September 25, 2019

1 Usage instructions

The following instructions go over usage instructions for the auger-analysis program. They also explain individual parts of the source code and give instructions for adding new options and customizing the software.

1.1 The graphical interface

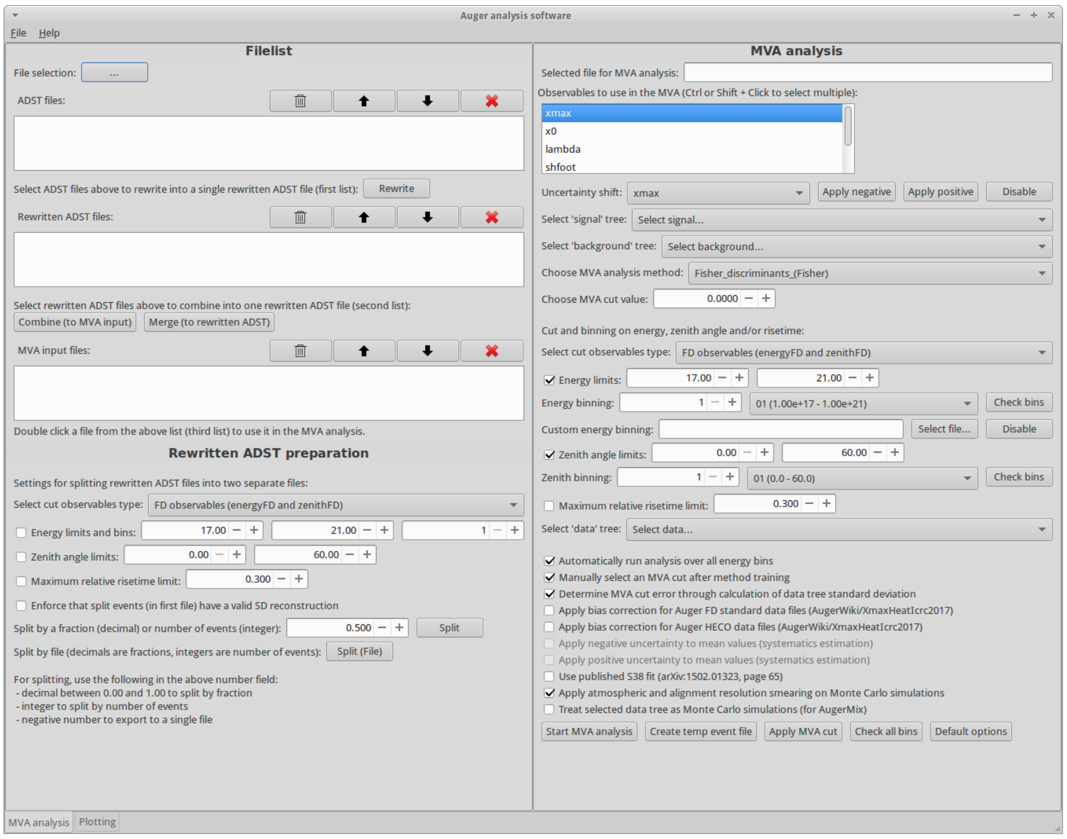
The GUI of the program for ADST version `XrYpZ` is executed with:

```
./start.sh vXrYpZ
```

Note that standard output to the terminal might slow down the program considerably. It is thus advised to save the standard output into a file with:

```
./start.sh vXrYpZ > results/printout.txt
```

This opens the initial graphical interface of the program:



File selection:

ADST files:

Select ADST files above to rewrite into a single rewritten ADST file (first list):

Rewritten ADST files:

Select rewritten ADST files above to combine into one rewritten ADST file (second list):

MVA input files:

Double click a file from the above list (third list) to use it in the MVA analysis.

Rewritten ADST preparation

Settings for splitting rewritten ADST files into two separate files:

Select cut observables type: FD observables (energyFD and zenithFD)

☐ Energy limits and bins: 17.00 21.00 1

☐ Zenith angle limits: 0.00 60.00

☐ Maximum relative risetime limit: 0.300

☐ Enforce that split events (in first file) have a valid SD reconstruction

Split by a fraction (decimal) or number of events (integer): 0.500

Split by file (decimals are fractions, integers are number of events):

For splitting, use the following in the above number field:
- decimal between 0.00 and 1.00 to split by fraction
- integer to split by number of events
- negative number to export to a single file

MVA analysis

Selected file for MVA analysis:

Observables to use in the MVA (Ctrl or Shift + Click to select multiple):
xmax
x0
lambda
shfoot

Uncertainty shift: xmax

Select 'signal' tree: Select signal...

Select 'background' tree: Select background...

Choose MVA analysis method: Fisher_discriminants_(Fisher)

Choose MVA cut value: 0.0000

Cut and binning on energy, zenith angle and/or risetime:

Select cut observables type: FD observables (energyFD and zenithFD)

✓ Energy limits: 17.00 21.00

Energy binning: 1 01 (1.00e+17 - 1.00e+21)

Custom energy binning:

✓ Zenith angle limits: 0.00 60.00

Zenith binning: 1 01 (0.0 - 60.0)

☐ Maximum relative risetime limit: 0.300

Select 'data' tree: Select data...

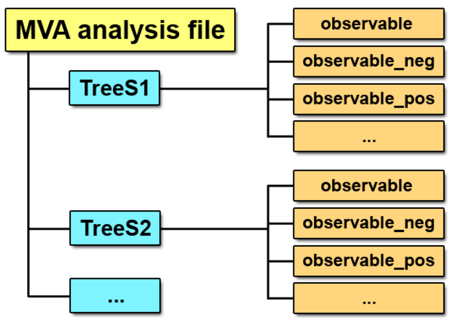
✓ Automatically run analysis over all energy bins
✓ Manually select an MVA cut after method training
✓ Determine MVA cut error through calculation of data tree standard deviation
☐ Apply bias correction for Auger FD standard data files (AugerWiki/XmaxHeatIcrc2017)
☐ Apply bias correction for Auger HECO data files (AugerWiki/XmaxHeatIcrc2017)
☐ Apply negative uncertainty to mean values (systematics estimation)
☐ Apply positive uncertainty to mean values (systematics estimation)
☐ Use published S38 fit (arXiv:1502.01323, page 65)
✓ Apply atmospheric and alignment resolution smearing on Monte Carlo simulations
☐ Treat selected data tree as Monte Carlo simulations (for AugerMix)

The analysis part of the interface is split into three different parts:

- **Filelist:** Handles opening of input files, which can be either Pierre Auger Observatory created ADST files (reconstructions from Offline) or the rewritten ADST files created with this software.

- **Rewritten ADST preparation:** Handles operations on input files that enable to create precise data sets (smaller simulation samples, mixed simulation samples,...).
- **MVA analysis:** Holds all options regarding the MVA analysis. This includes observables, signal/background samples, MVA methods, data binning and other options.

The **Filelist** part has an input file selector at the top, which passes input files through a check in order to determine their structure (handled by `SelectMvaFile` and `CheckFormat` functions in `./src/file_operations.cpp`). Valid selected files are then correctly displayed in one of the three listboxes: Top for ADST files, middle for rewritten ADST files and bottom for completely prepared MVA input files. The four buttons above each listbox can be used for ordering or removing listed files. Note that in some listboxes it is possible to select multiple files, by holding down `Ctrl` or `Shift` and selecting them.
Rewritten ADST files are created by selecting files in the top listbox and using the button `Rewrite`. This extracts observables needed for further analysis, which are listed in `./input/observables.txt`. The rewriting process is handled by functions `StartRewrite` in `./src/connect_functions.cpp` and the code in files `./src/adst_mva.cpp` and `./src/calc_observables.cpp`. In order for rewriting of new observables to work correctly, they need to be added in `./src/calc_observables.cpp`. Adding new observables is described in Chapter 2.2.
The middle listbox is used for either merging rewritten ADST files into single rewritten ADST files (for example merging different primary particle simulations for a mixed composition) or combining rewritten ADST files into MVA input files. These MVA input files will then typically hold at least three ROOT trees – one for signal, one for background and one for data. The structure of these files is displayed on the figure below.



The bottom listbox is only there to open MVA input files. Double-clicking a file on this list selects it as the current input to the MVA analysis.

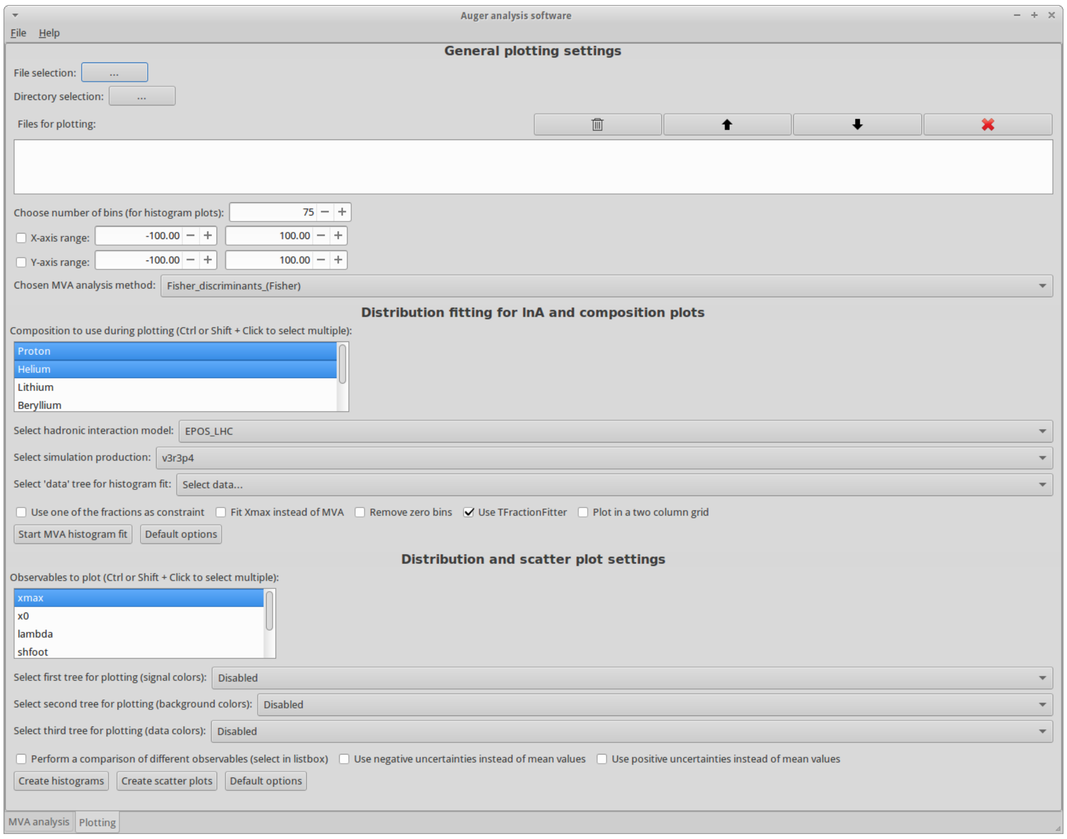
The **Rewritten ADST preparation** part can be used to do additional preparations on rewritten ADST files. This includes binning in energy, limiting the zenith angle and selecting the maximal relative risetime (only comes into play when using obsevable `risetimerecalc`). For example, this can be used, when we need a precise mass composition of simulation files by setting a number of events in each file. It can also be used to split a simulation set into two parts, for instance when creating MVA training and cross-validation sets. The split can be set as a fraction (using a decimal between 0 and 1) or as the number of events that will appear in the first file (1 or larger). If the splitting fraction is set to a negative number, all events will be filtered according to the settings and exported into a single file (useful for just binning in energy or limiting the zenith angle). The setting to enforce a valid SD reconstruction in the first file is there to make sure all observables have been correctly reconstructed. This helps, when we want a precise mass composition for mixed composition samples. The difference between `Split` and `Split (File)` is that the first obeys the fraction or number of events entered in the number field, while the second uses a file with splitting instructions. This file needs to have the same number of lines as the number of selected rewritten ADST files. Each line can hold a decimal number (between 0 and 1) to set fractions or an integer to set the number of events. The files will then sequentially check each line and perform the splitting automatically. In case a specific energy binning structure is needed, use custom energy binning in the MVA analysis part.

The **MVA analysis** part is the main part of this software and takes care of setting up and running the MVA analysis. The currently selected file is displayed in the top entry. In the listbox we can select observables that will be used as input features in the MVA analysis. Observables that are already included in the program are described in Chapter 2. The uncertainty shift setting is used to individually apply systematic uncertainties to observables in order to estimate the final systematic uncertainty of the analysis. Instructions for systematic uncertainty shifts need to be added to the `./src/mva_analysis.cpp` file as described in Chapter 1.2. The signal and background trees set training samples that the MVA analysis will try to separate according to the selected MVA method. All instructions for MVA methods are taken from the `./input/mva_options.txt` input file. The setting for the MVA cut value is currently only a holder for eventual event-by-event identification and is automatically or manually selected during the analysis as the cut for best separation between signal and background. Settings on binning and limits give the possibility to split the MVA analysis based on energy and zenith angle on-the-fly. The selection of the data tree is typically the sample we wish to determine mass composition for (either mock data or real data). Important to note is that some observables (for example `deltas38` and `deltarisetime`) are always transformed using the selected data tree. The additional settings are:

- *Automatically run analysis over all energy bins:*
  As the description suggests, this automatically runs the analysis over all selected energy bins. Note that this only works when we have a single MVA method selected.
- *Manually select an MVA cut after method training:*
  In case we wish to manually set the MVA cut value (mostly a holder for future event-by-event identification) after separation strength is known.

- *Determine MVA cut error through calculation of data tree standard deviation:* Currently only a holder for when statistical uncertainties can be determined directly from the MVA analysis. At the moment statistical uncertainties are handled by distribution fitting.
- *Apply bias correction for Auger FD standard data files (AugerWiki/XmaxHeatIcrc2017):*
  Apply bias corrections to the selected data tree according to the FD standard data approach, as described in [1].
- *Apply bias correction for Auger HECO data files (AugerWiki/XmaxHeatIcrc2017):* Apply bias corrections to the selected data tree according to the HECO data approach, as described in [1].
- *Use published S38 fit (arXiv:1502.01323, page 65):*
  When converting observable S_{1000} into ΔS_{38} instead use the previously published fitting function values, presented in [2].
- *Apply atmospheric and alignment resolution smearing on Monte Carlo simulations:*
  Apply the smearing and bias corrections for Monte Carlo simulation samples, as described in [3, 4]. Note that smearing only works for Hybrid of FD data.
- *Treat selected data tree as Monte Carlo simulations (for AugerMix):*
  Perform Monte Carlo corrections on the selected data tree. Otherwise it is assumed that the data tree has real data selected.

Moving to the `Plotting` tab in the bottom left, we get to the plotting and distribution fitting part of the program:


Auger analysis software

File Help

General plotting settings

File selection: ...

Directory selection: ...

Files for plotting:

Choose number of bins (for histogram plots): 75

☐ X-axis range: -100.00 100.00

☐ Y-axis range: -100.00 100.00

Chosen MVA analysis method: Fisher_discriminants_(Fisher)

Distribution fitting for lnA and composition plots

Composition to use during plotting (Ctrl or Shift + Click to select multiple):

- Proton ✓
- Helium ✓
- Lithium
- Beryllium

Select hadronic interaction model: EPOS_LHC

Select simulation production: v3r3p4

Select 'data' tree for histogram fit: Select data...

☐ Use one of the fractions as constraint ☐ Fit Xmax instead of MVA ☐ Remove zero bins ☑ Use TFractionFitter ☐ Plot in a two column grid

Start MVA histogram fit | Default options

Distribution and scatter plot settings

Observables to plot (Ctrl or Shift + Click to select multiple):

- xmax ✓
- x0
- lambda
- shfoot

Select first tree for plotting (signal colors): Disabled

Select second tree for plotting (background colors): Disabled

Select third tree for plotting (data colors): Disabled

☐ Perform a comparison of different observables (select in listbox) ☐ Use negative uncertainties instead of mean values ☐ Use positive uncertainties instead of mean values

Create histograms | Create scatter plots | Default options

MVA analysis | Plotting

The plotting part of the interface is split into three different parts:

- **General plotting settings:** Handles opening of MVA analysis output files, which are ROOT files similar in structure to MVA input files, but with an added MVA variable distribution.
- **Distribution fitting for lnA and composition plots:** Holds options for performing a distribution fitting procedure and extract elemental fractions from simulations. The complete fitting procedure is described in Chapter 1.3.
- **Distribution and scatter plot settings:** Holds options for plotting distributions of any variable or scatter plots of a collection of them. It is able to plot up to three data sets on a single plot.

The **General plotting** part has an input file or input directory selector at the top, which makes it possible to open MVA analysis output files. There are also a range of settings that will be used universaly for any plot created by this program. The number of bins sets the binning structure, which applies to both distribution fitting and just plotting variable distributions. X-axis and Y-axis ranges can force all plots to use specific ranges, but are currently not implemented in the plotting procedure. The MVA method setting ensures that any naming and ranges are correct for the output MVA variable.

The **Distribution fitting** part performs the distribution fitting procedure described in Chapter 1.3. In the listbox, we are able to select the composition mix that will be used for determining the mass composition. This composition must also be connected to the simulation trees that are in the MVA output file. Currently, the composition is setup in order to use this software directly with the Napoli shower library [5], which contains proton, helium, oxygen and iron as primary particles. The following settings choose the hadronic interaction model, data type and data tree we have used. The hadronic interaction model is the model used during event simulation and can be set in order to include previously published results [6, 7, 8] to composition plots. Using `None` will only show the results produced by this MVA analysis software. The simulation production just selects the Napoli library production we have used. The data tree can be the same as the data tree during the MVA analysis step, but we can also select the cross-validation or mock data sets. The remaining options are:

- *Use one of the fractions as constraint:*
  Instead of leaving the constraint
  $$\sum_{i=1}^{N} f_i = 1,$$
  up to the fitting approach, force one of the elemental fractions f_i to obey this constraint. For fitting with `TFractionFitter` this option can be disabled.

- *Fit Xmax instead of MVA:*
  Instead of using the output MVA variable distribution, use the depth of shower maximum X_{max} for the distribution fitting procedure. For further development of the software, this will be extended to use any observable.

- *Remove zero bins:*
  During distribution fitting, all distribution bins are considered in the fit, even if they are zero. Enabling this setting removes distribution bins that have zero values from the fit.

- *Use TFractionFitter:*
  Use the distribution fitting approach `TFractionFitter` [9] with maximum likelihood instead of the typical Minuit with χ^2.

- *Plot in a two column grid:*
  The combined elemental fraction plot places all elements in the composition in a column on one image. The two column grid splits this into two columns. Note that due to the restrictions of ROOT and the display size, the two column grid image might have worse quality.

The **Distribution and scatter plot** part plots distributions or scatter plots of observables in order to make comparisons. In the listbox, we can select between all observables, including the output MVA variable. We can then select up to three different trees from the output MVA file, taking signal colors (blue), background colors (red) or data colors (black). The following options give some additional flexibility:

- *Perform a comparison of different observables (select in listbox):*
  Instead of creating a normal plot, make a comparison between different observables from a single tree.

- *Use negative uncertainties instead of mean values:*
  Use mean values with added negative uncertainties instead of plotting mean values in the histogram or scatter plot.
- *Use positive uncertainties instead of mean values:*
  Use mean values with added positive uncertainties instead of plotting mean values in the histogram or scatter plot.

1.2 Handling uncertainties

1.3 Distribution fitting procedure

MVA and individual observable distribution fitting is carried out by combining a mixture of primary elements into a simulation distribution

$$H_{\text{sim}} = \sum_{i=1}^{N} f_i \, H_i, \tag{1.1}$$

where N is the number of elements in the mixture, f_i are fractions of individual elements and H_i are distributions of individual elements. The resulting distribution H_{sim} is then fitted to the data distribution H_{data}. Instead of using a χ^2 fitting procedure, a maximum likelihood fitting approach was used through the TFractionFitter fitting package [9] included in the ROOT framework. Specifically designed for fitting finite distributions with Poissonian statistics [10], it naturally satisfies the normalization condition

$$\sum_{i=1}^{N} f_i = 1 \tag{1.2}$$

and limits elemental fractions to a $[0, 1]$ range. The fitting procedure does not only enable MVA variable distribution fitting, but distribution fitting on any single observable. This skips the MVA analysis and tries to perform a distribution fit, with the same approach as described above. Fig. 1.1 shows the fitting procedure on MVA variable and X_{max} distributions for a single energy bin. The fitting procedure returns elemental fractions for all included elements in the composition.

1.4 Important input files

Any important input files, through which the user can adjust or change input settings, are located in the `./input` folder. The following are some of these files:

- **MVA method options:**
  Input file with instructions for each of the included MVA methods from the TMVA package (`./input/mva_options.txt`). The descriptive name will appear in the `Choose MVA analysis method` dropdown menu, while the method name is the name TMVA recognizes as an MVA method. Options for each of these methods can be adjusted, according to the TMVA

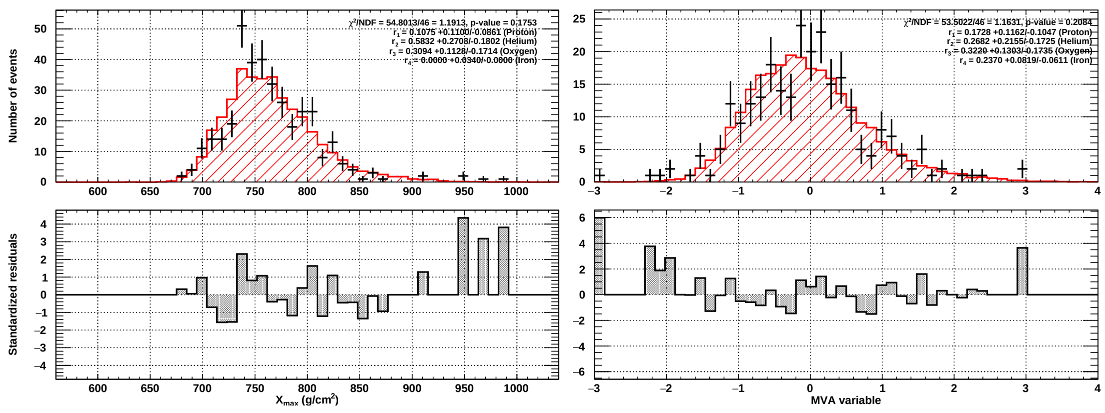


Figure 1.1: Example of distribution fits for X_{max} (left) and MVA variable from a Fisher analysis (right). A four elemental composition H_{sim} (red histogram) is fit onto a data distribution H_{data} (black points) using a maximum likelihood fitting approach. Bottom panels show standardized residuals ($R_i = \frac{n_i - m_i}{\sqrt{n_i}}$) between data and simulations.

documentation [11]. The file `./input/mva_options.txt` must not be renamed in order for this to work!

- **Observables file:**
  Input file with all observables we wish to extract from the ADST file and use in the MVA analysis (`./input/observables.txt`). The observable name appears in the observables listbox, while the selection sets the default selection of observables in the listbox. Note that the observable name must not include any whitespaces, because these are also used while naming output plots. Minimum and maximum limits are only used during plotting, where these two values are used to set limits for the appropriate axis. Axis labels give a description of the observable, which will be used during plotting. The file `./input/observables.txt` must not be renamed in order for this to work!

- **Custom energy binning file:**
  Example of a custom energy binning input file (`./input/custom_energy_bins.txt`). The two columns determine low and high values of each energy bin as shown in this example:

```
18.5 18.6
18.6 18.7
18.7 18.8
18.8 18.9
18.9 19.0
19.0 19.1
19.1 19.2
19.2 19.3
19.3 19.4
19.4 19.5
19.5 20.0
```

This file can be selected using the `Select file...` button in the custom energy binning section of the MVA analysis part of the GUI.

- **Splitting instructions file:**
  Example of a file, which sets instructions on how to automatically split multiple files. This is done through the `Split (File)` button in the Rewritten ADST preparation part of the GUI. Each line must include the number of events that will be saved to the first output file as shown in this example:

  ```
  476
  326
  89
  28
  33
  28
  1
  22
  0
  7
  0
  ```

  From this example, we have a total of 11 lines, which means this will sequentially split 11 selected input files automatically. When setting the splitting to 0, all events will be written to the second output file. Both output files will appear in the same folder as the input file, with an addition of `split1` or `split2` to their names, in order to separate them.

- **Previously published results files:**
  In order to plot previously published results in addition to the results created through the MVA analysis method, some input files must describe them:
  - File `./input/fractions_icrc17.txt` includes elemental fractions reported in [6] and can be found in [1].
  - Files `./input/lnA_moments_icrc2017_*.txt` include $\langle \ln A \rangle$ values reported in [6] and can be found in [1].
  - Files `./input/lnA_moments_prd2014_*.txt` include $\langle \ln A \rangle$ values reported in [7] and can be found in [12].
  - Files `./input/lnA_moments_sd*.txt` include $\langle \ln A \rangle$ values reported in [8] and can be found in [13].

Note that all of these input files have to be structured by the user accordingly or the plotting code needs to be adjusted. Functions handling the reading of these files are `ReadFracResultsFD`, `ReadLnaResultsFD` and `ReadLnaResultsSD`, all located in file `./src/mva_fit_histogram.cpp`.

2 Extensive air shower observables

This chapter describes the extensive air shower observables already included in the program and gives instructions on how to add new observables. Each of the descriptions also explains how it was extracted from the ADST file structure of Offline. Note that `fFile` denotes the reconstructed ADST file (`RecEventFile`) and `fRecEvent` denotes an event in this file (`RecEvent`). The rewriting and extraction of individual observables is handled in files `./src/adst_mva.cpp` and `./src/calc_observables.cpp`.

2.1 Observable descriptions

2.1.1 `xmax` – Depth of shower maximum

The depth of shower maximum $X_{\max}$ is the depth at which the extensive air shower reaches its maximum number of particles. This occurs due to energy loss of secondary particles, which at the critical energy of $\sim$GeV are less likely to produce new particles. The maximum clearly appears in measurements of UV light coming from the interaction of secondaries in the extensive air shower and nitrogen molecules. The so-called longitudinal profile of the shower (energy loss along the shower axis) is then fitted with a Gaisser-Hillas function

$$f_{\mathrm{GH}}(X) = \left(\frac{\mathrm{d}E}{\mathrm{d}X}\right)_{\max} \cdot \left(\frac{X - X_0}{X_{\max} - X_0}\right)^{\frac{X_{\max}-X_0}{\lambda}} \mathrm{e}^{\frac{X_{\max}-X}{\lambda}}, \tag{2.1}$$

where X is the depth along the shower axis, and λ, X_0, $X_{\max}$ and $\left(\frac{\mathrm{d}E}{\mathrm{d}X}\right)_{\max}$ are shape parameters of the shower. From the ADST structure, the depth of shower maximum measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
xmax = (acteyes[eye].GetFdRecShower()).GetXmax();
```

The reading and extraction of this observable is handled by function `GetXmax` in file `./src/calc_observables.cpp`.

2.1.2 `x0` – First interaction depth

X_0 describes the depth at which the initial interaction happened between the primary cosmic ray and an atmospheric molecule. This value depends on the simulation software and its atmospheric model, but is directly extracted from equation (2.1). From the ADST structure, the first interaction depth measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
x0 = (acteyes[eye].GetFdRecShower()).GetX0();
```

The reading and extraction of this observable is handled by function `GetX0` in file `./src/calc_observables.cpp`.

2.1.3 `lambda` – Gaisser-Hillas fitting parameter λ

Similarly to the previous two observables, λ is also a fitting parameter of the longitudinal profile and is directly extracted from equation (2.1). From the ADST structure, the λ parameter measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
lambda = (acteyes[eye].GetFdRecShower()).GetLambda();
```

The reading and extraction of this observable is handled by function `GetLambda` in file `./src/calc_observables.cpp`.

2.1.4 `shfoot` – Early development of the extensive air shower

The shower foot is an experimental observable which treats the longitudinal profile of a shower similarly as risetime determines the early deposit of energy from SD photomultiplier signals. The idea of this variable is to determine the depth at which the longitudinal profile reaches 10% of the maximum of the cumulative longitudinal profile. As such, the observable could be able to predict the early development of an extensive shower, which should be larger for heavier primary particles. Note, however, that this observable has not been thoroughly tested and should not be used, unless more detailed analysis is performed.
The shower foot has a default fractional value of `showerlimit` $= 0.1$, where it is estimated. For each of the FD buildings (denoted with `eye`), we first extract the longitudinal development of the shower with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
vector<double> *slantDepth = new vector<double>;
vector<double> *profiledEdX = new vector<double>;
vector<double> *profiledEdXerr = new vector<double>;
*slantDepth = (acteyes[eye].GetFdRecShower()).GetDepth();
*profiledEdX = (acteyes[eye].GetFdRecShower()).GetEnergyDeposit();
*profiledEdXerr = (acteyes[eye].GetFdRecShower()).GetEnergyDepositError();
```

This profile is then converted into its cumulative distribution with:

```
double x = 0., xerr = 0.;
vector<double> xfoot;
vector<double> yfoot;
vector<double> yerrfoot;
for(int i = 0; i < slantDepth->size(); i++)
{
  x += profiledEdX->at(i);
  xerr += profiledEdXerr->at(i);
  xfoot.push_back(slantDepth->at(i));
  yfoot.push_back(x);
  yerrfoot.push_back(xerr);
}
```

Once we have this distribution, we simply determine the final shower foot value, when the cumulative distribution reaches the `shfootlimit` fraction of its maximum

```
for(int i = 0; i < yfoot.size(); i++)
{
  if(yfoot[i] >= shfootlimit*(yfoot[yfoot.size()-1]))
  {
    // estimate the exact value between two points with linear interpolation
    // ...
    break;
  }
}
```

The reading and extraction of this observable is handled by functions `CalculateShowerFoot` and `GetShowerFoot` in file `./src/calc_observables.cpp`.

2.1.5 `energySD` – SD reconstructed primary energy

This is the reconstructed energy as measured by the surface detector (SD). From the ADST structure, E_{SD} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
energySD = sdrecshw->GetEnergy();
```

The reading and extraction of this observable is handled by function `GetSdEnergy` in file `./src/calc_observables.cpp`.

2.1.6 `energyFD` – FD reconstructed primary energy

This is the reconstructed energy as measured by one of the FD buildings. From the ADST structure, E_{FD} measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
energyFD = (acteyes[eye].GetFdRecShower()).GetEnergy();
```

The reading and extraction of this observable is handled by function `GetFdEnergy` in file `./src/calc_observables.cpp`.

2.1.7 `nrmu` – Number of muons at ground level

For the time being, the number of muons at ground level is only determined through the simulated part of the shower. As such, any real data events will be missing this information. From the ADST structure, the number of muons can be accessed with

```
GenShower *genshw;
genshw = fRecEvent->GetGenShower();
nrmu = genshw->GetMuonNumber();
```

The reading and extraction of this observable is handled by function GetNrMuons in file ./src/calc_observables.cpp.

2.1.8 nrstations – Number of triggered stations

This observable gives the number of SD stations that were triggered by an event. Each station is checked in order to see, if its signal is above the minimum VEM signal (minSignal), if it is not low gain saturated, if it is inside the limits for the distance from the shower axis (limitDistance), if its asymmetry corrected risetime has been calculated correctly and if its calculated uncertainty is positive. If the total number of stations tiggered by an event is lower than a predetermined limit of triggered stations for calculation of risetime (minPoint), this value will not be available. From the ADST structure, the number of triggered stations can be accessed with

```
vector<SdRecStation> actstations;
actstations = fRecEvent->GetSDEvent().GetStationVector();
int itemp;
for(int i = 0; i < actstations.size(); i++)
{
  if(actstations[i].IsCandidate())
  {
    /* --- Calculation of station risetime --- */
    bool btemp = true;
    if(actstations[i].GetTotalSignal() < minSignal) btemp = false;
    if(actstations[i].IsLowGainSaturated()) btemp = false;
    if( (actstations[i].GetSPDistance() < limitDistance[0]) || (actstations[i].
      ↪ GetSPDistance() > limitDistance[1]))
      btemp = false;
    if(risemean < 0.) btemp = false;
    if(riseerr < 0.) btemp = false;

    if(btemp)
      itemp++;
  }
}

if(itemp >= minPoints)
  nrstations = itemp;
```

The reading and extraction of this observable is handled by functions SetStationValues and GetNrStations in file ./src/calc_observables.cpp.

2.1.9 shwsize – SD signal at 1000 m from the shower axis

This returns the SD signal at 1000 m from the shower axis as determined by the fit of the lateral distribution function

$$S(r) = S_{1000} \left(\frac{r}{1000\,\mathrm{m}}\right)^{\beta} \cdot \left(\frac{r + 700\,\mathrm{m}}{1700\,\mathrm{m}}\right)^{\beta+\gamma}, \tag{2.2}$$

where r is the distance from the shower axis, S_{1000} is the SD signal at 1000 m from the shower axis, and β and γ are the LDF fitting parameters. This observable is an indicator of the muonic content of a shower, since showers induced by heavier primaries typically have a larger muonic content and form a larger footprint on the surface detector. From the ADST structure, S_{1000} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
shwsize = sdrecshw->GetShowerSize();
```

Note that the function `GetS1000` is deprecated and `GetShowerSize` is used instead. The reading and extraction of this observable is handled by function `GetShowerSize` in file `./src/calc_observables.cpp`.

2.1.10 `deltas38` – Relative SD signal

S_{1000} is however dependent on zenith angle, so it is reasonable to prepare a relative SD signal value (ΔS_{38}), which will be independent of both distance from shower axis and the zenith angle. The calculation follows these steps:

1. Scatter plots of station signal S_{1000} versus zenith angle ($\sec\theta$) are plotted for a range of energy bins.
2. These are then fitted with a scaled constant intensity cut function $f_{\text{scale}}(\theta)$ in order to remove dependence of S_{1000} on the zenith angle, and convert it to S_{38}.
3. When conversions to S_{38} are finished for all energy bins, they are combined and fitted with a power law function.
4. The relative signal ΔS_{38} is calculated by determining the separation between S_{38} of each event and the fitted power law function. Additionally, its uncertainty $\delta\Delta S_{38}$ is calculated through propagation of uncertainties.

S_{1000} are divided into energy bins between $10^{18.5}$ eV and $10^{20.0}$ eV. The binning step is selected to be $\log(E/\text{eV}) = 0.1$, except for the final bin in Pierre Auger data events, which spans between $10^{19.5}$ eV and $10^{20.0}$ eV. This has been selected in order to remedy the small amount of events at high energies, while still keeping as much information on simulations. Each of these subsets are then fitted with a scaled constant intensity cut function

$$f_{\text{scale}}(\theta) = S\, f_{\text{CIC}}(\theta) = S\, \left(1 + ax + bx^2 + cx^3\right), \tag{2.3}$$

where x is

$$x = \cos^2\theta - \cos^2(38^\circ), \tag{2.4}$$

and S, a, b and c are free fitting parameters. Parameter S will not come into play for further conversion of S_{1000}, but it represents the average S_{38} for the selected energy bin. The signal at 1000 m from the shower axis and at a reference zenith angle value of 38° is then defined as

$$S_{38} = \frac{S_{1000}}{f_{\text{CIC}}(\theta)}. \tag{2.5}$$

As an alternative to removing zenith angle dependence by fitting S_{1000} values, the previously published attenuation curve $f_{\text{CIC}}(\theta)$, with $a = 0.980 \pm 0.004$,

$b = -1.68 \pm 0.01$ and $c = -1.30 \pm 0.45$, can be used instead [14]. Once fits for all energy bins have been performed, all S_{38} values are fitted with a power law function

$$f_{38}(E_{\text{FD}}) = \left(\frac{E_{\text{FD}}}{A}\right)^{1/B}, \tag{2.6}$$

where A and B are free fitting parameters. As before, this fit can be replaced with previously published values of $A = (1.90 \pm 0.05) \times 10^{17}$ eV and $B = 1.025 \pm 0.007$ from [14]. At this point, the power law fitting function is used as a reference for calculating ΔS_{38} values

$$\Delta S_{38} = S_{38} - f_{38}(E_{\text{FD}}). \tag{2.7}$$

As opposed to other observables, this observable is calculated from SD station signals at 1000 m from the shower axis right before the MVA analysis step. This enables the user to select the data set, which will produce the reference zero value of the ΔS_{38} observable. The fitting is handled by function `SetDeltas` in file `./src/mva_analysis.cpp`, while the conversion is handled by functions `ConvertToS38` and `ConvertToDeltaS38` in file `./src/observables.cpp`.

2.1.11 `ldfbeta` – Lateral distribution function fitting parameter β

β is a fitting parameter in the lateral distribution function and is directly extracted from equation (2.2). From the ADST structure, β can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
shwsize = sdrecshw->GetBeta();
```

The reading and extraction of this observable is handled by function `GetBeta` in file `./src/calc_observables.cpp`.

2.1.12 `curvature` – The curvature of the shower front

This observable describes the curvature of the shower front in units of m^{-1} as measured by the surface detector (SD). From the ADST structure, the curvature can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
curvature = sdrecshw->GetCurvature();
```

The reading and extraction of this observable is handled by function `GetCurvature` in file `./src/calc_observables.cpp`.

2.1.13 `risetime, risetimerecalc` – SD risetime at 1000 m from the shower axis

The SD risetime is the time it takes the cumulative SD station signal to get from 10% to 50% of its maximal value. It is an indicator of electromagnetic versus muonic content in a shower, since muons produce sharp peaks in the station signal (formed high in the atmosphere) and electromagnetic particles form the broad distribution (formed closer to the surface). Shorter risetimes are thus typically attributed to showers induced by heavier primary particles. Because risetime is dependent on the distance from the shower axis, one of the reference points is to perform a fit through all SD station risetimes in an event

$$f(r) = 40\,\text{ns} + a \cdot r + b \cdot r^2 \tag{2.8}$$

and take the value at 1000 m from the shower axis. Important options for determining this observable are:

- **Range of valid SD station distances from the shower axis:**
  Setting minimum and maximum distances for valid SD station risetime enables removal of stations that would cause biases or have problems due to detector sampling. The default value of the range is between 0 m and 1800 m. The range used in [8] is however between 300 m and 1400 m for energies below $10^{19.6}$ eV, and 300 m and 2000 m for primary energies above $10^{19.6}$ eV.

- **Minimum total SD station signal:**
  The minimum SD station signal sets the threshold, which removes any stations with a low signal measurement. The default value of the threshold is 10 VEM (VEM is the Vertical-Equivalent-Muon unit and describes the signal produced by a muon passing the SD station vertically and through the station center). The threshold used in [8] reduces this to 5 VEM in order to not remove too many SD stations at lower energies.

- **Including low-gain saturated stations:**
  The SD signal is recorded in both the high-gain and low-gain channel. This is due to the large extent of signal sizes, when measuring extensive air showers. When the high-gain channel is saturated (signal is higher than the maximum of the ADC converter), analysis is performed on the low-gain channel. If the low-gain channel is saturated, the SD station is typically discarded from any further analysis.

- **Evaluation distance:**
  The evaluation or reference distance from the shower axis removes any dependency on distance from shower axis. Typically, the SD risetime value is selected at 1000 m from the shower axis.

- **Minimum number of triggered stations:**
  In order to perform a fit through triggered stations with equation (2.8), a minimum number of triggered stations needs to be determined. Typically, an event is considered to be valid if at least 3 stations are triggered and do not fall outside of previous selections mentioned above.

The difference between the `risetime` and `risetimerecalc` observables is only in selected options. `risetime` uses the default options and can be accessed from the ADST structure with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
risetime = sdrecshw->GetRiseTimeResults().GetRiseTime1000();
```

The reading and extraction of this observable is handled by function `GetRisetime` in file `./src/calc_observables.cpp`.
`risetimerecalc`, on the other hand, repeats the reconstruction of SD risetime, but uses options from [8]. The reading and extraction of this observable is handled by functions `CalculateRisetime` and `GetRisetime` in file `./src/calc_observables.cpp`.

2.1.14 `deltarisetime` – Relative SD risetime

A further complication arises, when checking the zenith angle dependence of SD station risetime. Due to that, it is reasonable to prepare a relative SD risetime value (Δ_R), which will be independent of both distance from shower axis and the zenith angle. The conversion uses a similar treatment to the Δ_s method introduced in [8]. During the conversion, the following steps are made:

1. Station risetimes with high-gain saturation are treated differently, because they introduce a bias that is visible while fitting.
2. Scatter plots of station risetime $t_{1/2}$ versus distance r are plotted for a range of zenith angle bins at a reference energy bin (between $10^{18.9}$ eV and $10^{19.1}$ eV). The reference energy bin is slightly larger than in [8], because it was primarily prepared for the hybrid dataset, which has smaller statistics. The energy reference bin and zenith angle binning (`binLimit`) is defined in the `SetDeltas` function of file `./src/mva_analysis.cpp`. This zenith angle binning removes the dependence of risetime on zenith angle.
3. Both sets of station risetimes (high-gain saturated and non-saturated) are separately fitted with benchmark functions. This removes the dependence of risetime on distance from the shower axis.
4. SD station relative risetimes Δ_i are calculated by determining the separation between the station risetime and the appropriate benchmark function inside the appropriate zenith angle bin.
5. Event-wide Δ_R and its uncertainty $\delta\Delta_R$ are calculated from the average of station relative risetimes.

The benchmark fitting functions are selected to be

$$t_{1/2}^{\text{bench,HG-sat}} = 40\,\text{ns} + \sqrt{A^2 + B^2 r^2} - A, \tag{2.9}$$

for a fit to high-gain saturated SD station risetimes and

$$t_{1/2}^{\text{bench}} = 40\,\text{ns} + M\,(\sqrt{A^2 + B^2 r^2} - A), \tag{2.10}$$

for a fit to non-saturated SD station risetimes. During the fitting procedure, high-gain saturated risetimes are fitted first, because they typically have a smaller spread. The two fitting parameters A and B are then fixed for the fit to non-saturated risetimes, where the only free parameter M describes the bias. With benchmark functions set in all zenith angle bins, we can now calculate the station relative risetime Δ_i, by determining the separation of $t_{1/2}$ to the appropriate benchmark function

$$\Delta_i = t_{1/2} - t_{1/2}^{\text{bench}}. \tag{2.11}$$

Calculation of Δ_i again differs from the approach used in [8], because low statistics of hybrid data makes the estimation of $\sigma_{1/2}$ difficult. Finally, Δ_R is simply the average of all stations relative risetimes involved in a single event

$$\Delta_R = \frac{1}{N} \sum_{i=1}^{N} \Delta_i, \tag{2.12}$$

where N is the number of triggered stations in a shower event. All uncertainties on S_{38} and ΔS_{38} are calculated through uncertainty propagation.
As opposed to other observables, this observable is calculated from SD station risetimes right before the MVA analysis step. This enables the user to select the data set, which will produce the reference zero value of the Δ_R observable. The fitting is handled by function `SetDeltas` in file `./src/mva_analysis.cpp`, while the conversion is handled by function `ConvertToDelta` in file `./src/observables.cpp`.

2.1.15 `aop` – Area-over-peak

The area-over-peak (AoP) is an observable that checks the area under the photomultiplier signal trace versus the peak value of the same trace. It is an indicator of electromagnetic versus muonic content in a shower, since muons produce sharp peaks in the station signal (formed high in the atmosphere) and electromagnetic particles form the broad distribution (formed closer to the surface). Showers with higher electromagnetic content will have a larger area under the signal, thus exhibiting a larger AoP value. The area (charge) and peak value of a specific photomultiplier in each of the SD stations are determined with

```
double charge, peak, aop;
vector<SdRecStation> stationVector =
  fRecEvent->GetSDEvent().GetStationVector();
for(int i = 0; i < stationVector.size(); i++)
{
  if(stationVector[i].IsCandidate())
  {
    // loop over photomultipliers
    for(int j = 1; j <= 3; j++)
    {
      charge = stationVector[i].GetCharge(j);
      peak = stationVector[i].GetPeak(j);
      aop = charge/peak;
    }
  }
}
```

Individual photomultiplier AoP values are then summed together into a mean AoP value, depending on the number of active photomultipliers and active SD stations in an event. The reading and extraction of this observable is handled by functions `CalculateAoP` and `GetAoP` in file `./src/calc_observables.cpp`.

2.1.16 `zenithSD` – SD reconstructed zenith angle

This is the reconstructed zenith angle as measured by the surface detector (SD). From the ADST structure, θ_{SD} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
zenithSD = sdrecshw->GetZenith();
```

The reading and extraction of this observable is handled by function `GetSdZenith` in file `./src/calc_observables.cpp`.

2.1.17 `azimuthSD` – SD reconstructed azimuth angle

This is the reconstructed azimuth angle as measured by the surface detector (SD). From the ADST structure, ϕ_{SD} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
azimuthSD = sdrecshw->GetAzimuth();
```

The reading and extraction of this observable is handled by function `GetSdAzimuth` in file `./src/calc_observables.cpp`.

2.1.18 `zenithFD` – FD reconstructed zenith angle

This is the reconstructed zenith angle as measured by one of the FD buildings. From the ADST structure, θ_{FD} measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
zenithFD = (acteyes[eye].GetFdRecShower()).GetZenith();
```

The reading and extraction of this observable is handled by function `GetFdZenith` in file `./src/calc_observables.cpp`.

2.1.19 `azimuthFD` – FD reconstructed azimuth angle

This is the reconstructed azimuth angle as measured by one of the FD buildings. From the ADST structure, ϕ_{FD} measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
azimuthFD = (acteyes[eye].GetFdRecShower()).GetAzimuth();
```

The reading and extraction of this observable is handled by function `GetFdAzimuth` in file `./src/calc_observables.cpp`.

2.1.20 `latitudeSD` – SD reconstructed galactic latitude

This is the reconstructed galactic latitude as measured by the surface detector (SD). From the ADST structure, B_{SD} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
latitudeSD = sdrecshw->GetGalacticLatitude();
```

The reading and extraction of this observable is handled by function `GetSdLatitude` in file `./src/calc_observables.cpp`.

2.1.21 `longitudeSD` – SD reconstructed galactic longitude

This is the reconstructed galactic longitude as measured by the surface detector (SD). From the ADST structure, L_{SD} can be accessed with

```
SdRecShower *sdrecshw;
*sdrecshw = fRecEvent->GetSDEvent().GetSdRecShower();
longitudeSD = sdrecshw->GetGalacticLongitude();
```

The reading and extraction of this observable is handled by function `GetSdLongitude` in file `./src/calc_observables.cpp`.

2.1.22 `latitudeFD` – FD reconstructed galactic latitude

This is the reconstructed galactic latitude as measured by one of the FD buildings. From the ADST structure, B_{FD} measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
latitudeFD = (acteyes[eye].GetFdRecShower()).GetGalacticLatitude();
```

The reading and extraction of this observable is handled by function `GetFdLatitude` in file `./src/calc_observables.cpp`.

2.1.23 `longitudeFD` – FD reconstructed galactic longitude

This is the reconstructed galactic longitude as measured by one of the FD buildings. From the ADST structure, L_{FD} measured by an FD building (denoted with `eye`) can be accessed with

```
vector<FDEvent> acteyes;
acteyes = fRecEvent->GetFDEvents();
longitudeFD = (acteyes[eye].GetFdRecShower()).GetGalacticLongitude();
```

The reading and extraction of this observable is handled by function `GetFdLongitude` in file `./src/calc_observables.cpp`.

2.2 Adding new observables

Adding new observables is easy and requires moderate knowledge of the ADST file structure from Offline and simple knowledge of the C++ programming language. In order to add a new observable, first determine a unique name for it and add it to file `./input/observables.txt` as a new observable. This will automatically take the new observable into the analysis process. Now, we just need to extract or calculate the value of the observable in the actual code of the program. See the code of other observables for additional help, while changing the code. Follow these steps to correctly edit the code (for example, lets name the new observable `sigma`):

1. Determine if this is an SD observable, FD observable or a simulated observable, as this will determine its placement in the source code.

2. Declare the function in `./include/adst_mva.h` inside the public part of the class. You are free to name the function in any way, but it is advisable to keep the naming conventions similar for all observables. For example, we can add an SD or simulated observable with

   ```
   float GetSigma(int type);
   ```

   To declare the function for an FD observable, instead use

   ```
   float GetSigma(int eye, int type);
   ```

3. Set the default value for the new observable in file `./src/calc_observables.cpp`. If this is an SD observable, add the code in the `ZeroSdObservables` function, if it is an FD observable, add it in the `ZeroFdObservables` function, and if this is a simulated observable, add it in the `ZeroSimObservables` function. For example, we can add

   ```
   cursig[j]->SetValue("sigma", -1.);
   ```

   Note that the default value of virtually all observables is -1 in order to be able to distinguish between events with a valid reconstruction, those that have an invalid reconstruction and correctly reconstructed events.

4. We now have to add the code for the actual calculation of the new observable. Again look at the type of the observable and correctly add it to the appropriate function `SetSdObservables`, `SetFdObservables` or `SetSimObservables`.

   - If the observable is an SD or simulated observable, it is enough to get the value from the function declared in step two. For example, we can add

     ```
     cursig[j]->SetValue("sigma", GetSigma(j));
     ```

   - If the observable is an FD observable, there is an additional step, which handles the combining of stereo events. In the function `SetFdObservables` find the use of functions that retrieve the observable value and add your own function declared in step two. For example, we can add

     ```
     else if(j == 9)
        dtemp[k] = GetSigma(i, k);
     ```

     Since all FD observables are determined together, we must also tell the code, if it has the correct observable. After the above code addition, find the use of observable names and add your own. For example, we can add

     ```
     else if(j == 9)
        *stemp = "sigma";
     ```

Make sure that the numerical value we are comparing to j in both above code segments is always one larger than for the previously used observable.

5. Define the function that we declared in the second step. Go to the end of file `./src/calc_observables.cpp` and add the definition, which will calculate or retrieve the new observable. For example, we can add

```
float AdstMva::GetSigma(int type)
{
   /* Code segment for the mean value of the observable */
   if(type == 0)
      // ...
   /* Code segment for the absolute negative statistical uncertainty of
       ↪ the observable */
   else if(type == 1)
      // ...
   /* Code segment for the absolute positive statistical uncertainty of
       ↪ the observable */
   else if(type == 2)
      // ...
}
```

   This will give your new observable a mean and statistical uncertainty values.

2.3 Applying corrections before the MVA training step

For certain purposes it is important to apply corrections to simulations and data just before starting the MVA training. This can be either calculation of a new observable (as for the above examples of ΔS_{38} and Δ_R), or applying corrections such as detector smearing, bias corrections,... Since sometimes, it is not known which dataset corresponds to simulations and which to data before the user specifies them, it is important to have a section of code, where all of the selection, cuts and corrections are applied before acquiring the final input dataset for the MVA analysis. This is performed through function `MvaSetTrees` in file `./src/mva_analysis.cpp`, which:

1. Reads the complete tree structure in the ROOT file for each of the included datasets. This includes all of the observables that have been saved to the file.
2. Prepares the same tree structure for a temporary tree in a new ROOT file, which will be used as the final input to the MVA analysis (`temporary_mvatree_file.root`). This way it is possible to apply on-the-fly changes to your input data.
3. Reads fitting results for ΔS_{38} and Δ_R, observables which require on-the-fly calculation.
4. Applies optional systematic uncertainty shifts in order to estimate the final systematic uncertainty of the analysis. This is implemented through

function `ApplyUncertainty` defined in file `./src/observables.cpp`. Note that only a select few systematics have been implemented so far (X_{max} and risetime), any others should be included into the `ApplyUncertainty` function.

5. Applies optional bias corrections to X_{max} and FD energy for standard FD and HECO data, as described in [1]. This is implemented through functions `ApplyCorrectionFD` and `ApplyCorrectionHECO` defined in file `./src/observables.cpp`.
6. Applies optional atmospheric and alignment resolution smearing to X_{max} for Monte Carlo simulations, as described in [3, 4]. This is implemented through function `ApplySmearing` defined in file `./src/observables.cpp`.
7. Converts absolute observables S_{1000} and t_{1000} into relative observables ΔS_{38} and Δ_R, through functions `ConvertToS38`, `ConvertToDeltaS38` and `ConvertToDelta` defined in file `./src/observables.cpp`.
8. Converts the zenith angle θ into a usable value for the MVA analysis $\sec\theta$. This is implemented through function `SetupZenith` defined in file `./src/observables`.
9. Creates a selection of events, which have valid observables, and fall within the set energy and zenith angle selection cuts. The selection is done with function `IsInsideCuts` defined in file `./src/mva_analysis.cpp`.
10. Writes out the output file structure and saves it to the output file, which is used for MVA training.

3 MVA analysis procedure

Once all observables are rewritten, cuts chosen, and the MVA input file created (following the steps in Chapter 2.3), the actual MVA analysis step can continue. On the graphical user interface (GUI) select options for the MVA analysis: from observables to use as machine learning input features, MVA method, signal, background and data trees, energy and zenith cuts, to additional options corresponding to bias corrections. The input file to the MVA analysis is created on-the-fly and named `temporary_mvatree_file.root`. This file doesn't include any events with invalid reconstructions and only takes events inside the selected energy and zenith angle cuts. The analysis is performed through function `StartMvaAnalysis` in file `./src/connect_functions.cpp` and follows these steps:

1. Creates the input file `temporary_mvatree_file.root` using functions `MvaTreeFile` and `MvaSetTrees` as described in Chapter 2.3.
2. Runs the `PerformMvaAnalysis` function in file `./src/mva_analysis.cpp`.
3. Opens both the created input file and an MVA output file. The MVA output file is named by the user, when saving the analysis. This output file can then later be used for classification purposes.
4. Prepares the MVA Factory object, which holds selected options and methods and handles the analysis.

   ```
   TMVA::Factory *factory = new TMVA::Factory("TMVAClassification", ofile, "
       ↪ !V:!Silent:Color:DrawProgressBar:Transformations=I;D;P;G;D:
       ↪ AnalysisType=Classification");
   ```

   The naming `TMVAClassification` just defines the name that will be used for the complete analysis and can be recalled to get MVA training results during the classification step. The third argument of the function give some options for the complete analysis, for which the complete list can be found in [11].
5. Sets the save directory for weights files created during the analysis. These files are needed during the classification step.

   ```
   (TMVA::gConfig().GetIONames()).fWeightFileDir = ((*currentAnalysisDir) +
       ↪ "/weights").c_str();
   ```

6. Only for ROOT version 6: Create the DataLoader object, which will hold all of the information connected to signal, background and MVA methods.

   ```
   TMVA::DataLoader *dataloader = new TMVA::DataLoader("");
   ```

   For ROOT version 5, you use the Factory instead of the DataLoader object.
7. Chooses observables that will be used as input features to the MVA analysis and chooses signal and background trees for training purposes.

```
// ROOT version 5
for(int i = 0; i < nrobs; i++)
  factory->AddVariable(observables[i].c_str(), 'F');
factory->AddSignalTree(signalTree, 1.0);
factory->AddBackgroundTree(backgroundTree[i], 1.0);

// ROOT version 6
for(int i = 0; i < nrobs; i++)
  dataloader->AddVariable(observables[i].c_str(), 'F');
dataloader->AddSignalTree(signalTree, 1.0);
dataloader->AddBackgroundTree(backgroundTree[i], 1.0);
```

Note that the option `'F'` denotes a decimal point of `float` or `double` precision. Another posibility is also to have an integer with option `'I'`.

8. Prepares training and test trees, by taking all event files from signal and background trees and then splitting them into two subsets.

```
factory->PrepareTrainingAndTestTree("", "", "nTrain_Signal=0:
    ↪ nTrain_Background=0:SplitMode=Random:NormMode=NumEvents:!V");
```

When using 0 for the number of signal and background events to use during training, it will take all available events for training and testing. It will additionally randomly split each set in half, and then use one for training and the other for testing.

Bibliography

[1] Pierre Auger Wiki pages for the mass composition working group, https://www.auger.unam.mx/AugerWiki/XmaxHeatIcrc2017.

[2] A. Aab *et al.* (The Pierre Auger Collaboration), *The Pierre Auger Cosmic Ray Observatory*, arXiv:1502.01323v5.

[3] A. Aab *et al.*, *Depth of maximum of air-shower profiles at the Pierre Auger Observatory. I. Measurements at energies above* $10^{17.8}$ eV, Phys. Rev. D **90** (2014) 122005.

[4] M. Unger, J. Bellido, *Supplementary material for the long* X_{max} *paper*, https://web.ikp.kit.edu/munger/Xmax/xmaxPaper2014/suppl_01_09_2014.pdf.

[5] Napoli shower library, https://www.auger.unam.mx/AugerWiki/NapoliLibrary.

[6] J. Bellido for the Pierre Auger Collaboration, *Depth of maximum of air-shower profiles at the Pierre Auger Observatory: Measurements above* $10^{17.2}$ eV *and Composition Implications*, PoS(ICRC 2017) (2017) 506.

[7] A. Aab *et al.*, *Depth of maximum of air-shower profiles at the Pierre Auger Observatory, II. Composition implications*, Phys. Rev. D **90** (2014) 122006.

[8] A. Aab *et al.*, *Inferences on mass composition and tests of hadronic interactions from 0.3 to 100 EeV using the water-Cherenkov detectors of the Pierre Auger Observatory*, Phys. Rev. D **96** (2017) 122003.

[9] TFractionFitter, https://root.cern.ch/root/html532/TFractionFitter.html.

[10] R. barlow, C. Beeston, *Fitting using finite Monte Carlo samples*, Comput. Phys. Commun. **77** (1993) 219 – 228.

[11] TMVA documentation, 4.2.0 version, trunk version.

[12] M. Unger Xmax pages for mass composition, https://web.ikp.kit.edu/munger/Xmax.

[13] SD pages for mass composition, https://www.auger.unam.mx/AugerWiki/XmaxSurfaceDetector.

[14] A. Schulz for The Pierre Auger Collaboration, *The measurement of the energy spectrum of cosmic rays above* 3×10^{17} eV *with the Pierre Auger Observatory*, ICRC 2013, 27 – 30, arXiv:1307.5059, 18.7.2013.