pair i, j of estimates the dependence of the relative improvement in the uncertainty follows Figure 1(b), the factors f_{ijk} are obtained by a scan in the value of the respective factor using the range $1 \to 0$. The maximum is guaranteed to exist for $\rho_{ijk} = 1/z_{ijk} > 0$. Clearly, if the correlation initially assigned is such that it lies to the left of this point, the initial situation already corresponds to the maximum to be calculated, i.e. the real maximum is not attempted to be found in this procedure.

The algorithm works as follows: For `IFuRho = 0`, the global factor f is found by a scan from $1 \to 0$. For `IFuRho =` $\pm$ `1`, the f_k are obtained independently `IFuRho = 1`, (consecutively `IFuRho = -1`) for all sources $k > 0$, i.e. when determining f_k the values for sources k' with $k' \neq k$ are set to unity (their already found values). Finally, for `IFuRho = 2` the f_{ij} are found consecutively, while using the already determined values for $i' < i$ and $j' < j$. Given this procedure, the covariance matrix can be manipulated in such a way that the inversion gets unstable. The

software has been protected against this occurrence. Finally, the outcome can be digested by a call to `PrintMaxVar()`.

3.5 Setters

All setters are implemented in such a way that i and k always refer to their initial values for estimates and uncertainty sources that were given by the user during the filling step. This way the user does not need to keep track of the actual index an estimate or uncertainty has within the presently active list. The setters only work if the input is not fixed.

`void SetActiveEst(Int_t i)`: Enable estimate i, i.e. it will be used in subsequent calls to `Solve()`.

`void SetActiveUnc(Int_t k)`: Enable uncertainty k, i.e. it will be used in subsequent calls to `Solve()`.

`void SetInActiveEst(Int_t i)`: Disable estimate i, i.e. it will not be used in subsequent calls to `Solve()`.

`void SetInActiveUnc(Int_t k)`: Disable uncertainty k, i.e. it will not be used in subsequent calls to `Solve()`.

`void SetRhoValUnc(Double_t RhoVal)`:
`void SetRhoValUnc(Int_t k, Double_t RhoVal)`:
`void SetRhoValUnc(Int_t k, Int_t l, Double_t RhoVal)`: The first implementation of this function will set the correlations of all active uncertainty sources and all groups l to `RhoVal`. This value should be within the range $-1 <$ `RhoVal` < 1. The second will do the same, but only for the source k. The third one only applies to the group l of source k. See the constructor for the definition of the groups l.

`void SetNotRhoValUnc()`:
`void SetNotRhoValUnc(Int_t k)`: The first implementation of this function will revert to the originally provided correlations of all active uncertainty sources. The second will do the same, but only for the source k.

`void SetRhoFacUnc(Double_t RhoFac)`:
`void SetRhoFacUnc(Int_t k, Double_t RhoFac)`:
`void SetRhoFacUnc(Int_t k, Int_t l, Double_t RhoFac)`: The first implementation of this function will scale the originally provided correlations of all active uncertainty sources and all groups l by a factor `RhoFac`. This factor should be within the range $-1 <$ `RhoFac` < 1. The second will do the same, but only for the source k. The third one only applies to the group l of source k. See the constructor for the definition of the groups l. Clearly, uncorrelated sources are not affected by this.

`void SetNotRhoFacUnc()`:
`void SetNotRhoFacUnc(Int_t k)`: The first implementation of this function will revert to the originally provided correlations of all active uncertainty sources. The second will do the same, but only for the source k.

The following functions implement the so called *reduced correlations*[1]. *NOTE:* It is recommended to *NOT* use these functions when achieving scientific results because of the weakness of the concept. Here, they are only implemented to enable comparisons.

`void SetRhoRedUnc()`:
`void SetRhoRedUnc(Int_t k)`: For all active uncertainty sources and all **fully correlated** pairs of estimates, the first implementation of this function will replace the correlation by the reduced correlation. The second will do the same, but only for the source k.

`void SetNotRhoRedUnc()`:
`void SetNotRhoRedUnc(Int_t k)`: The first implementation of this function will revert to the originally provided correlations of all active uncertainty sources. The second will do the same, but only for the source k.

By construction, changed- scaled- and reduced correlations are mutually exclusive. Consequently, for each source of uncertainty the use of only one of the options is supported by the software.

The following functions allow to steer which uncertainties are taken as relative and which as absolute in subsequent calls to `SolveRelUnc()`.

`void SetRelUnc()`:
`void SetRelUnc(Int_t k)`: The first implementation of this function will declare all active uncertainty sources as relative uncertainties. The second will do the same, but only for the source k. In this implementation the default behaviour of the detailed implementation discussed next is used for all estimates and the respective uncertainty source.

`void SetRelUnc(Int_t i, Int_t k, Double_t* ActCof)`: For each estimate i and each uncertainty source k the dependence of the variance on the combined value x is defined by using the coefficients from the array `ActCof` $= \{a_0, a_1, a_2\}$ in the second order polynomial: $\sigma_{ik}^2 = a_0 + a_1\,x + a_2\,x^2$.

In the default implementation it is assumed that the statistical uncertainty (estimate) is proportional to $\sqrt{N}$ (N), where N is the number of events, and the systematic uncertainties to be linear in x. Consequently, in this case only one coefficient each is different from zero. For the statistical uncertainty ($k = 0$) this is $a_1 = \sigma_{i0}^2/x_\mathrm{i}$, and for all systematic uncertainties $k > 0$

[1]Reduced correlations assume that for each pair (i, j) of estimates and a given source of uncertainty k the smaller of the individual uncertainties, e.g. $\sigma_{1k} < \sigma_{2k}$, is fully correlated, and the remainder is uncorrelated. This replaces the covariance $\rho_{12k}\sigma_{1k}\sigma_{2k}$ by the square of the smaller of the individual uncertainties σ_{1k}^2 for this source, which is equivalent to assuming the correlation to amount to the ratio of the smaller to the larger uncertainty, $\rho_{12k} = \sigma_{1k}/\sigma_{2k} = 1/z_k$, see Section 1 for the consequences of this.

it is $a_2 = \sigma_{ik}^2/x_{\mathrm{i}}^2$. If this behaviour is valid for the combination under investigation, a single call to `void SetRelUnc()` should be used, otherwise individual user defined functions have to be provided. If this is needed, for any uncertainty source k the functions for all estimates i have to be given.

`void SetNotRelUnc()`:
`void SetNotRelUnc(Int_t k)`: The first implementation of this function will declare all active uncertainty sources as absolute uncertainties and revert to the initially provided values. The second will do the same, but only for the source k.

3.6 Getters

3.6.1 Getters for active estimates and uncertainties

The following functions give access to the actual numbers of active estimates, uncertainties and observables. This information is only available after a call to `FixInp()`, otherwise the return value is zero.

`Int_t GetActEst() const`: Returns the number of active estimates.

`Int_t GetActUnc() const`: Returns the number of active uncertainties.

`Int_t GetActObs() const`: Returns the number of active observables. Although the interface does not allow to disable observables, still this number will differ from the value of `NumObs` originally supplied, whenever all estimates determining one of the observables have been deactivated by calling `SetInActiveEst()`.

The following functions give access to the names of the active estimates, uncertainties and observables. This information is only available after a call to `FixInp()`, otherwise, as well as for inactive estimates, the return value is `NULL`.

`TString GetNamEst(Int_t i) const`: Returns the name of the active estimate i.

`TString GetNamUnc(Int_t k) const`: Returns the name of the active uncertainty k.

`TString GetNamObs(Int_t n) const`: Returns the name of the active observable n.

The following functions give access to the actual lists of estimates, uncertainties and observables. Again, this information is only available after a call to `FixInp()`. In this case the return value is 1 otherwise it is 0. These functions return a pointer to the first element of an array of `Int_t` values. The structures are filled always starting from element 0. The dimensions are dynamical, i.e. they depend on the number of active estimates, uncertainties and observables that may well differ from the dimensions originally supplied to the constructor of the class. As a consequence, if the structures are defined by the user and filled using the original dimensions, the last part of the structures will contain senseless non zero values, whenever estimates or un-

certainty sources are disabled and the functions are called a second time.

`Int_t GetIndEst(Int_t* IndEst) const`: Returns the list of active estimates. The dimension is: IndEst(GetActEst()).

`Int_t GetIndUnc(Int_t* IndUnc) const`: Returns the list of active uncertainties. The dimension is: IndUnc(GetActUnc()).

`Int_t GetIndObs(Int_t* IndObs) const`: Returns the list of active observables. The dimension is: IndObs(GetActObs()).

`Int_t GetPreEst(Int_t n) const`: Returns the index i of most Pre-cise Est-imate for observable n.

The following functions give access to various quantities for the active estimates and uncertainties. See above for their availability and return values. These functions come in pairs and return a pointer to either a `TMatrixD` or the first element of an array of `Double_t` values. The structures are filled always starting from element $(0, 0)$ or 0. The dimensions of the matrices are given below, the dimension of the arrays should be the product of the number of columns and rows of the matrices. The user has to take care of the proper dimension of the structure in the calling function. Also here the dimensions are dynamical (see above for the consequences).

`Int_t GetCov(TMatrixD* UseCov) const`:
`Int_t GetCov(Double_t* RetCov) const`: Returns the covariance matrix of the estimates. The dimension is: UseCov(GetActEst(),GetActEst()).

`Int_t GetCovInvert(TMatrixD* UseCovI) const`:
`Int_t GetCovInvert(Double_t* RetCovI) const`: Returns the inverse of the covariance matrix of the estimates. The dimension is: UseCovI(GetActEst(),GetActEst()).

`Int_t GetRho(TMatrixD* UseRho) const`:
`Int_t GetRho(Double_t* RetRho) const`: Returns the correlation matrix of the estimates. The dimension is: UseRho(GetActEst(),GetActEst()).

`Int_t GetParams(Int_t Ifl, TMatrixD* UseParams) const`:
`Int_t GetParams(Int_t Ifl, Double_t* RetParams) const`: Returns the matrices of parameters for hypothetical pairwise combinations. See `PrintParams()` for the meaning of `Ifl`. The dimension is: UseParams(GetActEst(),GetActEst()).

3.6.2 Getters for the consistency of the combination

This information is only available after a call to `Solve()`, otherwise the return value of the functions is zero.

`Double_t GetChiq() const`: Returns the χ^2 value of the result.

`Int_t GetNdof() const`: Returns the number of degrees of freedom N_{dof}, i.e. the difference of the number of active estimates and active observables.

`Double_t GetProb() const`: Returns the χ^2 probability $P(\chi^2, N_{\mathrm{dof}})$ of the result.

`Double_t GetPull(Int_t i) const`: Returns the pull of the estimate i. The pull is defined as the difference of the estimate and the observable, divided by the square root of the difference of the variances of the two.

3.6.3 Getters for active observables

The following functions give access to various quantities for results for the active observables that are obtained from the combination of the active estimates given their active uncertainties. Again, this information is only available after a call to `Solve()`. Also here, this is indicated by the return value of the integer function, which is 1 if successful, i.e. `Solve()` was called, and 0 otherwise. These functions also come in pairs.

`Int_t GetCovRes(TMatrixD* UseCovRes) const`:
`Int_t GetCovRes(Double_t* RetCovRes) const`: Returns the covariance matrix of the observables. The dimension is: UseCovRes(GetActObs(),GetActObs()).

`Int_t GetRhoRes(TMatrixD* UseRhoRes) const`:
`Int_t GetRhoRes(Double_t* RetRhoRes) const`: Returns the correlation matrix of the observables. The dimension is: UseRhoRes(GetActObs(),GetActObs()).

`Int_t GetWeight(TMatrixD* UseWeight) const`:
`Int_t GetWeight(Double_t* RetWeight) const`: Returns the matrix of the BLUE weights of the estimates for the various observables. The dimension is: UseWeight(GetActEst(),GetActObs()).

`Int_t GetResult(TMatrixD* UseResult) const`:
`Int_t GetResult(Double_t* RetResult) const`: Returns the matrix of the results of the observables in a form similar to what is expected for the filling of the estimates in `FillEst()` described above. Each observable is stored in one row, where the first element is the value, followed by the individual uncertainties. The dimension is: UseResult(GetActObs(),GetActUnc()+1).

`Int_t GetUncert(TMatrixD* UseUncert) const`:
`Int_t GetUncert(Double_t* RetUncert) const`: Returns the matrix of the total uncertainties of the observables. The dimension is: UseUncert(GetActObs(),1).

`Int_t GetNumScaFac() const`:
Returns the number of groups of correlations l defined in the constructor.

`Int_t GetNumScaRho() const`:

Returns the number of steps in the correlations used in `SolveScaRho`, which is ten.

`Int_t GetScaVal(Int_t n, TMatrixD* UseScaVal) const`:
`Int_t GetScaVal(Int_t n, Double_t* RetScaVal) const`: Returns the result of `SolveScaRho()` for the differences in the values of the observable n. The dimension is:
UseScaVal(GetActUnc()*GetNumScaFac(),GetNumScaRho()).

`Int_t GetScaUnc(Int_t n, TMatrixD* UseScaUnc) const`:
`Int_t GetScaUnc(Int_t n, Double_t* RetScaUnc) const`: Returns the result of `SolveScaRho()` for the differences in the uncertainties of the observable n. The dimension is:
UseScaUnc(GetActUnc()*GetNumScaFac(),GetNumScaRho()).

3.7 Print out

The software provides some print out during the various steps. Naturally, printing more information helps developing the user functions, but afterwards it only distracts from the important information. Consequently, the level of details reported to the user can be steered.

`void SetPrintLevel(Int_t p)`: Set the level of details for the print out $0 \leq p \leq 2$, with increasing details for increasing values of p.

There exist four groups of `Print` functions. The first group returns information related to the presently active estimates and uncertainties, and the second information related to the result for the observables. Given the flexibility of the `Set` functions described above, they only reflect the correct status after a call to `FixInp()` or `Solve()`, respectively. Consequently, they are disabled until the respective function has been called.

The third group consists of just one function that shows the present status of the input and the results. Depending on the print level it calls a number of functions from the first two groups. Finally, the fourth group returns the finding of specific `Solve...()` functions described above.

3.7.1 Print functions for active estimates

Again, also for the print functions, the indices i, k refer to the original estimate and uncertainty source indices, respectively.

`void PrintListEst() const`: Prints the list of the active estimates.

`void PrintListUnc() const`: Prints the list of the active uncertainties.

`void PrintNamEst() const`: Prints the names of the active estimates.

`void PrintNamUnc() const`: Prints the names of the active uncertainties.

`void PrintEst(Int_t i) const`: Prints the information for the active estimate i.

`void PrintEst() const`: Prints the information for all active estimates.

`void PrintCofRelUnc() const`:
`void PrintCofRelUnc(Int_t k) const`: The first implementation of this function will print for all active estimates the coefficients for all active and relative uncertainties. The second will do the same, but only for the uncertainty source k.

`void PrintCor(Int_t k) const`: Prints the correlation matrix of the active uncertainty source k.

`void PrintCor() const`: Prints the correlation matrix of all active uncertainty sources.

`void PrintCov(Int_t k) const`: Prints the contribution of the uncertainty source k to the covariance matrix of the active estimates.

`void PrintCov() const`: Prints the covariance matrix of the active estimates.

`void PrintCovInvert() const`: Prints the inverted covariance matrix of the active estimates.

`void PrintRho() const`: Prints the correlation matrix of the active estimates.

`void PrintCompatEst() const`: Prints the pair wise compatibility of the estimates of the same observable given their correlation. The compatibility is based on a χ^2 and the corresponding probability using $P(\chi^2, N_{\rm dof}) = P(\chi^2, 1)$. The χ^2 is defined as $\chi^2 = (x_1 - x_2)^2/(\sigma_1^2 + \sigma_2^2 - 2\,\rho_{12}\,\sigma_1\,\sigma_2)$.

`void PrintParams(Int_t Ifl) const`: Prints the matrices of parameters for hypothetical pairwise combinations of the estimates i and j provided they determine the same observable. If not, for this pair zero is returned instead. Given the symmetry, only the lower half of the matrix is filled. The parameter printed depends on the value of `Ifl`. For `Ifl = 0` the ratio of the uncertainties is returned, i.e. σ_i/σ_j with $j > i$. This ratio corresponds to z if $\sigma_i > \sigma_j$ and $1/z$ otherwise. For `Ifl > 0` the values of Eqs. 1–6 are returned.

`void PrintPull(Int_t i) const`: Prints the pull of the active estimate i.

`void PrintPull() const`: Prints the pull of all active estimates.

3.7.2 Print functions for the observables

`void PrintListObs() const`: Prints the list of active observables.

`void PrintNamObs() const`: Prints the names of the active observables.

`void PrintCovRes() const`: Prints the covariance matrix of the results for all observables.

`void PrintRhoRes() const`: Prints the correlation matrix of the results for all observables.

`void PrintWeight() const`: Prints the weight matrix of the results for all observables (Columns) and estimates (Rows).

`void PrintResult() const`: Prints the result for each observable. First the linear combination of the individual estimates is given. Then the combined values for the observables are listed together with the full breakdown of their uncertainties. Finally, the separation into the statistical uncertainty ($k = 0$) and the total systematic uncertainty (the square root of the quadratic sum of the contributions from all sources with $k > 0$) is given.

`void PrintCompatObs() const`: Prints the pair wise compatibility of the observables given their correlation. For the definition of the χ^2 used see the explanation of `PrintCompatEst()`. Obviously, here this compatibility only makes sense if the observables should coincide, i.e. they relate to the same physics parameter. If that is not the case this information should be ignored.

`void PrintChiPro() const`: Prints the χ^2 together with the number of degrees of freedom N_{dof}, and the χ^2 probability $P(\chi^2, N_{\mathrm{dof}})$ of the result.

3.7.3 Print functions for the overall status

`void PrintStatus() const`: Prints the status of input and output depending on the state like: fixed or solved and the print level.

3.7.4 Print functions for specific Solve...() functions

`void PrintAccImp() const`: Prints the findings of `SolveAccImp(Dx)`. The order of importance is given. For each observable, the parameters for the hypothetical pair-wise combinations of each of its estimate with the most precise one is given. The change is reported for the combined value and its uncertainty while including the estimates one by one according to importance in the combination. Finally, the list of estimates to be used in the combination is given that corresponds to the relative improvement `Dx` requested in the call to `SolveAccImp(Dx)`.

`void PrintScaRho() const`: Prints the differences in the values and uncertainties for all observables obtained in the correlation scan performed by `SolveScaRho(RhoFla)`. The matrix with the remaining correlation groups l is given. The ranges in r used are listed per uncertainty source k, and group of correlation l. The number of inversion failures is reported if needed. For each observable the differences in the values and uncertainties are reported per source k and for all 10 values of r. Finally, the total differences are reported with the following meaning. For the independent scan, i.e. `RhoFla = 0` the total is the quadratic sum of all sources ignoring inversion failures, i.e. entries reported as -1.00. In contrast, for the simultaneously scan, i.e. `RhoFla = 1` the total coincides with the last line of the previously accumulated result.

`void PrintInfWei() const`: Prints the information weights defined above in the description of `SolveInfWei()` for the active estimates.

`void PrintMaxVar() const`: Prints the findings of `SolveMaxVar()`. The variance of the combined result and the correlation matrix of the estimates are given before and after the maximisation of the variance. In addition listed are the number of times an unstable matrix inversion has been detected. Finally, the fitted factors are given, depending on the value of `IFuRho` used in the call to `SolveMaxVar(IFuRho)`.

3.8 Utilities

For the special situation of two estimates of a single observable discussed above, the data can be inspected more closely. Two sets of functions are implemented. The first set is independent of the data structure. The second set (at present containing only a single function) works on a pair of active estimates. Both sets are discussed in turn.

In addition, a utility is provided to inspect the situation in the case of instable matrix inversions. Finally, for publishing the results two utilities to create LaTeX and PDF output are provided.

3.8.1 Data structure independent utilities for a pair of estimates

The first two function can be used for arbitrary values of ρ and z, to either evaluate Eqs. 1–6, or to produce figures analogous to Figures 1(a)–2(d).

`Double_t GetPara(Int_t ifl, Double_t rho, Double_t zva) const`: Returns for given values of $rho = \rho$ and $zva = z$ the values of Eq. ifl.

`Double_t FunPara(Double_t* x, Double_t* par) const`: This function implements the possibility to use `GetPara()` as a TF1 function. The meaning of the parameters is as follows: for all cases `par[1] = ifl`. For the situation that z is a parameter, and ρ is the function variable, as e.g. in Eq. 1, $par[0] = z$ and $x = \rho$. For the situation that ρ is a parameter, and z is the function variable, as e.g. in Eq. 6, the situation is reversed, i.e. $par[0] = \rho$ and $x = z$.

For the user to implement this as a TF1 function the following notation should be used:
TF1* Func = new TF1(FuncName,this,&Blue::FunPara,xlow,xhig,2,"Blue","FunPara");
Only when this syntax is followed the normal ROOT methods for TF1 functions can be used.

Finally, the last utility exploits the characteristics of an arbitrary pair of estimates.

`void DrawSens(Double_t xv1, Double_t xv2, Double_t sv1, Double_t sv2, Double_t rho, TString FilNam) const`: The required input is $xv1 = x_1$, $xv2 = x_2$, $sv1 = \sigma_1$, $sv2 = \sigma_2$, $rho = \rho$, see Section 1 for details. The same up and down variations of ρ and z as discussed below for `InspectPair()` are performed, and the result is visualised in a figure similar to Figure 3. This figure is finally stored in the file `FilNam.pdf`.

3.8.2 Data structure dependent utility for a pair of estimates

`void InspectPair(Int_t i,Int_t j) const`:
`void InspectPair(Int_t i,Int_t j, TString FilNam) const`: The pair of active estimates i, j is inspected more closely. When the second implementation is invoked also `DrawSens` (see above) is called.

First, the compatibility of the estimates is evaluated. If the estimates are not consistent, no combination should be performed! Then the actual combination is performed and the values of Eqs. 1–6 are reported. Subsequently, the parameters ρ and z are varied by about $\pm 10\%$ in the following way. A variation of ± 0.1 is attempted in ρ. In addition, the variation is restricted to stay within $-0.99 < \rho < 0.99$ such that, depending on the initial value of ρ, the actual range may be smaller. Similarly, for z an upward variation to $z_{\rm up} = 1.1 \cdot z$ is performed. The downward variation to $z_{\rm dn} = 0.9 \cdot z$ is further restricted to not fall below the minimum of $z_{\rm dn} = 1.01$. This ensures that x_1 remains the more precise estimate. The combination is repeated for all possible pairs of values using the three cases each for $(z_{\rm dn}, z, z_{\rm up})$ and $(\rho_{\rm dn},\ \rho,\ \rho_{\rm up})$. All nine results and the observed range in x and σ_x are reported.

3.8.3 Utility to inspect instable matrix inversions

`Int_t InspectResult() const`:
For some of the solving methods, especially when manipulating individual elements of the covariance matrix, unstable matrix inversions can occur. At present, three non exclusive situations are distinguished. Firstly, an individual uncertainty of an observable gets negative or its evaluation results in a `-nan` value, secondly the same happens to the total uncertainty of an observable, and thirdly the total uncertainty of an observable is larger than the one of its most precise estimate. The return value of the function indicates which of the situations occurred. Starting from an initial return value of zero, in the first, second and third situation, -1, -10 and -100 is added to it.

For a user call to `Solve()`, in any of these cases a message is issued by the software. If this occurs, the situation can be inspected by setting the print level to greater than zero and calling `InspectResult()`, which will also report the occurrence of negative Eigenvalues of the covariance matrix if present.

3.8.4 Utilities for publishing

`void LatexResult(TString FilNam) const`:
`void LatexResult(TString FilNam, TString ForVal, TString ForUnc, TString ForWei, TString ForRho, TString ForPul) const`: Creates a LaTeX file `FilNam.tex` with a number of tables. The tables provided are: one table with the active estimates together with the observables, one with the correlations of the estimates, one with the blue weights and the pulls and finally, for `NumObs>1`, one table with the correlation of the observables.

The first implementation uses default formats `ForXxx` where `Xxx` stands for the Val-ues, Uncertainties, Wei-ghts, Correlations (Rho), and finally the Pul-ls. The formats used are: %5.2f for values and uncertainties, and %4.2f for weights, correlations and pulls. If these are not suitable for the case under study they can be individually provided by the user using the second implementation. After creation, this file can be processed from the shell using the local LaTeX

implementation.

`void DisplayResult(Int_t n, TString FilNam) const`:
`void DisplayResult(Int_t n, TString FilNam, TString ForVal, TString ForUnc) const`: Creates a function `FilNam_Obs_n.cxx` which after compiling (like any of the examples listed below) produces a file `FilNam_Obs_n.pdf` with a figure containing the active estimates that determine the observable n together with the result of the combination. For the definition of the formats `ForXxx` see the description for `LatexResult()`.

4 Examples

To demonstrate the usage of the software a number of example functions are provided. They reproduce the numerical values of all combinations performed in the respective publication (but for differences that are explained below). In some cases a few more combinations are performed based on the information contained in the original publications. In addition, the functions show examples of how to retrieve the results into local data structures. The examples are listed in the following:

`B_NIMA_270_110.cxx()`: Function that reproduces all results discussed in [1].

`B_NIMA_500_391.cxx(Int_t Flag)`: Function that reproduces all results discussed in [2].

`B_EPJC_72_2046.cxx(int Flag)`: Function that reproduces all results discussed in [7].

`B_Peelles.cxx()`: Function that reproduces Peelle's Puzzle, see [**?**].

`B_arXiv_1107_5255.cxx(int Flag)`: Function that reproduces the 2011 (v3) combination of the Tevatron results on the top quark mass [8].

`B_arXiv_1305_3929.cxx(int Flag)`: Function that reproduces the 2013 (v2) combination of the Tevatron results on the top quark mass [9].

`B_arXiv_1307_4003.cxx(int Flag)`: Function that reproduces the results in [6]. (A different minimum with respect to the one quoted in Table 6 is found for the maximisation of the variance for `IFuRho = 3`. See the code for further details.)

`B_ATLAS_CONF_2012_095.cxx(int Flag)`: Function that reproduces the 2012 combination of the LHC results on the top quark mass [10].

`B_ATLAS_CONF_2012_134.cxx(int Flag)`: Function that reproduces the 2012 combination of the LHC results on the cross-section of top quark pair production [11].

`B_ATLAS_CONF_2013_033.cxx(int Flag)`: Function that reproduces the 2013 combination of the LHC results on the W-boson polarisation in top quark pair events [12]. (Some discrepancies with

respect to the published Tables 6 and 7 were found and are under investigation with the authors.)

`B_ATLAS_CONF_2013_098.cxx(int Flag)`: Function that reproduces the 2013 combination of the LHC results on the single top-quark cross-section in the t-channel [13] using the BLUE method with relative uncertainties.

`B_ATLAS_CONF_2013_102.cxx(int Flag)`: Function that reproduces the 2013 combination of the LHC results on the top-quark mass [14]. (A typo for the χ^2 value quoted in Table 4 was found and has been acknowledged by the authors.)

`B_PRD41_982.cxx(int Flag)`: Function that reproduces the combination of Ref. [14] using the BLUE method with individual relative uncertainties.

For each example `B_name.cxx` a script `B_name.inp` is provided that enables the creation of an output file for that example by typing: `root -b < B_name.inp > B_name.out`. To further ease the usage, two shell scripts `BlueOne` and `BlueAll` are provided. A single example is run by typing `BlueOne B_name` at the shell prompt. To use all input files `B_name.inp` in the current directory simply type `BlueAll` at the shell prompt. To verify the absence of programming mistakes within the user software that can be detected by the compiler also `CompOne` and `CompAll` are provided. They should be used in an analogous way to `BlueOne` and `BlueAll`, but this time to compile `B_name.cxx`.

5 Conversion of input files

To facilitate the conversion for users that have been working with the Fortran software [3], a utility is provided that takes a corresponding ASCII input file and converts it to a function that is similar to the examples listed above.

`void ForttoBlue(TString FilNam, TString ForVal, TString ForRho) const`: This function uses the input file `FilNam.in` and creates a file `B_FilNam.cxx` together with a corresponding steering file `B_FilNam.inp`. Afterwards `B_FilNam.cxx` can be expanded by the user and finally, it should be used the same way as the examples described in Section 4.

Running the Fortran software on `FilNam.in` should give the same result than what is obtained using `B_FilNam.cxx`. The format statement `ForVal` applies to the write statements for the estimates and uncertainties, and `ForRho` to the entries in the correlation matrices. See `LatexResult()` for a more detailed description of the meaning. Since this utility performs formatted reading from a file, strict requirements on the content of `FilNam.in` are imposed, e.g. blanks in names are not supported. The full list of requirements is listed when running `ForttoBlue()`. The function `ForttoBlue()` reports the findings during execution, such that in the case of failures the input files should be easily adaptable.

The utility works for the `FilNam.in` files that I use. In addition, to ease the usage, an example input file `EPJC_72_2046Fort.in` is provided together with `ForttoBlue.inp`. After creating the function `B_EPJC_72_2046Fort.cxx` with `ForttoBlue()`, the result from the Fortran software on `EPJC_72_2046Fort.in`, as well as those from running the newly created function for `Flag` $= 0$,

i.e. `B_EPJC_72_2046Fort(0)` or the distributed example `B_EPJC_72_2046(0)`, are identical.

6 Hints on the software installation

The software version x.y.z is distributed via the corresponding hepforge project page [15] as a gziped tar file named Blue-x.y.z.tar.gz, where the present version is $x.y.z = 1.9.0$. The software is not expected to depend on the installed version of the ROOT package. It has been compiled with a number of ROOT versions, however, most tests have been performed with ROOT 5.34/04. To install and use it perform the following steps:

1. To unzip the file: `gzip -d Blue-1.9.0.tar.gz`
2. To untar the file: `tar -xf Blue-1.9.0.tar`
3. To compile the class: `make`
4. Start ROOT
5. To load the Blue library: `gSystem->Load("libBlue.so");`
6. To get access to any of the example functions: e.g. `.L B_EPJC_72_2046.cxx++`
7. To execute a specific combination of this example: `B_EPJC_72_2046(1)`

For a more automated usage see the above descriptions of `BlueOne` and `BlueAll`. Finally, using the script `Install` a version x.y.z can be installed and the examples run by typing `Install Blue-x.y.z.tar.gz`.

In addition to the interface described in this manual, the software contains a number of `private:` member functions. However, differently from regular `C++` code, when the ACLiC system is used for the examples as suggested above, these member functions are **not** prohibited from being used outside of the class. Clearly, using those functions is strongly discouraged and can lead to unexpected results.

7 Conclusions

In this manual, a software package to perform the combination of several estimates of a number of observables was presented. The software is freely available from the corresponding hepforge project page. Given it is based on ROOT, it is distributed under the GNU Lesser General Public License. When using this software in publications, please give reference to the web page. Should you spot any mistake or peculiarity, please inform the author. If you want to be informed about new versions of the software by e-mail, let me know, either via the hepforge page or by direct e-mail.

Acknowledgements

I like to thank Sven Menke and Giorgio Cortiana for useful discussions on the project and their assistance. I am grateful to Sven for his valuable help on implementation issues, and to Giorgio for using the code and providing feedback.

A The BLUE formulas for two estimates

In the following Eq. 1 and Eq. 2 are derived with the BLUE formalism. The covariance matrix for the solutions of the linear combinations in the BLUE formalism is given by Eq. 5 of Ref. [2]. For the two estimate case of one observable this reduces to:

$$\sigma_x^2 = \begin{pmatrix} 1-\beta & \beta \end{pmatrix} \cdot \begin{pmatrix} \sigma_1^2 & \rho\,\sigma_1\,\sigma_2 \\ \rho\,\sigma_1\,\sigma_2 & \sigma_2^2 \end{pmatrix} \cdot \begin{pmatrix} 1-\beta \\ \beta \end{pmatrix} \tag{11}$$

dividing by σ_1^2 and using $z = \sigma_2/\sigma_1$ yields:

$$\frac{\sigma_x^2}{\sigma_1^2} = \begin{pmatrix} 1-\beta & \beta \end{pmatrix} \cdot \begin{pmatrix} 1 & \rho z \\ \rho z & z^2 \end{pmatrix} \cdot \begin{pmatrix} 1-\beta \\ \beta \end{pmatrix} \tag{12}$$

multiplication yields:

$$\begin{aligned} \frac{\sigma_x^2}{\sigma_1^2} &= (1-\beta)^2 + 2\rho z\beta(1-\beta) + \beta^2 z^2 \\ &= 1 - 2\beta(1-\rho z) + \beta^2(1 - 2\rho z + z^2) \end{aligned} \tag{13}$$

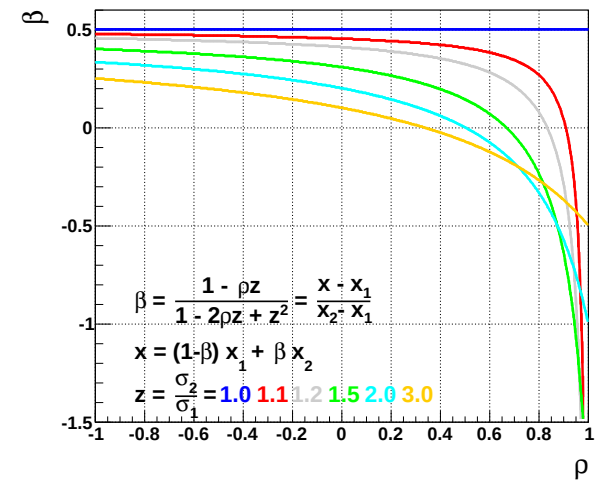
taking the derivative with respect to β equal to zero yields:

$$\frac{\partial}{\partial\,\beta}\left(\frac{\sigma_x^2}{\sigma_1^2}\right) = -2(1-\rho z) + 2\beta(1 - 2\rho z + z^2) = 0 \tag{14}$$

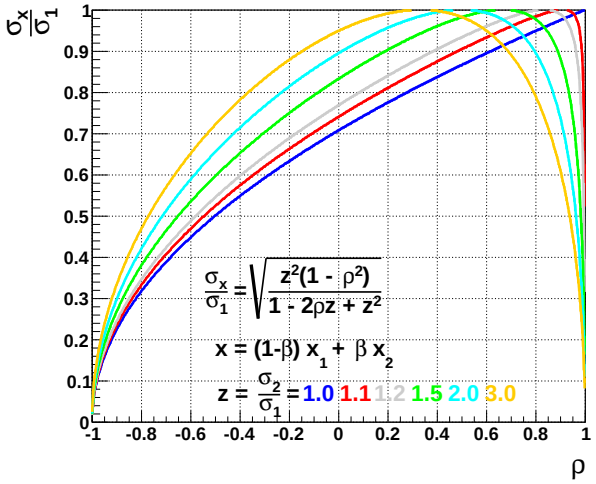
Finally, solving for β gives Eq. 1. Inserting the result for β into Eq. 13 yields:

$$\begin{aligned} \frac{\sigma_x^2}{\sigma_1^2} &= 1 - 2\frac{(1-\rho z)^2}{1 - 2\rho z + z^2} + \frac{(1-\rho z)^2}{1 - 2\rho z + z^2} \\ &= \frac{(1 - 2\rho z + z^2) - (1-\rho z)^2}{1 - 2\rho z + z^2} \end{aligned} \tag{15}$$

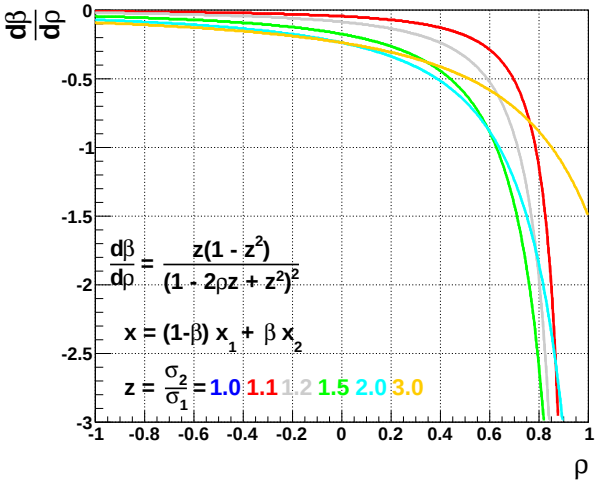
which after evaluating the numerator terms yields Eq. 2.



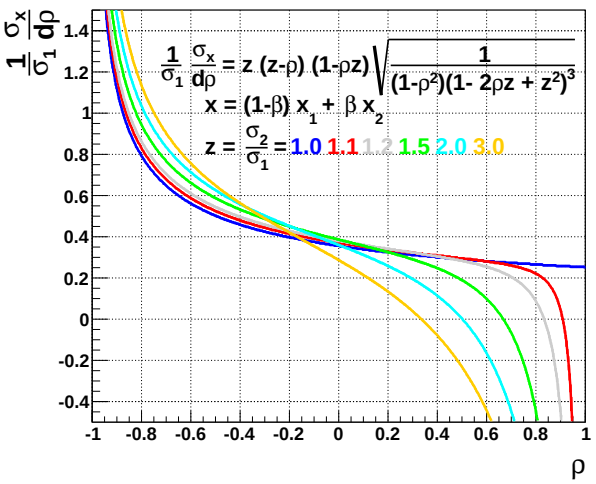
(a) β as a function of ρ



(b) σ_x/σ_1 as a function of ρ

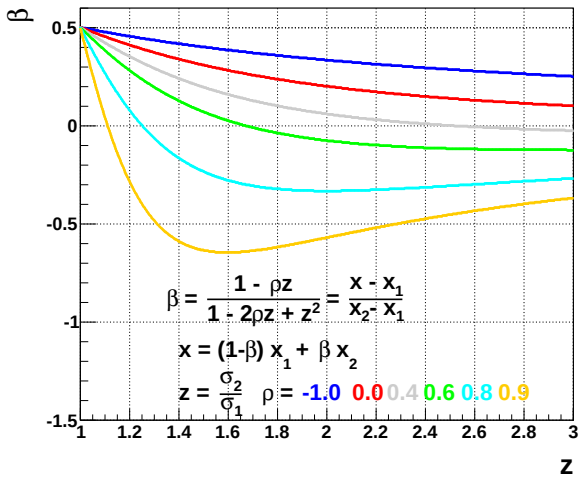


(c) $\mathrm{d}\beta/\mathrm{d}\rho$ as a function of ρ

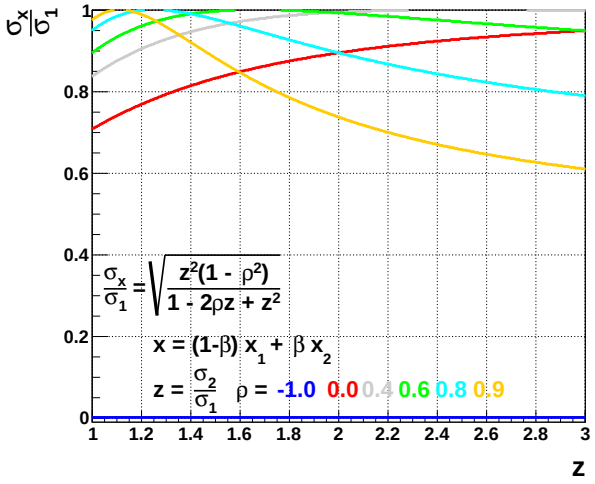


(d) $1/\sigma_1\ \mathrm{d}\sigma_x/\mathrm{d}\rho$ as a function of ρ

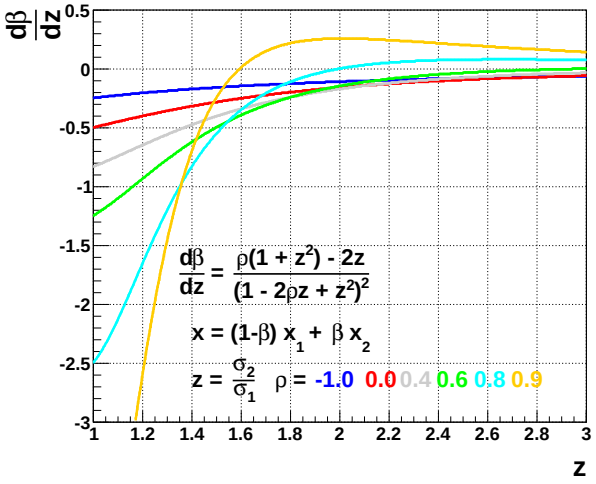
Figure 1: Several variables as a function of ρ for a number of z values. Shown are (a) β and (b) σ_x/σ_1 together with there derivatives with respect to ρ, (c) $\mathrm{d}\beta/\mathrm{d}\rho$ and (d) $1/\sigma_1\ \mathrm{d}\sigma_x/\mathrm{d}\rho$



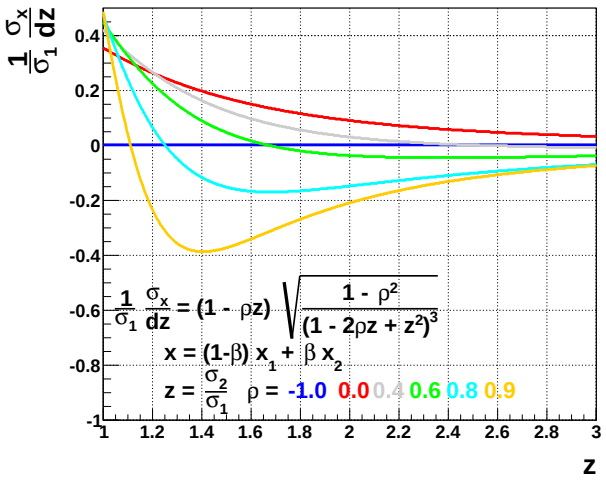
(a) β as a function of z



(b) σ_x/σ_1 as a function of z



(c) $\mathrm{d}\beta/\mathrm{d}\rho$ as a function of z



(d) $1/\sigma_1\ \mathrm{d}\sigma_x/\mathrm{d}\rho$ as a function of z

Figure 2: Several variables as a function of z for a number of ρ values. Shown are (a) β and (b) σ_x/σ_1 together with their derivatives with respect to z in (c) $\mathrm{d}\beta/\mathrm{dz}$ and (d) $1/\sigma_1\ \mathrm{d}\sigma_x/\mathrm{dz}$.

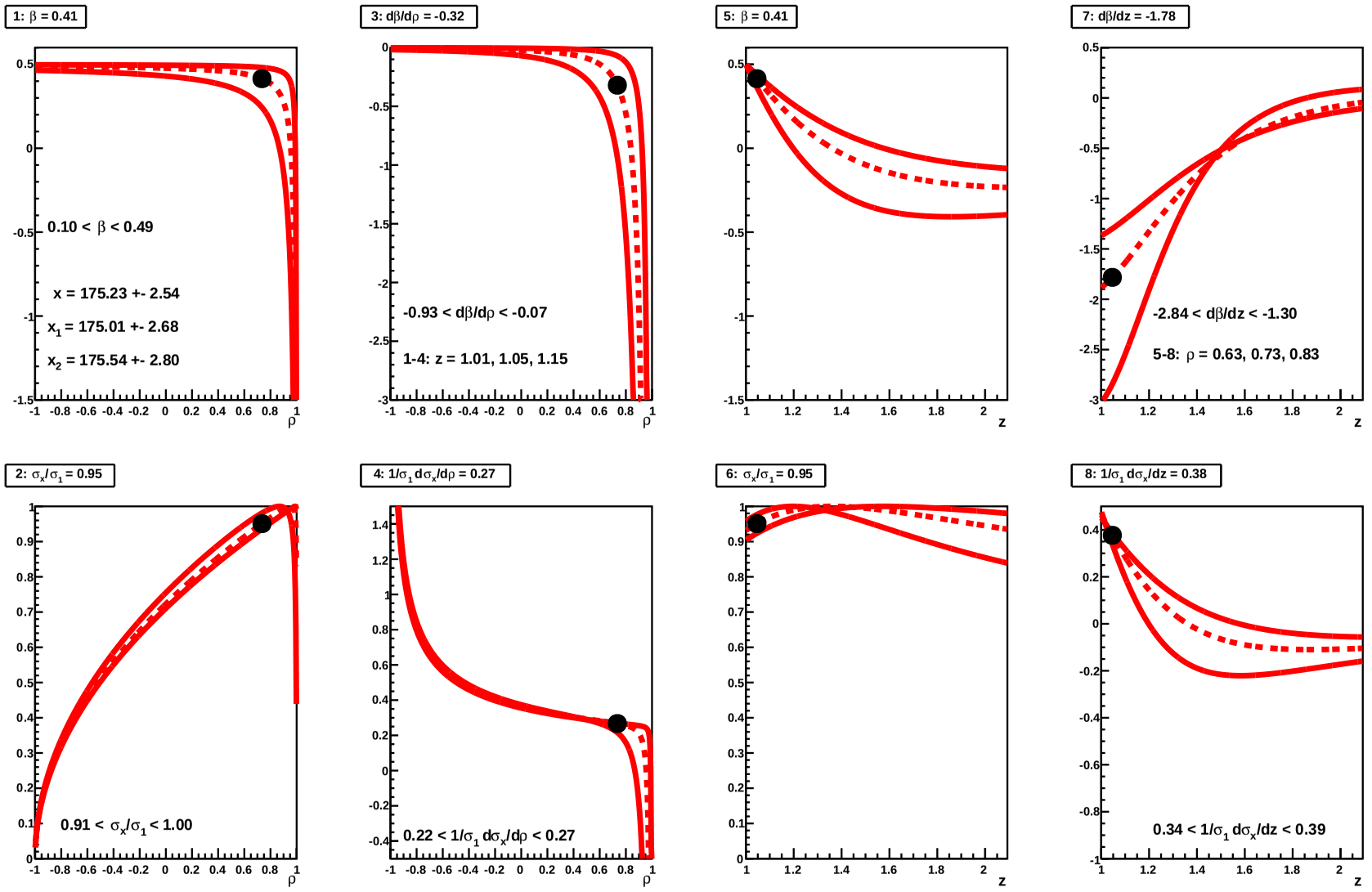


Figure 3: The sub-figures 1–8 correspond to Figures 1–2 for the special case investigated, i.e. the black point representing the actual values of ρ and z. The estimates x_1 and x_2, as well as the combined value x, together with their uncertainties, are listed. In addition quoted are the three values of ρ and z used for the two pairs of three curves. For the derivatives of β and σ_x/σ_1 with respect to ρ and z, for each sub-figure the range of values is given, obtained for the three curves shown, while keeping the respectve value of the other parameter. Finally, for β and σ_x/σ_1 their full range is quoted, obtained using all nine possible pairs of the ρ and z values.

B The release notes

The changes made to the software are listed in reverse order. Only the main points are given, for details please refer to the description of the interface in the main part of the text.

Changes from 1.8.0 to 1.9.0

1. Fix a bug in the calculation of $1/\sigma_1 d\sigma_x/d\rho$ in `GetPara(Ifl = 4)`. For the first factor of the equation, z^2 was used instead of z, see Eq. 4. Given this change, the display of the derivatives in `DrawSens()` has been changed, see Figure 3. In addition, this figure has been expanded by also showing the functional dependence of β and σ_x/σ_1 on z for various values of ρ, and by displaying the ranges of several parameters. Some other print out within the sub-figures has been removeded.
   This bug did not affect any combination, but all print out related to `GetPara(Ifl = 4)`.
2. Change a few examples.

Changes from 1.7.0 to 1.8.0

1. Complete change of the handling of scaled uncertainties. Remove a bug in `SetRhoFacUnc(...)` that occurred in case of InActive uncertainties.
2. Add `SolveScaRho()` to perform automated scans of different correlation assumptions for groups of estimates (defined in the constructor), and the corresponding `PrintScaRho()` to report the findings.
3. To serve this, add new functionality to the constructor, and to `SetRhoFacUnc(...)` and `SetRhoValUnc(...)`.
4. Add functions to Get the Num-ber of Sca-le Fac-tor groups `GetNumScaFac()`, and the Num-ber of Sca-le factor Rho values `GetNumScaRho()`.
5. Add getters, `GetScaVal(...)` and `GetScaUnc(...)` to return the differences in Val-ues and Unc-ertainties obtained with `SolveScaRho()` to the user.
6. Expand the information listed by `PrintAccImp()`. Remove a bug in `PrintAccImp()` that occurred for the improvement reported by adding the first estimate in case of InActive estimates.
7. Add `InspectResult()` to allow a closer look at the input in case of instable matrix inversions.
8. Fix a bug in the quoted pull values in `LatexResult(...)` that could occurr in case of InActive estimates.
9. Move some print statements from `Solve()` to `PrintStatus()`.
10. Change a few examples.

Changes from 1.6.0 to 1.7.0

1. Adapt `Makefile` for 32bit ROOT installations on 64bit machines.
2. Add one option to `SolveMaxVar()`.
3. Add various options to `SolveAccImp()`, keep the old function to run the new default option.
4. Add a function to Get the most Pre-cise Est-imate of a given observable `GetPreEst()`.
5. Change some print out in `DrawSens()`.

6. Changes some print statements is `PrintStatus()` and `PrintAccImp()`.
7. Fix a typo in Peeles, rename `B_Peeles` to `B_Peelles`.
8. Change a few examples.

Changes from v1.5 to 1.6.0

1. From this version onwards the software is hosted at `http://blue.hepforge.org/`. I also converted to the suggested numbering of the software versions.
2. Change from using ACLiC for the class to using compiled code. Convert to a consistent usage of const variables in member function calls. Add the corresponding `Makefile` and change the `B_name.inp` files accordingly.
3. Related to the consistent const variable usage in function calls the following protections against out of bound values have been altered. The protection against not allowed values of `Dx` in `SolveAccImp()` and `SolveRelUnc()`; the protection against not allowed values of `rho` in `GetPara()`; and the protection against x_2 being the more precise result in `DrawSens()`.
4. Remove all calls to `TString::Itoa()` which is only available from ROOT 5.34 onwards.
5. Rename the setters for `RelativeUnc` to `RelUnc` to match the name of the corresponding solver `SolveRelUnc()`.
6. Change some write statements in `LatexResult()`.
7. Fix a bug in `DrawSens()`. For the data point of the derivative of σ_x/σ_1 with respect to ρ the value itself was shown rather than its absolute value. Change some write statements. Change the minimum z to 1.01.
8. Optimise the calculation in the getters for arrays of Double_t values.
9. Include a print out of the various parameters of hypothetical pairwise combinations `PrintParams()`. Add the corresponding `GetParams()` functions.
10. Include a print out of the χ^2 and $P(\chi^2)$ for the combination `PrintChiPro()`.
11. Include some changes to the print out in `PrintMaxVar()`.
12. Update a number of example routines.
13. Adapt BlueAll as to only use `B_....inp` to protect files created via `ForttoBlue.inp` and changed by the user from being overwritten.
14. Add CompOne and CompAll to compile `B_....cxx` user functions to check for compiler errors.
15. Remove `SolveIterative()` as announced before.

Changes from v1.4 to v1.5

1. Expand `SolveMaxVar()` to also properly work in the case of InActive estimates and/or uncertainties. Fix a bug that for `IFuRho = 1` resulted in $f_k = -1$ for the situation where the maximum variance occurs for the initial situation, i.e. for $f_k = 1$. Finally, the algorithm was protected against numerical instabilities for `IFuRho = 2`.
2. Fix a bug in `FixInp()` when disabling estimates while using Relative Uncertainties.
3. Add the possibility to assign names to estimates (XXX=Est), uncertainties (Unc) and observables (Obs). Add three functions each to fill: `FillNamXXX()`, retrieve: `GetNamXXX()`, and print: `PrintNamXXX()` those.
4. Update a number of print, as well as the LaTeX and display routines to make use of these

names. Expand the content of the produced LaTeX file.
5. Add possible Format statements to `LatexResult()` and `DisplayResult()`.
6. Expand the constructor to print the software version and the date to the log file.
7. Include a utility called `ForttoBlue()` that works outside of the class. This utility converts input files for the Grunewald `Fort`-an software to functions to be used within the `Blue` class.
8. Update a number of example routines. Remove a return statement that had been inserted for test purposes into `B_arXiv_1307_4003.cxx`. This prevented a number of results from being printed.
9. Update the description of `DisplayResult()`.
10. Fix the description of the meaning of `IFuRho` for `SolveMaxVar()`.

Changes from v1.3 to v1.4

1. Add functions to retrieve the list of active estimates `GetIndEst(...)`, uncertainties `GetIndUnc(...)` and observables `GetIndObs(...)` into user arrays.
2. Add a function to print the contribution of an uncertainty source k to the covariance matrix `PrintCov(k)`.
3. Add four new methods to solve: according to importance `SolveAccImp(Dx)`, with the calculation of information weights `SolveInfWei()`, by only using estimates that have positive BLUE weights `SolvePosWei()`, or by maximising the variance of the result while changing the assumptions of the correlations of estimates for the various uncertainty sources `SolveMaxVar(IFuRho)`.
4. Add three new functions to print the findings of the respective solver: `PrintAccImp()`, `PrintInfWei()` and `PrintMaxVar()`.
5. Add two new functions to ease publishing the results: `LatexResult()` and `DisplayResult()` that create a tex file with a table of estimates and observables, or a figure.
6. Rename the BLUE method with relative uncertainties `SolveIterative()` to `SolveRelUnc()`.
7. Optimise `SolveRelUnc()`, protect against non filled functions for individual estimates when using `SetRelativeUnc(i, k, ActCof)`.
8. Add a new examples: B_arXiv_1307_4003.
9. Change a few examples implementing the new functionalities.
10. Fix bug in print out of stat + syst uncertainty breakdown in `PrintResult()` for the case of more than one observable.

Changes from v1.2 to v1.3

1. Add the possibility to fill percentage uncertainties to `FillEst()`.
2. Add the possibility to have fixed vales for the correlations for a given uncertainty, see `SetRhoValUnc()`.
3. Add the possibility to have relative and absolute uncertainties for the iterative BLUE method, see `SetRelativeUnc()`.
4. Add the iterative BLUE method `SolveIterative()`.
5. Add some print out to `PrintResult()`.
6. Add two print functions `PrintStatus()` and `PrintCofRelativeUnc()`.

7. Add three new examples: B_ATLAS_CONF_2013_098, B_ATLAS_CONF_2013_102 and B_PRD41_982.
8. Expand some examples.
9. Fix a bug in `PrintCor(k)` and `PrintEst(i)`. The correct entry was printed, but for the case of InActive uncertainties or estimates wrong error messages were produced.
10. Optimise `FixInp()`, protect against zero total uncertainties for estimates in the case of InActive uncertainties, and change some print out.

Changes from v1.1 to v1.2

1. Add a constructor for a single observable.
2. Add the possibility to scale the correlation for individual uncertainty sources.
3. Add the calculation of the χ^2 of the result.
4. Add the calculation and print function for the pull of the estimates.
5. Add the calculation of the compatibility of the estimates and observables.
6. Add the print function for the lists of estimates, uncertainties and observables.
7. Facilitate the retrieval of the total uncertainties of the observables.
8. Protect some private functions against wrong entries.
9. Improve some print functions and print statements.
10. Expand some examples.

Changes from v1.0 to v1.1

1. Change a number of print statements.
2. Add the possibility to set the level of details for the print out.
3. Add checks of the correlation matrices to FillCor().
   1) Check full matrices for symmetry.
   2) Check all matrices for range $(-1 < \rho < 1)$.
   3) Set all diagonal elements to one without notice.
4. Add B_ATLAS_CONF_2013_033.
5. Add B_ATLAS_CONF_2012_134.
6. Fix typo in Cor11 of B_arXiv_1107_5255.cxx.

References

[1] L. Lyons and D. Gibaut and P. Clifford, How to combine correlated estimates of a single physical quantity, Nucl. Instr. and Meth. A270 (1988) 110.

[2] A. Valassi, Combining correlated measurements of several different quantities, Nucl. Instr. and Meth. A500 (2003) 391.

[3] M. Grunewald, private communication, unpublished software.

[4] R. Brun and F. Rademakers, ROOT - An Object Oriented Data Analysis Framework, Nucl. Instr. and Meth. A389 (1997) 81–86, Proceedings of AIHENP'96 Workshop, Lausanne, Sep. 1996.

[5] L. Lyons and A.J. Martin and D.H. Saxon, On the determination of the B lifetime by combining the results of different experiments, Phys. Rev. D41 (1990) 982.

[6] A. Valassi and R. Chierici, Information and treatment of unknown correlations in the combinination of measurements using the BLUE method (v3). arXiv:arXiv:1307.4003.

[7] The ATLAS Collaboration, G. Aad, et al., Measurement of the top quark mass with the template method in the top antitop $\rightarrow$ lepton + jets channel using atlas data, Eur. Phys. J. C72 (2012) 2046. arXiv:arXiv:1203.5755.

[8] The Tevatron Electroweak Working Group for the CDF and DØ Collaborations, Combination of CDF and DØ results on the mass of the top quark using up to 5.8 fb^{-1} of data (v3). arXiv:arXiv:1107.5255.

[9] The Tevatron Electroweak Working Group for the CDF and DØ Collaborations, Combination of CDF and DØ results on the mass of the top quark using up to 8.7 fb^{-1} at the tevatron (v2). arXiv:arxiv:1305.3929.

[10] The ATLAS and CMS Collaborations, Combination of ATLAS and CMS results on the mass of the top quark using up to 4.9 fb^{-1} of data, ATLAS-CONF-2012-095, CMS-PAS-TOP-12-001.
URL http://cdsweb.cern.ch/record/1460441

[11] The ATLAS and CMS Collaborations, Combination of ATLAS and CMS top-quark pair cross-section measurements using proton-proton collisions at $\sqrt{s} = 7$ TeV, ATLAS-CONF-2012-134, CMS-PAS-TOP-12-003.
URL http://cdsweb.cern.ch/record/1478422

[12] The ATLAS and CMS Collaborations, Combination of the ATLAS and CMS measurements of the W-boson polarization in top-quark decays, ATLAS-CONF-2013-033, CMS PAS TOP-12-025.
URL http://cdsweb.cern.ch/record/1527531

[13] The ATLAS and CMS Collaborations, Combination of single top-quark cross-section measurements in the t-channel at $\sqrt{s} = 8$ TeV with the ATLAS and CMS experiments, ATLAS-CONF-2013-098, CMS-PAS-TOP-12-002.
URL http://cdsweb.cern.ch/record/1601029

[14] The ATLAS and CMS Collaborations, Combination of ATLAS and CMS results on the mass of the top-quark mass using up to 4.9 fb^{-1} of $\sqrt{s} = 7$ TeV LHC data, ATLAS-CONF-2013-102, CMS-PAS-TOP-13-005.
URL http://cdsweb.cern.ch/record/1601811

[15] R. Nisius, BLUE: a ROOT class to combine a number of correlated estimates of one or more observables using the Best Linear Unbiased Estimate method.
URL http://blue.hepforge.org