**ATLAS PUB Note**

ATL-PHYS-PUB-2021-011

September 6, 2021

# Evaluating statistical uncertainties and correlations using the bootstrap method

The ATLAS Collaboration

The bootstrap method is a powerful technique to evaluate the statistical uncertainty of a measurement and correlations between bins. This method uses a set of replicas of the nominal dataset, derived by introducing Poisson perturbations corresponding to statistical fluctuations. Each replica is then analyzed in the same way as the nominal dataset to arrive at a set of replica measurements. The statistical uncertainty and correlations can then be extracted from these replica measurements. This note describes a version of the bootstrap method suitable for data analysis in high energy physics and provides an associated software implementation. Various applications are discussed, such as determining the statistical error on systematic uncertainties. A novel feature of the provided software is that the fluctuations that generate the bootstrap replicas are deterministic. This makes it is possible to evaluate statistical correlations between measurements that are using fully or partially overlapping input data, even if the associated analyses are performed by different teams, or years apart.

[06-09-2021] Updated link to published BootstrapGenerator software to point to permanent Zenodo DOI.

# Contents



# 1 Introduction

Rigorous evaluation of statistical uncertainties in particle physics measurements and searches can be a challenging task. These kinds of analyses are often complex, and many involve non-trivial procedures such as propagation of statistical uncertainties across different subsets of the data or through detector-correction procedures. Complications in evaluating statistical uncertainties and correlations arise due to partial correlation of events in related measurements, the migration of events between bins, and in general the breakdown of the assumption of Gaussian behaviour in low-statistics regions. In such cases, the usual formulae used to calculate statistical uncertainties may become unreliable, or too complex to propagate (for example, through procedures like Iterative Bayesian Unfolding [1]).

This note presents a method based on the *Bootstrapping technique* [2, 3], which provides a reliable approach to propagating and evaluating statistical uncertainties, and which was first used by ATLAS in Ref. [4]. With this method, pseudo-experiments are generated in a coherent way such that it is possible to evaluate statistical correlation amongst measurements performed by different analyses. Tools have been developed within the ATLAS collaboration that extend the usual ROOT histogramming classes to formalise the use

of this method. Since these tools are of general use, the collaboration is making these tools public. The software is available on Zenodo at the link given by Ref. [5]. This is the companion note for these tools. It explains the method and describes its software implementation.

The note is structured as follows. First, a glossary of the nomenclature used in this note is provided in Section 2. Second, the method is explained qualitatively in Section 3, along with examples of potential applications of the method. Third, the details of how the method is implemented in the newly-released public code is explained in Section 4. Finally, a detailed example of how the code can be used are is given in Appendix A.

## 2 Nomenclature

For maximal clarity, and to minimise ambiguities or confusion, we begin the note with a glossary of the terms used in the rest of the note.

- **Ensemble**: a set of replicas for a given event or dataset. See **Replica**.
- **Measurement**: in this note, a *measurement* refers to the calculation of the value of an observable or parameter of interest, and the related uncertainties. For example, the event yield of a given bin of a differential cross-section measurement should be understood as a separate measurement in this context.
- **Nominal**: the *nominal event* is taken to be the raw, unmodified event from a data or simulation sample, before the application of the Bootstrap method. The set of nominal events taken together is the *nominal dataset*. The *nominal analysis* refers to the set of operations performed using the nominal events leading to the *nominal measurement* of some observable. One may speak of a *nominal histogram* to mean a histogram filled with nominal events.
- **Pseudo-dataset**: see **Replica**.
- **Pseudo-experiment**: see **Replica**.
- **Pseudo-random number**: A *pseudo-random number* is a number generated in a quasi-random way by a computer programme. Although these are colloquially referred to as "random", they in fact do repeat after a long period, and are reproducible given a particular seed. As such, they are properly called pseudo-random.
- **Replica**: a *replica* is a "copy" of the nominal dataset that contains its own unique statistical fluctuations. In this note, each *replica dataset* contains all the events of the nominal dataset, but each event has been assigned a unique and deterministically-generated weight sampled from a Poisson distribution with a mean of unity. A replica may also be referred to as a *pseudo-dataset* or *pseudo-experiment*. A set of $N$ replica datasets together is referred to as an *ensemble* of replica datasets, pseudo-datasets or pseudo-experiments. Colloquially, the word *toy* can be used to refer to these concepts, although it may refer to either the replica dataset or the measurement performed using the replica depending on the context. The *replica analysis* refers to the set of operations performed using a given ensemble of replica events leading to a *replica measurement* of some observable. One may speak of a *replica histogram* to mean a histogram filled with the corresponding replica events.

- **Seed**: the *seed* of a pseudo-random number generator is a value (or set of values) that is given to the generator to initialise it. Seeds are used to guarantee reproducibility. For a given initialising set of seeds, the sequence of pseudo-random numbers produced by the generator will always be the same. In this note, three seeds (corresponding to event number, run number and sample number for simulation) are used to uniquely but reproducibly initialise the pseudo-random number generator for each event, when generating the Poisson weights for the ensemble of replica events.
- **Toy**: see **Replica**.

# 3 Description of method

## 3.1 Description of method

For a given analysis, events from a statistical process are collected to form a dataset. This dataset is subsequently analyzed to perform a measurement. The bootstrap method can be thought of as considering alternative datasets otherwise collected under the same conditions, i.e. datasets that might have been collected in parallel universes. As defined in Section 2, such alternative datasets are referred to as replica datasets, and lead to corresponding replica measurements.

In the original bootstrap method [2, 3], each pseudo-dataset is created by sampling $N_{\text{nom}}$ events with replacement from the nominal dataset, where $N_{\text{nom}}$ is the number of events in the nominal dataset. This results in an ensemble of pseudo-datasets that all have the same size. This does not reflect the reality of collider experiments, where the size of the dataset is itself a Poisson variable. Hence, a common approach is to instead sample a unique weight for each event from a Poisson distribution with a mean of unity. A weight of 0, which occurs with 36.8% probability, effectively removes an event from the dataset, while a weight of e.g. 3 (6.1% probability) would correspond to the event being selected 3 times. The size of the resulting replica datasets will follow a Poisson distribution with mean $N_{\text{nom}}$, since sums of independent Poisson variables is a Poisson variable with mean equal to the sum of means ($\text{Pois}(N) \sim \sum_{i=1}^{N} \text{Pois}(1)$). This approach also has a computational advantage since one can create a whole ensemble of pseudo-datasets in a single loop through the nominal dataset by sampling a vector of Poisson weights for each event, corresponding to the different replicas.

The method employed by ATLAS (see Ref. [5] for link to associated software) assigns a unique and deterministic Poisson weight for each event and replica that is based on the event number and an index of the associated replica dataset. As a consequence, complex statistical correlations between two different analyses that use partially overlapping datasets can be fairly easily evaluated as the replica datasets will contain the same statistical fluctuation for the shared events. This applies to all features extracted from the dataset, division, subtraction, likelihood fits, and so on. If the measurement concerns multiple objects within an event (for example jet production rates), then all objects from a given event will correctly be treated as fully correlated.

For example, consider the ratio $\hat{r} = \hat{a}/\hat{b}$ between two measurements $\hat{a}$ and $\hat{b}$, performed with partially overlapping data. The nominal measurement $\hat{r}_0$ is performed using the nominal dataset, while a series of bootstrap measurements are performed using an ensemble of pseudo-experiments. A replica measurement $\hat{r}_i = \hat{a}_i/\hat{b}_i$ is performed using replica datasets $i$. In such replica measurements, the shared evens have the same fluctuations away from the nominal dataset, which will affect the measurements coherently: in the same direction if the measurements are positively correlated or in opposite direction if they are

anti-correlated. The distribution of measurements $\hat{r}_i$ can be treated as the probability distribution function of $\hat{r}$ and an uncertainty on this quantity can be derived from this distribution, e.g. from its standard deviation if appropriate.

Furthermore, using the ensemble of bootstrap replicas, it is possible to evaluate the statistical covariance between any two observables $a$ and $b$ according to

$$\mathrm{COV}(a,b) = \frac{1}{N_{\mathrm{rep}}} \sum_{i=0}^{N_{\mathrm{rep}}} (a_i - \bar{a})(b_i - \bar{b}), \tag{1}$$

where $N_{\mathrm{rep}}$ is the number of bootstrap replicas, and $a_i$ and $b_i$ are the measured values of the observables in bootstrap replica $i$. The variance of $a$, which is often taken as the square of its statistical uncertainty, is obtained from Eq. 1 by setting $b = a$, i.e. $\sigma_a = \sqrt{\mathrm{COV}(a,a)}$ and in the same way $\sigma_b = \sqrt{\mathrm{COV}(b,b)}$. Similarly, the correlation between the measurements is given by $\rho_{ab} = \mathrm{COV}(a,b)/(\sigma_a\,\sigma_b)$.

Further reading on this method can be found in Refs. [6] and [7]. The bootstrap implementation that is described in this note was first employed by the ATLAS collaboration in jet cross-section measurements at 7 TeV [4], 8 TeV [8] and 13 TeV [9], where the method was used to propagate statistical uncertainties through the particle-level unfolding procedure, and evaluate the statistical correlations between different measurements. It has also been used in jet calibration and performance studies, such as Refs. [10] and [11].

Specific examples that may help in understanding how the method operates and the situations where it can be useful are given in the follow sub-sections.

### 3.2 Example 1: Calculation of the statistical uncertainties for an unfolded measurement

Measurements of particle-level differential cross-sections have correlations between bins due to event migrations between the bins (at the unfolding step), but also due to the fact that it is possible to have several histogram entries per event. These cases are correctly handled by the replica histograms, if the unfolding is performed separately for each replica. In the Refs. [8] and [9], an ensemble of 10000 replicas was used to calculate a covariance matrix for the inclusive jet cross-section in each jet rapidity bin. The total statistical uncertainty was obtained from the covariance matrix, where bin-to-bin correlations were also encoded. The separate contributions from the data and from the MC statistics were obtained from the same procedure by fluctuating only either the data or the simulated events. Furthermore, an overall covariance matrix was constructed to describe the full statistical covariance among all analysis bins.

### 3.3 Example 2: Propagation of control or signal region uncertainties

In many analyses, one or several control regions are used to evaluate a background process in a so-called signal or search region. It is most often quite cumbersome to propagate the effect of statistical uncertainties in the control region(s) into the signal region. The bootstrap method is very useful in this situation as it stores replica histograms for each distribution. The full analysis is repeated for each replica, including determining the transfer factors (or likelihood fit). This leads to a slightly different constraint on the signal or search region for each replica, which in turn yields the propagated statistical uncertainty on the final measurement. Similarly, if a search has overlapping signal regions, these are not always statistically combined due to the difficulty in evaluating correlation from the shared events between regions. The

exclusion from the most sensitive region is often chosen in such cases, which weakens the final result since information from other regions is thrown away. With the bootstrap method one could use all the signal regions since the correlations from the overlapping events are correctly accounted for.

### 3.4 Example 3: Propagation of uncertainties in an unbinned fit

This method can also be used for an unbinned fit, where the fit is repeated for each replica. While the `BootstrapGenerator` class (see Section 4.1) can be used to deterministically seed the pseudo-random generation of numbers, we do not delve deeper into this possibility as this note concentrates on the implementation of the method in histograms, which are binned by definition.

### 3.5 Example 4: Correlation between multiple measurements

Since each event has a unique seed determined by run/event/sample number, one can fill different distributions and get the same fluctuations in their replica histograms. This means that a given event will be associated with a particular set of fluctuations, regardless of which histogram(s) are filled with it. Therefore, different measurements, potentially made by different teams, which use a subset of the same events, would fluctuate their objects in the same way for those events. This allows the calculation of cross-correlations, or correlations between different measured spectra, either within the same analysis or different one. Indeed, if the replica histograms are preserved and published, a post-hoc assessment of the statistical cross-correlation of different measurements is possible. There are already examples of ATLAS measurements that have preserved their replica histograms in their HEPData entries, such as Refs. [12] and [13].

### 3.6 Example 5: Statistical uncertainty on systematics

When evealuating a systematic uncertainty, a histogram filled using a nominal calibration is often divided by a histogram filled using the same sample but an alternative calibration, resulting in a relative systematic uncertainty. Often, the corresponding bins in each histogram are filled with many of the same events, the correlation must be taken into account when evaluating the statistical uncertainty on their ratio. By generating replicas for the nominal histogram and systematic histogram, and dividing the synchronized replicas, the proper statistical uncertainty—including correlations—can be found by simply calculating the standard deviation of replica measurements for each histogram bin. This information is particularly useful as an input when smoothing the systematic uncertainty shape, as was done for example in Refs. [10] and [11].

## 4 Implementation

The ROOT data analysis package is ubiquitous in the HEP community, so the Bootstrap method was implemented as an extension to the ROOT histogramming classes in C++. The ROOT classes have a Python interface called PyROOT. The additional Bootstrap classes are similarly integrated into PyROOT. The code for these additional Bootsrap objects is collected in Ref. [5], and the implementation is discussed below.

### 4.1 BootstrapGenerator

The BootstrapGenerator class is the one that sets the pseudo-random number seed and generates the pseudo-random numbers used to fill the replica histograms in the Bootstrap method. In order to be integrated into the ROOT framework, the BootstrapGenerator class inherits from the generic ROOT classes TNamed, like most other ROOT objects. It also inherits from TArrayI, in order to efficiently store arrays of integers.

The key BootstrapGenerator member variables are:

- fNReplica: this is an integer that stores the number of Bootstrap replica histograms to use in the Bootstrap method (usually of the order of 1000);
- fSeeds: an array of integers storing the seeds to use when generating pseudo-random numbers;
- fStoch: a pointer to a StochasticLib2 object, an external C++ pseudo-random number generator class;
- fArray: inherited from the TArrayI, an array of integers storing the pseudo-randomly generated numbers;

The BootstrapGenerator class has several constructors that set these member variables, populating them either with default values or those provided as arguments, and initialises the pseudo-random number generator. There are also dedicated setter and getter methods to manipulate the member variables, although the user should not need to do so in most cases.

The method that does the heavy lifting in the BootstrapGenerator class is the Generate method, which takes the event's run number and event number as unsigned integer inputs. If one is using the Bootstrap method on MC simulation, and additional integer may be provided that corresponds to the sample number of the simulated sample (since MC samples may use the same run and event numbers). The method collects this set of integers and uses them to set the seed of the pseudo-random number generator. In this way, each event of any data or MC sample is assigned a unique, reproducible seed. Then, an array of size equal to the number of replica histograms fNReplica, which houses the relevant pseudo-random numbers drawn from a Poisson distribution with rate parameter equal to 1.

The pseudo-random numbers are then unique for a given set of run, event and channel numbers. They are used to fluctuate the amount by which each replica histogram is filled, as described below.

### 4.2 TH1Bootstrap and derived classes

*In this section and hereafter we will focus on the TH1Bootstrap class, but TH2Bootstrap and TH3Bootstrap classes are also implemented, which behave exactly the same as their one-dimensional counterparts.*

The most common type of histogram used in ROOT is the TH1 class, which presents one-dimensional histograms. It has derived classes TH1F and TH1D that represent histograms where the $x$-axis values are stored as C++ float and double values respectively. These objects are referred to collectively as TH1* objects hereafter. The implementation of the Bootstrap method described here seeks to create versions of these classes that automatically apply the Bootstrap method, with very little change in usage

```
// Initialisation using a BootstrapGenerator
nrep = 1000; //number of replicas
auto gen = new BootstrapGenerator("Gen", "Gen", nrep);
auto reco_hist1 = new TH1DBootstrap("reco_hist1", "", 10, 0, 10, nrep, gen);
auto reco_hist2 = new TH1DBootstrap("reco_hist2", "", 10, 0, 10, nrep, gen);

// Initialisation without  a BootstrapGenerator
auto truth_hist = new TH1DBootstrap("truth_hist", "", 10, 0, 10, nrep)
```

Listing 1: An example of how `TH1Bootstrap` objects are initialised either with an explicit `BootstrapGenerator` object which can be shared between `TH1DBootstrap` instances or with an implicit internal `BootstrapGenerator`.

for the user compared to the `TH1*` classes. They are referred to as `TH1Bootstrap`, `TH1FBootstrap` and `TH1DBootstrap` (or `TH1*Bootstrap` collectively).

The `TH1Bootstrap` class is the base class from which `TH1FBootstrap` and `TH1DBootstrap` inherit. It forward-declares the methods of `TH1FBootstrap` and `TH1DBootstrap`, and has member variables for the number of replica histograms and the associated `BootstrapGenerator` object (see Section 4.1) that generates the pseudo-random Poisson numbers that affect the replica histograms.

### 4.3 `TH1Bootstrap` initialisation

The Bootstrap histogram classes are effectively wrappers around regular ROOT histogram objects. The "main" histogram, which takes the place of the `TH1*` object the user would be used to, is referred to as the as the "Nominal" histogram and is a member variable of the corresponding `TH1*Bootstrap` class (accessed by a dedicated `GetNominal()` method). In addition to the nominal histogram, the Bootstrap histogram classes have an array of other `TH1*` objects, which represent the replica histograms. These normally do not need to be individually manipulated by the user, but can be accessed by `GetReplica( int iReplica)` method. When a new `TH1*Bootstrap` object is created, the nominal and all replica histograms are all initialised at once with the binning specified by the user.

Each `TH1Bootstrap` object may either create its own `BootstrapGenerator` object on initialisation, or one may specify a particular instance of `BootstrapGenerator` when constructing a `TH1Bootstrap`. The latter option is preferred, since it's more efficient in terms of memory use, and simplifies the `Fill` method described below.

The typical constructor for a `TH1Bootstrap` object would therefore be: `TH1Bootstrap(const char *name, const char *title, int nreplica, BootstrapGenerator *boot = nullptr)`, where if the final `BootstrapGenerator` argument is left empty, a dedicated object will be created, and the user is responsible for passing the event, run and sample numbers to the object each time it is filled. Example pseudo-code that show how these objects are initialised can be found in Listing 1, with a more complete example in Appendix A.

### 4.4 `TH1Bootstrap` filling

In the simplest case, `TH1*Bootstrap` objects use the set of pseudo-random Poisson values already generated by their `BootstrapGenerator` object to fill the replica histograms. This means that the

```
// number of replica histograms to use, with common BootstrapGenerator
int nrep = 1000; // number of replicas
auto gen = new BootstrapGenerator("Gen", "Gen", nrep);

// Book bootstrap objects that use the above generator
auto reco_hist = new TH1DBootstrap("reco_hist", "", 10, 0, 10, nrep, gen);
auto truth_hist = new TH1DBootstrap("truth_hist", "", 10, 0, 10, nrep, gen);

for (int i = 1; i < nevent; i++) { // for each event...

  //(...)

  // The next line generate the random Poisson weights for the current event
  // nrep random weights are produced, one for each bootstrap replica
  gen->Generate(/*run number*/ 12345, /*event number*/ i, /*sample number if MC*/
    6789);

  // The Poisson weights can be accessed using GetWeight().
  int bsWeight5 = gen->GetWeight(5); // Poisson weight for BS replica 5

  // Fill the bootstrap objects, which access the pseudo-random
  // numbers from the associated BootstrapGenerator under the hood
  // where particle and trutParticle are some four-momenta for reco-level
  // and particle-level particles...
  reco_hist->Fill(particle.Pt(), 1.0);
  truth_hist->Fill(truthParticle.Pt(), 1.0);

  //(...)
}
```

Listing 2: An example of how `TH1Bootstrap` objects are filled using a common `BootstrapGenerator` object.

associated `Generate()` method should be called for each event (setting the unique seed with event, run and sample numbers), before filling any Bootstrap histograms, via the usual `Fill(double x,double w)` method. If the `TH1Bootstrap` was not associated to a particular `BootstrapGenerator` upon initialisation, one should also specify the run, event and sample numbers when filling so that the internal `BootstrapGenerator` may set the correct seed: `Fill(double x, double w, unsigned int RunNumber, unsigned int EventNumber, unsigned int mcChannelNumber)`.

In either case, when filling a Bootstrap histogram, the set of pseudo-random Poisson integers is accessed from the `BootstrapGenerator`. The fill is executed separately for each replica histogram, but unlike the nominal that is filled once, the replica histograms are filled according to the pseudo-random integers from the array. This means the replica histograms may be filled with weight 0, 1 or more depending in the value of the random integer. Examples of pseudo-code showing how to fill a `TH1Bootstrap` can be found in Listings 2 and 3, with a more complete example in Appendix A.

### 4.5 `TH1Bootstrap` manipulation

The `TH1Bootstrap` classes also come with implementations of the usual methods for histogram manipulation that ROOT users would expect: `AddBinContent()`, `Add()`, `Multiply()`, `Divide()`, `Scale()`, `Rebin()`... are all implemented, and apply these operations to the nominal histogram, but also to each

```
// Althernatively,  don't specify a generator in which case one will be made
// behind the scenes, but you'll need to provide extra information when filling
int nrep = 1000; // number of replicas
auto other_hist = new TH1DBootstrap("other_hist", "other_hist", 10, 0., 10., nrep);

for (int i = 1; i < nevent; i++) { // for each event...

  //(...)

  other_hist->Fill(truthPart.Pt(), 1.0, /*run number*/ 12345, /*event number*/ i, /*
    sample number if MC*/ 6789);
}
```

Listing 3: An example of how `TH1Bootstrap` objects are filled using an implicit internal `BootstrapGenerator` object.

of the replica histograms. This means that once a set of `TH1Bootstrap` objects have been initialised, the user can continue the analysis, filling, combining, taking ratios, and so on, without worrying about the replica histograms, which are handled under the hood. Pseudo-code that shows the manipulation of `TH1Bootstrap` can be found in Listing 4, with a more complete example in Appendix A.

### 4.6 `TH1Bootstrap` statistical error and correlations

Once the filling and manipulation is done, one can call the helper functions `GetBootstrapMean()`, `GetBootstrapRMS()` to calculate the mean and error in each bin from the replica histograms, and `GetBootstrapCorel()`, `GetCovarianceMatrix()`, `GetCorrelationMatrix()`... to assess statistical correlations between bins. The covariance between two bin event yields $b_i$ and $b_j$ of bins $i$ and $j$, respectively, is given according to Eq. 1 using the formula below:

$$C_{ij} = \frac{1}{N_{\text{rep}}} \sum_{k=1}^{N_{\text{rep}}} (b_{ik} - \bar{b}_i)(b_{jk} - \bar{b}_j), \tag{2}$$

where $k$ is an index corresponding to the replica, $N_{\text{rep}}$ is the number of replicas, $\bar{b}_i$ and $\bar{b}_j$ are the average event yields of $b_i$ and $b_j$, respectively, across all replicas.

These simple methods complete the bootstrap method, allowing the user to calculate statistical errors and correlations correctly, without worrying about keeping track of the replica histograms themselves. To combine results with other measurements, one need only keep bootstrap replica histograms and call the statistical errors and correlation methods after combining the results. This is achieved with the `Append()` function, which appends two 1- or 2-D `THBootstrap` instances. For the 1-D case, this appends the bins from one histogram to another for both the nominal instance and the replicas. In the 2-D case, the histograms are first collapsed to one dimension, by appending the x-bins from the second y-bin to those from the first y-bin, and so on. The resulting `TH1Bootstrap` can then provides correlations between the two spectra that were appended. Example pseudo-code showing how `TH1Bootstrap` objects can be appended, and how the correlation matrix is extracted, can be found in Listing 5, with a more complete example in Appendix A.

```
// Book bootstrap objects
int nrep = 1000; //number of replicas
float lumi = 36000;
auto gen = new BootstrapGenerator("Gen", "Gen", nrep);
auto reco_hist = new TH1DBootstrap("reco_hist", "reco_hist", 10, 0., 10., nrep, gen)
    ;
auto background_hist = new TH1DBootstrap("background_hist", "background_hist", 10,
    0., 10., nrep, gen);
auto truth_hist = new TH1DBootstrap("truth_hist", "truth_hist", 10, 0., 10., nrep,
    gen);
auto data_hist = new TH1DBootstrap("data_hist", "data_hist", 10, 0., 10., nrep);


// Fill them
for (int i = 1; i < nevent; i++) { // for each event...
  gen->Generate(runNumber, eventNumber, sampleNumber);
  //(... fill Boostraps ...)
}

// Manipulate them as normal TH1 objects  these operations will be applied also
// to each replica histogram in addition to the nominal histogram.

// Bin-by-bin corrections factors. Copy truth, divide bin-by-bin by reco
auto c_factors = (TH1DBootstrap*)truth_hist->Clone();
c_factors->Divide(reco_hist);


auto unfolded_hist = data_hist->Clone();
unfolded_hist->Add(background_hist, -1);
unfolded_hist->Multiply(c_factors);
unfolded_hist->Rebin(2);
unfolded_hist->Scale(1./lumi);

// The next line calculates the bootstrap stat uncertainties and
// updates the bin errors of the nominal histogram accordingly
unfolded_hist->SetErrBootstrapRMS();

// The bootstrap errors can now be accessed like this:
double bsError1 = unfold_hist->GetNominal()->GetBinError(1);
```

Listing 4: An example of how `TH1Bootstrap` objects are manipulated using the usual `TH1` operations like `Add` `Scale` and so on.

```
1
2  // Define two bootstrap histograms for two different observables A and B
3  int Nrep = 1000;
4  TH1DBootstrap spectrumA(100, 0, 100, Nrep);
5  TH1DBootstrap spectrumB(100, 0, 100, Nrep);
6
7  // Fill both spectra with different variables.
8  // To calculate correlations between the different variables,
9  // we put the measurements "side-by-side", append to 200 bins
10 spectrumA.Append(spectrumB);
11
12 // We can now access the full correlation matrix (200 x 200)
13 auto corrMatrix = spectrumA.GetCorrelationMatrix();
14
15 // Statistical correlation between first and second bin of A
16 double corr_A1_A2 = corrMatrix->GetBinContent(1,2);
17 // Statistical cross-correlation between first bin of A and first of B:
18 double corr_A1_B1 = corrMatrix->GetBinContent(1,101);
```

Listing 5: An example of how to extract a correlation matrix from a `TH1Bootstrap` object that includes cross-correlations between two different observables.

## 5 How to cite

If you use the `TH*DBootstrap` classes described above for your work, please reference this note and the following publications where the code was first used, using the `.bib` entries in Listing 6.

## 6 Conclusion

This note was written to accompany the public release of the ATLAS `BootstrapGenerator` code [5], and to explain the main features of the boostrap method and how it is implemented in software. The method allows for a rigorous and reproducible evaluation of the statistical covariance across several related measurements. The software package, which is made public by ATLAS extends the usual ROOT histogram classes to implement this method, with minimal change to the user.

```
  @Article{STDM-2012-03,
    author         = "{ATLAS Collaboration}",
    title          = "{Measurement of dijet cross sections in \(pp\) collisions at
    \(7\,\text{TeV}\) centre-of-mass energy using the ATLAS detector}",
    journal        = "JHEP",
    volume         = "05",
    year           = "2014",
    pages          = "059",
    doi            = "10.1007/JHEP05(2014)059",
    reportNumber   = "CERN-PH-EP-2013-192",
    eprint         = "1312.3524",
    archivePrefix  = "arXiv",
    primaryClass   = "hep-ex",
  }
@Article{STDM-2015-01,
  author         = "{ATLAS Collaboration}",
  title          = "{Measurement of the inclusive jet cross-sections in proton--
    proton collisions at \(\sqrt{s} = 8\,\text{TeV}\) with the ATLAS detector}",
  journal        = "JHEP",
  volume         = "09",
  year           = "2017",
  pages          = "020",
  doi            = "10.1007/JHEP09(2017)020",
  reportNumber   = "CERN-EP-2017-043",
  eprint         = "1706.03192",
  archivePrefix  = "arXiv",
  primaryClass   = "hep-ex",
}
@Article{STDM-2016-03,
  author         = "{ATLAS Collaboration}",
  title          = "{Measurement of inclusive jet and dijet cross-sections in proton
    --proton collisions at \(\sqrt{s} = 13\,\text{TeV}\) with the ATLAS detector}",
  journal        = "JHEP",
  volume         = "05",
  year           = "2018",
  pages          = "195",
  doi            = "10.1007/JHEP05(2018)195",
  reportNumber   = "CERN-EP-2017-157",
  eprint         = "1711.02692",
  archivePrefix  = "arXiv",
  primaryClass   = "hep-ex",
}
```

Listing 6: Citations to include if using the classes described in this note.

# Appendix

## A Example

A detailed an annotated example on how to use the bootsrap classes for bin-by-bin unfolding is given in Listing 7. The code can be compiled in C++:

```

#ifdef __CLING__
R__LOAD_LIBRARY(libBootstrapGenerator.so)
#endif

//import Boosrap classes
#include "BootstrapGenerator/BootstrapGenerator.h"
#include "BootstrapGenerator/TH1DBootstrap.h"
#include "BootstrapGenerator/TH2DBootstrap.h"

#include <cstdio>

#include "TH2D.h"
#include "TRandom3.h"
#include "TVector3.h"
#include "TLorentzVector.h"
#include "TFile.h"
#include "TStopwatch.h"
#include "TSVDUnfold.h"
#include "TMath.h"
#include "TCanvas.h"
#include "TLegend.h"

void UnfoldingExample()
{
  // use 1000 replicas, and 1 million events
  int nrep = 1000, nevent = 100000;

  // Book a generator:
  auto gen  = new BootstrapGenerator("Gen", "Gen", nrep);

  // Transfer matrix
  auto transfer_hist2d = new TH2DBootstrap("transfer_hist2d", "transfer_hist2d", 10,
     0., 10., 10, 0., 10., nrep, gen);

  // Spectra
  auto truth_hist = new TH1DBootstrap("truth_hist", "truth_hist", 10, 0., 10., nrep,
     gen); // Truth spectra
  auto reco_hist = new TH1DBootstrap("reco_hist", "reco_hist", 10, 0., 10., nrep,
    gen); // Reco spectra
  auto data_hist = new TH1DBootstrap("data_hist", "data_hist", 10, 0., 10., nrep,
    gen); // Data spectra

  // "Systematics"
  auto data_hist_up = new TH1DBootstrap("data_hist_up", "data_hist_up", 10, 0., 10.,
     nrep, gen); // Systematic shifts of data Up and
  auto data_hist_dn = new TH1DBootstrap("data_hist_dn", "data_hist_dn", 10, 0., 10.,
     nrep, gen); // Down
```

```
43
44  // Generate particles with E between 0 and 10 at random angles:
45  TRandom3 *rnd = new TRandom3();
46
47  for (int i = 1; i < nevent; i++) {
48    if (i % 10000 == 0) {
49      printf("Processed %d events\n",i);
50    }
51
52    // Truth particle. Here we simply sample from uniform distribution
53    double pT = rnd->Rndm(1.0) * 10.;
54    double phi = TMath::TwoPi() * rnd->Rndm();
55    double eta = rnd->Gaus(1.0);
56
57    // Detector "smeared" quantities
58    double pT_reco = pT * rnd->Gaus(0.98, 0.1); // assume bias -2%, smear 10%
59    double phi_reco = phi * rnd->Gaus(1., 0.01); // assume no bias, smear 1%
60    double th_reco = eta * rnd->Gaus(1., 0.01); // assume smear 1%
61
62    // fill Lorentz vectors for truth and reco
63    TLorentzVector p_truth, p_reco;
64    p_truth.SetPtEtaPhiM(pT, eta, phi, 0); // assume m=0
65    p_reco.SetPtEtaPhiM(pT_reco, eta_reco, phi_reco, 0); // assume m=0
66
67
68    // "Systematics" shift of 1%
69    TLorentzVector p_reco_up, p_reco_dn;
70    p_reco_up.SetPtEtaPhiM(pT_reco, eta_reco, phi_reco, 0); p_reco_up * 1.01;
71    p_reco_dn.SetPtEtaPhiM(pT_reco, eta_reco, phi_reco, 0); p_reco_up * 0.99;
72
73    // Update weights:
74    gen->Generate(219305, i);
75
76    truth_hist->Fill(p_truth.Pt(), 1.0);
77    if (rnd->Rndm() > 0.05) { // 5% inefficiency in detector reconstruction
78      transfer_hist2d->Fill(p_reco.Pt(), p_truth.Pt());
79      reco_hist->Fill(p_reco.Pt(), 1.0);
80      if (rnd->Rndm() < 0.2) { // Only 20% of the events make the "data" spectrum
81        data_hist->Fill(p_reco.Pt(), 1.0);
82        data_hist_up->Fill(p_reco_up.Pt(), 1.0);
83        data_hist_dn->Fill(p_reco_dn.Pt(), 1.0);
84      }
85    }
86
87  }
88
89  // Unfolding result
90  TH1D  *unfolded_hist = nullptr;
91  TH1D  *unfolded_hist_up = nullptr;
92  TH1D  *unfolded_hist_dn = nullptr;
93
94  // Unfolding replicas results
95  TH1D **unfolded_replicas = new TH1D*[nrep];
96  TH1D **unfolded_replicas_up = new TH1D*[nrep];
97  TH1D **unfolded_replicas_dn = new TH1D*[nrep];
98
99  // Nominal unfolding
```

```
auto tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist->GetNominal(),
    (TH1D*)reco_hist->GetNominal(),
    (TH1D*)truth_hist->GetNominal(),
    (TH2D*)transfer_hist2d->GetNominal());
unfolded_hist = tsvd_unfold_object->Unfold(6.0); // use kreg = 6.0
delete tsvd_unfold_object;

tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist_up->GetNominal(),
    (TH1D*)reco_hist->GetNominal(),
    (TH1D*)truth_hist->GetNominal(),
    (TH2D*)transfer_hist2d->GetNominal());
unfolded_hist_up = tsvd_unfold_object->Unfold(6.0); // use kreg = 6.0
unfolded_hist_up->Add(unfolded_hist, -1); // Make bootstrap of the difference to
  get error on systematic
delete tsvd_unfold_object;

tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist_dn->GetNominal(),
    (TH1D*)reco_hist->GetNominal(),
    (TH1D*)truth_hist->GetNominal(),
    (TH2D*)transfer_hist2d->GetNominal());
unfolded_hist_dn = tsvd_unfold_object->Unfold(6.0); // use kreg = 6.0
unfolded_hist_dn->Add(unfolded_hist, -1); // Make bootstrap of the difference to
  get error on systematic
delete tsvd_unfold_object;

// Replica unfolding
for (int i = 0; i < nrep; ++i) {
  tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist->GetReplica(i),
      (TH1D*)reco_hist->GetReplica(i), // Use replicas of transfer matrix so that
      (TH1D*)truth_hist->GetReplica(i), // MC uncertainty is included.
      (TH2D*)transfer_hist2d->GetReplica(i));
  unfolded_replicas[i] = tsvd_unfold_object->Unfold(6.0);
  delete tsvd_unfold_object;

  tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist_up->GetReplica(i),
      (TH1D*)reco_hist->GetNominal(),
      (TH1D*)truth_hist->GetNominal(),
      (TH2D*)transfer_hist2d->GetNominal());
  unfolded_replicas_up[i] = tsvd_unfold_object->Unfold(6.0);
  unfolded_replicas_up[i]->Add(unfolded_replicas[i], -1); // Make bootstrap of the
   difference to get error on systematic
  delete tsvd_unfold_object;

  tsvd_unfold_object = new TSVDUnfold((TH1D*)data_hist_dn->GetReplica(i),
      (TH1D*)reco_hist->GetNominal(),
      (TH1D*)truth_hist->GetNominal(),
      (TH2D*)transfer_hist2d->GetNominal());
  unfolded_replicas_dn[i] = tsvd_unfold_object->Unfold(6.0);
  unfolded_replicas_dn[i]->Add(unfolded_replicas[i], -1); // Make bootstrap of the
   difference to get error on systematic
  delete tsvd_unfold_object;
}

// Collect unfolded result with synchornized replicas
auto unfolded_bootstrap  = new TH1DBootstrap("result", "result", unfolded_hist,
  unfolded_replicas, nrep);
```

```
151 unfolded_bootstrap->SetErrBootstrapRMS(); // Only containts statistical error from
      data in this example
152
153 // Evaluate error on systematic shift error
154 auto unfolded_bootstrap_up = new TH1DBootstrap("relSysUp", "sysUp",
      unfolded_hist_up, unfolded_replicas_up, nrep);
155 auto unfolded_bootstrap_dn = new TH1DBootstrap("relSysDown", "sysDown",
      unfolded_hist_dn, unfolded_replicas_dn, nrep);
156
157 unfolded_bootstrap_up->SetErrBootstrapRMS();
158 unfolded_bootstrap_up->Divide(unfolded_bootstrap); // make it relative
159
160 unfolded_bootstrap_dn->SetErrBootstrapRMS();
161 unfolded_bootstrap_dn->Divide(unfolded_bootstrap);
162
163 // Save unfolded result, with synchronized replicas
164 TFile file("unfold.root", "RECREATE");
165 unfolded_bootstrap->Write();
166 ((TH1D*)unfolded_bootstrap_up->GetNominal())->Write();
167 ((TH1D*)unfolded_bootstrap_dn->GetNominal())->Write();
168 file.Close();
169
170 // Make a plot showing things work
171 TCanvas c("c", "c", 600, 600);
172
173 truth_hist->Scale(1.0/5.0);
174 truth_hist->GetNominal()->SetLineColor(kRed);
175 truth_hist->GetNominal()->Draw("hist");
176
177 unfolded_bootstrap->GetNominal()->SetLineColor(kBlue);
178 unfolded_bootstrap->GetNominal()->Draw("hist same");
179
180 data_hist->GetNominal()->Draw("hist same");
181
182 TLegend legend(0.5, 0.75, 0.9, 0.90);
183 legend.AddEntry(truth_hist->GetNominal(), "True spectra", "l");
184 legend.AddEntry(data_hist->GetNominal(), "Reco spectra", "l");
185 legend.AddEntry(unfolded_bootstrap->GetNominal(), "Unfolded spectra", "l");
186 legend.Draw();
187
188 c.SaveAs("unfolding.png");
189
190 TH2D axis("axis", "axis", 1, 0, 10, 1, -0.10, 0.10);
191 axis.Draw("axis");
192
193 unfolded_bootstrap_up->GetNominal()->SetMarkerStyle(1);
194 unfolded_bootstrap_up->GetNominal()->SetMarkerColor(kRed);
195 unfolded_bootstrap_up->GetNominal()->SetLineColor(kRed);
196 unfolded_bootstrap_up->GetNominal()->Draw("pe same");
197
198 unfolded_bootstrap_dn->GetNominal()->SetMarkerStyle(1);
199 unfolded_bootstrap_dn->GetNominal()->SetMarkerColor(kBlue);
200 unfolded_bootstrap_dn->GetNominal()->SetLineColor(kBlue);
201 unfolded_bootstrap_dn->GetNominal()->Draw("pe same");
202
203 TLegend legend_systematics(0.2, 0.80, 0.6, 0.90);
```

```
  legend_systematics.AddEntry(unfolded_bootstrap_up->GetNominal(), "Positive
    uncertainty shift", "lpe");
  legend_systematics.AddEntry(unfolded_bootstrap_dn->GetNominal(), "Negative
    uncertainty shfit", "lpe");
  legend_systematics.Draw();

  c.SaveAs("syserr.png");

  delete unfolded_bootstrap;
}
```

Listing 7: An complete example of how to use `TH1Bootstrap` objects for a a simple bin-by-bin unfolding in C++.

## References


[1] G. D'Agostini, *A multidimensional unfolding method based on Bayes' theorem*, Nucl. Instrum. Meth. A **362** (1995) 487, ISSN: 0168-9002 (cit. on p. 2).

[2] B. Efron, *Bootstrap Methods: Another Look at the Jackknife*, Annals Statist. **7** (1979) 1 (cit. on pp. 2, 4).

[3] B. Efron and R. Tibshirani, *An Introduction to the Bootstrap*, Chapman & Hall, 1994 (cit. on pp. 2, 4).

[4] ATLAS Collaboration, *Measurement of dijet cross sections in pp collisions at 7 TeV centre-of-mass energy using the ATLAS detector*, JHEP **05** (2014) 059, arXiv: 1312.3524 [hep-ex] (cit. on pp. 2, 5).

[5] ATLAS Collaboration, *BootstrapGenerator*, version 1.11.2, Zenodo, 2021, URL: https://doi.org/10.5281/zenodo.5361038 (cit. on pp. 3, 4, 6, 12).

[6] G. Bohm and G. Zech, *Introduction to statistics and measurement analysis for physicists*, 2005, ISBN: 978-3-945931-13-4 (cit. on p. 5).

[7] G. J. Babu, P. K. Pathak, and C. R. Rao, *Second-Order Correctness of the Poisson Bootstrap*, The Annals of Statistics **27** (1999) 1666, ISSN: 00905364, URL: http://www.jstor.org/stable/2674086 (cit. on p. 5).

[8] ATLAS Collaboration, *Measurement of the inclusive jet cross-sections in proton–proton collisions at $\sqrt{s}$ = 8 TeV with the ATLAS detector*, JHEP **09** (2017) 020, arXiv: 1706.03192 [hep-ex] (cit. on p. 5).

[9] ATLAS Collaboration, *Measurement of inclusive jet and dijet cross-sections in proton–proton collisions at $\sqrt{s}$ = 13 TeV with the ATLAS detector*, JHEP **05** (2018) 195, arXiv: 1711.02692 [hep-ex] (cit. on p. 5).

[10] ATLAS Collaboration, *Determination of jet calibration and energy resolution in proton-proton collisions at $\sqrt{s}$ = 8 TeV using the ATLAS detector*, (2019), arXiv: 1910.04482 [hep-ex] (cit. on pp. 5, 6).

[11] ATLAS Collaboration, *Jet energy scale measurements and their systematic uncertainties in proton-proton collisions at $\sqrt{s}$ = 13 TeV with the ATLAS detector*, Phys. Rev. D **96** (2017) 072002, arXiv: 1703.09665 [hep-ex] (cit. on pp. 5, 6).

[12] ATLAS Collaboration, *HEPData Record: Measurement of the inclusive jet cross-sections in proton-proton collisions at 8 TeV with the ATLAS detector*, 2017, URL: https://www.hepdata.net/record/ins1604271?version=1 (cit. on p. 6).

[13] ATLAS Collaboration, *HEPData Record: Measurement of inclusive jet and dijet cross-sections in proton-proton collisions at 13 TeV with the ATLAS detector*, 2017, URL: https://www.hepdata.net/record/ins1634970?version=1 (cit. on p. 6).