AIDA Alignment package user guide

version 1.0

Silvia Borghi, Christoph Hombach, Chris Parkes

University of Manchester
School of Physics and Astronomy
Christoph.Hombach@hep.manchester.ac.uk

Monday 28th April, 2014

Contents

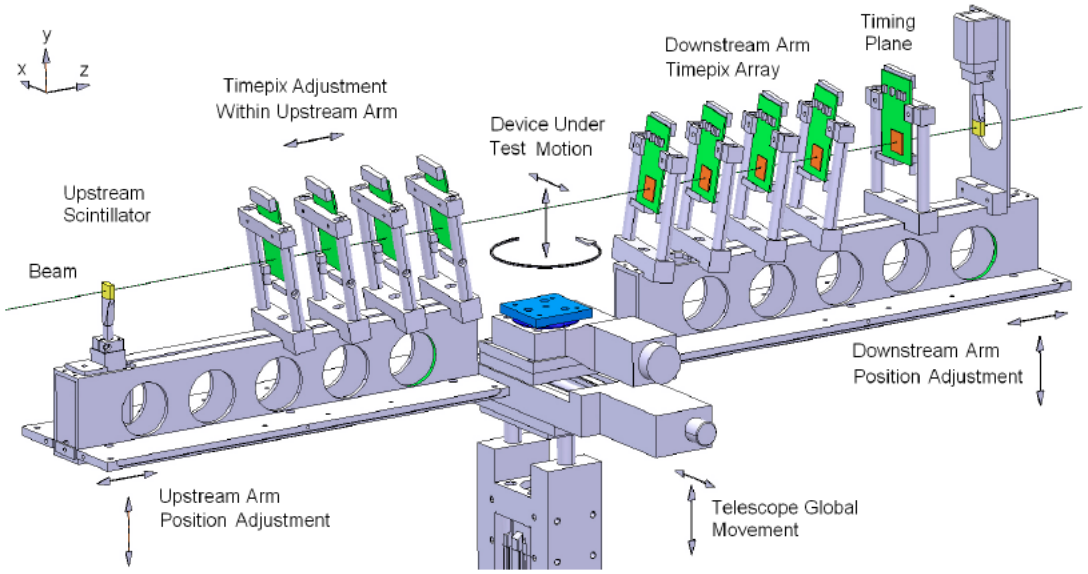

Figure 1: Layout of the TimePix Telescope with respect to the beam axis (from [2])

1 Introduction

The accurate determination of the positions of elements of a detector is essential for obtaining the optimal detector performance. Therefore alignment methods which evaluate the position of detector elements are needed for any particle physics experiment.
In this document software based alignment algorithms are discussed. A software-package called BACH (Basic Alignment and reconstruction CHain), written to provide a common software-tool to align telescope-like detectors, was developed. It describes the complete analysis chain of a telescope detector, from simulation to reconstruction. A telescope detector is used in testbeams to study certain aspects of a detector device. It consists of two arms on which sensors are placed. With those an accurate reconstruction of tracks can be performed. In between those a device under test (DUT) can be placed and be studied (see Fig. 1). The telescope-design is advantageous for detector studies, since it is, compared to complex detector systems, fairly simple and controllable. A precise, fast and reliable alignment procedure is especially important for these telescope detector systems, since its alignment can change frequently. For example, a common usage is performing an angle-scan of the DUT and therefore it needs to be aligned every time the position of the DUT changes.
A powerful alignment-algorithm, based on `MILLEPEDE` [1], an algorithm that inverts large matrices, and is implemented for alignment purposes, are discussed in detail. A C++-version of `MILLIPEDE` is implemented in the software. The analysis chain and alignment procedures presented here were successfully applied to the LHCb-AIDA telescope [2] and used for the undertaken analyses.

2 Alignment

Accurate measurements of particle trajectories rely on a precise knowledge of the positions of the elements of the detector. The first evaluation is provided by mechanical survey measurements. These often have a worse precision than the hit resolution. For a silicon detector

telescope the mounting precision of the silicon sensors is typically known to a few millimetres unless a dedicated survey measurement has been made, whereas the hit resolution could be at a few micron scale. Since the setup of the telescope can change on a run-by-run basis due to interventions or temperature changes, more precise survey measurements often cannot be performed during the data taking. Nevertheless a good survey measurement is required as a starting point for track based alignment-algorithms and should be carried out wherever possible.
Most software-based alignment algorithms for tracking detectors are based on track residuals. A residual is the distance between a measured hit point on a sensor and the interception point of the corresponding track with the sensor-plane. Misalignments would bias the residual-distribution as the real position of a sensor differs from the assumed position in the track reconstruction.
The alignment constants can be evaluated from residuals using a range of methods. The underlying idea is to solve a minimisation problem that yields the optimal set of alignment constants given the measured residuals. Two basic approaches can be defined: iterative techniques and global alignment techniques. A complete overview of the alignment methods used in several experiments were presented at "LHC Detector Alignment Workshop" [3].
One type of iterative procedure determines the alignment parameters using the knowledge of the shape of residuals distributions as a function of one or more coordinates. As the fitted track depends on the alignment constants, this method requires to repeat the procedure several iterations to determine the optimal solution. This technique has been used to align for example the SLD vertex detector [4].
Another iterative approach solves the alignment problem inside the track fit: the set of alignment parameters evolves with every track being processed. This method is able to account for correlations between the individual alignment parameters. However, this is very CPU time consuming for large misalignments as it requires several iterations to converge. This approach is often used with Kalman filter track fit and updates the alignment parameters as part of the track fit [5].
A similar approach using the Kalman filter and including the covariance matrix has been applied in LHCb [6]. One advantage of this method is to facilitate an easy treatment of multiple scattering and energy loss.
In global alignment techniques the residuals are handled as a linear function of both track and alignment parameters. Thus the solution that minimises the total χ^2 can be obtained with a single iteration over the data. One common approach is based on `MILLEPEDE` and it determines the solution for all alignment constants (and track parameters) at the same time by a single matrix inversion. This technique is quite widely used, for example in LHC experiments [7], [8], [9], [10].
The alignment algorithm developed in the AIDA alignment package and presented in this report is based on the `MILLEPEDE`-method. In section 2.1 the mathematical method is discussed and its implementation for a telescope detector is described.

2.1 Mathematical description of the alignment algorithm

The algorithm is based on the `MILLEPEDE`-method, a non-iterative method by a single matrix inversion technique to minimize a χ^2 function. This approach takes into account the correlation between the track parameters and the misalignment and determines both track

and alignment parameters in the χ^2 minimization. Indeed, the residuals can be written as a combination of the track parameters (called local parameters as they are different for each track) and the misalignment constants (called global parameters as they are the same for the whole data sample). The following section describes the `MILLEPEDE` principle of inverting a matrix by partitioning and its implication in testbeam scenarios.
In a linear model with no misalignments the measured quantity $\mathbf{x} = (x, y, z)$ is described as

$$\mathbf{x} = \left(\alpha_{ij}^T \delta_i\right) + \mathbf{r}, \tag{1}$$

where α is a matrix containing the track parameters, δ is a vector of input parameters, called *local* derivatives, and $\mathbf{r}$ is the residual vector. j represents the track-index and i the index of the module. In the case of linear tracks, these can be described as two straight lines in the the xz- and yz-plane and the x,y and z coordinate of a track j at a module i is:

$$x_i^{track} = a_j \cdot z_i + b_j \tag{2}$$
$$y_i^{track} = c_j \cdot z_i + d_j \tag{3}$$
$$z_i^{track} = z_i. \tag{4}$$

Therefore $\alpha_{ij}^T = \begin{pmatrix} 1 & z_{hit} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & z_{hit} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & z_{hit} \end{pmatrix}$ and $\delta_i = \begin{pmatrix} b_j \\ a_j \\ d_j \\ c_j \\ 0 \\ 1 \end{pmatrix}$. If one takes misalignments into account, equation 1 is expanded by a term applying to *global* parameters.

$$\mathbf{x} = \left(\mathbf{a}^T \mathbf{d}\right) + \left(\alpha^T \delta\right) + \mathbf{r}. \tag{5}$$

Here, $\mathbf{d}$ contains the alignment parameter, $\mathbf{a}^T$ is a matrix of *global* derivatives. In general there are six misalignment transformation for each module: the translations in x, y, z and the rotations around the $x-$, $y-$ and $z-$axis (α, β, γ). Therefore we can write $\mathbf{d}^T = (\Delta_x, \Delta_y, \Delta_z, \Delta_\alpha, \Delta_\beta, \Delta_\gamma)$. A measured hit is expressed as $\mathbf{x}_{hit} = (x_{hit}, y_{hit}, z_{hit})$. In case of a misalignment the real hit $\mathbf{x}_{hit}^{real}$ is related to the measured hit by:

$$\mathbf{x}_{hit}^{real} = \Delta_\gamma \Delta_\beta \Delta_\alpha \left(\mathbf{x}_{hit} - \begin{pmatrix} \Delta_x \\ \Delta_y \\ \Delta_z \end{pmatrix} \right) = \Delta_R \left(\mathbf{x}_{hit} - \Delta_r \right). \tag{6}$$

Assuming rotations are small, Δ_R is given by:

$$\Delta_R = \begin{pmatrix} 1 & \Delta_\gamma & \Delta_\beta \\ -\Delta_\gamma & 1 & \Delta_\alpha \\ -\Delta_\beta & -\Delta_\alpha & 1 \end{pmatrix}. \tag{7}$$

Neglecting non-linear terms, equation 6 can be written as:

$$\begin{pmatrix} x_{hit}^{real} \\ y_{hit}^{real} \\ z_{hit}^{real} \end{pmatrix} = \begin{pmatrix} x_{hit} - \Delta_x + y_{hit}\Delta_\gamma + z_{hit}\Delta_\beta \\ y_{hit} - \Delta_y - x_{hit}\Delta_\gamma + z_{hit}\Delta_\alpha \\ z_{hit} - \Delta_z - x_{hit}\Delta_\beta - y_{hit}\Delta_\alpha \end{pmatrix}. \tag{8}$$

Hence the first term in eq 5 becomes

$$\left(\mathbf{a}^T\mathbf{d}\right) = \sum \left(\frac{\partial \mathbf{x}_{hit}^{real}}{\partial d}\right) d = \begin{pmatrix} -1 & 0 & 0 & 0 & z_{hit} & y_{hit} \\ 0 & -1 & 0 & z_{hit} & 0 & x_{hit} \\ 0 & 0 & -1 & -y_{hit} & -x_{hit} & 0 \end{pmatrix} \begin{pmatrix} \Delta_x \\ \Delta_y \\ \Delta_z \\ \Delta_\alpha \\ \Delta_\beta \\ \Delta_\gamma \end{pmatrix} \tag{9}$$

Assuming there is a set of $N = \nu \cdot n$ local measurements, with ν the number of tracks and n the number of modules, the sum of the squared residuals weighted by the variances σ_{ijk}, called χ^2, is given by:

$$\chi^2 = \sum_{i=0}^{n} \sum_{j=0}^{\nu} \sum_{k}^{x,y,z} \frac{\left(\left[x_{ij} - \left(\mathbf{a}_{ij}^T\mathbf{d}_i\right) - \left(\alpha_{ij}^T\delta_j\right)\right]_k\right)^2}{\sigma_{ijk}^2}. \tag{10}$$

For every module i and every track j there is a measured value x_{ij}. The alignment parameter in $\mathbf{d}$ are the same for every module i and are independent of the track j. Likewise the track parameters in δ_j depend only on the track j, not on the module. The variance σ_{ijk} for every measurement is assumed to be known.
This χ^2 function is a function of the global parameters and the local parameters as variables. This is the key to the alignment problem. If one finds a set of track and alignment parameters, that minimises the χ^2 function, one has found the set of parameters, which correspond to the real alignment. Since we only consider linear terms, this can be done in an analytically way. By differentiating χ^2 with respect to the global and local parameters to obtain the minimum one observes:

$$\left(\frac{\partial \chi^2}{\partial d}\right) = 2 \cdot \sum_{ijk} w_{ijk} \left[\mathbf{d}_i^T\mathbf{a}_{ij}\mathbf{a}_{ij}^T - \left(x_{ij} - \alpha_{ij}^T\delta_j\right)\mathbf{a}_{ij}^T\right]_k = 0 \tag{11}$$

$$\left(\frac{\partial \chi^2}{\partial \delta}\right)_j = 2 \cdot \sum_{ik} w_{ijk} \left[\delta_j^T\alpha_{ij}\alpha_{ij}^T - \left(x_{ij} - \mathbf{a}_{ij}^T\mathbf{d}_i\right)\alpha_{ij}^T\right]_k = 0, \tag{12}$$

given, that the tracks are independent. Here $w_{ijk} = \frac{1}{\sigma_{ijk}}$. Note that the global parameters have a dimension of the number of misalignment constants times the number of modules and the local parameters a dimension of the number of track parameters times the number of tracks. The total number of equations sums up to the number of global and local parameters. These equations can be reformulated and parameterised in the following way:

$$\sum_{ijk} w_{ijk}\left[\mathbf{a}_{ij}\mathbf{a}_{ij}^T\mathbf{d}_i^T + \mathbf{a}_{ij}\alpha_{ij}^T\delta_j\right] = \sum_{ijk} w_{ijk}\mathbf{a}_{ij}x_{ijk} \Leftrightarrow \sum_{jk} C_j\mathbf{d}_i + \sum_{j} G_j\delta_j = \sum_{j} b_j \tag{13}$$

$$\sum_{ik} w_{ijk}\left[\alpha_{ij}\alpha_{ij}^T\delta_j^T + \mathbf{a}_{ij}\alpha_{ij}^T\mathbf{d}_i\right] = \sum_{ik} w_{ijk}\alpha_{ij}x_{ijk} \Leftrightarrow \Gamma_j\delta_j + G_i^T\mathbf{d} = \beta_j, \tag{14}$$

with:

$$\left\{ \begin{array}{ll} C_j = \sum_{ik} w_{ijk}\mathbf{a}_{ij}\mathbf{a}_{ij}^T & G_j = \sum_{ik} w_{ijk}a_{ij}\alpha_{ij}^T \\ b_j = \sum_{ik} w_{ijk}\mathbf{a}_{ij}x_{ijk} & \beta_j = \sum_{j} w_{ijk}\alpha_{ij}x_{ijk} \\ \Gamma_j = \sum_{j} w_{ijk}\alpha_{ij}\alpha_{ij}^T & \end{array} \right\}. \tag{15}$$

These equations can be written in a matrix form, leading to:

$$\left(\begin{array}{c|c|c|c} \sum_j \mathbf{C}_j & \cdots & \mathbf{G}_l & \cdots \\ \hline \vdots & \ddots & 0 & 0 \\ \hline \mathbf{G}_l^T & 0 & \mathbf{\Gamma}_l & 0 \\ \hline \vdots & 0 & 0 & \ddots \end{array}\right) \cdot \left(\begin{array}{c} \mathbf{d}_i \\ \hline \vdots \\ \hline \boldsymbol{\delta}_l \\ \hline \vdots \end{array}\right) = \left(\begin{array}{c} \sum_j \mathbf{b}_j \\ \hline \vdots \\ \hline \boldsymbol{\beta}_l \\ \hline \vdots \end{array}\right). \tag{16}$$

The solution for the alignment problem is found, if the matrix on the left hand side is inverted and multiplied by the vector at the right hand side. This matrix is usually very large and inverting large matrices is very time consuming. For instance, if one had 8 modules to align and the six alignment parameters discussed above, $\sum_j C_j$ is a 48×48 matrix. In the case of 3D points in a telescope scenario, Γ_l has a dimension of 6×6. If one observes 1000 tracks the total dimension of the matrix to invert would be 6048×6048. Inverting this matrix in an acceptable amount of time is the key merit of `MILLEPEDE`. It uses the fact, that for an alignment problem one is not interested in the result of the track parameter. So the aim is to determine $\mathbf{d}$. The first step is modify the matrix equation. The rows corresponding to the track parameters are multiplied from the right with $G_l\Gamma_l^{-1}$ and those are added to the rows corresponding to the alignment parameters. The procedure leads to the simplified system:

$$\left(\begin{array}{c|c|c|c} \sum_j \mathbf{C}_j - \mathbf{G}_j\mathbf{\Gamma}_j^{-1}\mathbf{G}_j^T & \cdots & 0 & \cdots \\ \hline \vdots & \ddots & 0 & 0 \\ \hline \mathbf{G}_l^T & 0 & \mathbf{\Gamma}_l & 0 \\ \hline \vdots & 0 & 0 & \ddots \end{array}\right) \cdot \left(\begin{array}{c} \mathbf{d}_i \\ \hline \vdots \\ \hline \boldsymbol{\delta}_l \\ \hline \vdots \end{array}\right) = \left(\begin{array}{c} \sum_j \mathbf{b}_j - \mathbf{G}_j\mathbf{\Gamma}_j^{-1}\beta_j \\ \hline \vdots \\ \hline \boldsymbol{\beta}_l \\ \hline \vdots \end{array}\right). \tag{17}$$

With this modification the upper rows corresponding to the alignment parameters is no longer dependent on the track parameters. The alignment parameters can now be calculated by solving the equation system corresponding to these rows with a dimension of the number of global parameters. The interesting part can be written as:

$$\mathbf{C}'\mathbf{d} = \mathbf{b}' \text{ with } \left\{ \begin{array}{c} \mathbf{C}' = \sum_j \mathbf{C}_j - \mathbf{G}_j\mathbf{\Gamma}_j^{-1}\mathbf{G}_j^T \\ \mathbf{b}' = \sum_j \mathbf{b}_j - \mathbf{G}_j\mathbf{\Gamma}_j^{-1}\beta_j \end{array} \right\}. \tag{18}$$

The result can be found by solving $\mathbf{d} = \mathbf{C}'^{-1} \cdot \mathbf{b}'$. It should be noted, that the solution does not depend on the track parameters, only on the measured data and the global and local derivatives $\mathbf{a}$ and α.

The inversion is done using the so called 'Gauss-Pivot'-Method, which takes n steps. Each of these steps is as follows:

- Pick a so called *pivot element* C_{kk} on the diagonal.
- The pivot element is replaced by the negative inverse of the element.
- The elements C_{ik} and C_{kj} in the pivot row and column are divided by the pivot element C_{kk}.
- The elements C_{ij} that are neither on the pivot row nor column are transformed by

$$C_{ij}^{\star} = C_{ij} - \frac{C_{kj}C_{jk}}{C_{kk}}. \tag{19}$$

The negative inverse matrix is obtained after the pivot point was selected in any order in the n different positions of the diagonal. The algorithm will fail for a zero pivot element and become inaccurate if it is close to zero. Therefore the element with the greatest absolute value is chosen to be the pivot element. A more detailed discussion on this method can be found in [11].

2.2 Constraint equations

In the last chapter it was assumed, that there is only one set of alignment- and track parameters, that minimises the χ^2-function. In general this is not the case, as the function could have multiple minima. Hence there are several solutions, that do not correspond to the real detector alignment. These are called *weak modes*. They are trajectories in the space of alignment parameters which are minima. Therefore they are correlated misalignments of all parameters. Figure (2) shows a set of transformations for a telescope-like detector, that are geometrical representatives of solutions to the alignment problem, but deform the detector geometry.

Furthermore there are certain alignment parameters that do not have a large impact on the residual and on the χ^2-distributions. Consequently the alignment-algorithm has little sensitivity to those. For instance in a telescope scenario, with tracks being perpendicular to the sensors, a rotation around the $x-$ or $y-$ axis has a small impact on the residual distribution. These modes have to be treated very carefully.

One essential element of the alignment procedure is the determination of proper constraints to avoid displacement of the detector. There are two possible ways to do so. One way is to fix one or more detector elements. This means that the global parameters of these elements are not changed with respect to the survey position during the alignment procedure.

Another possibility is to introduce constraint-equations. Linear relationships between the alignment parameters, like

$$f^T \cdot \mathbf{d} = f_0, \tag{20}$$

can be taken into account by using the Lagrange multiplier method. For each constraint an additional term with an additional parameter λ is introduced to the χ^2:

$$\chi^2 = \sum_{i=0}^{n}\sum_{j=0}^{\nu}\sum_{k}^{x,y,z} \frac{\left(\left[x_{ij} - \left(\mathbf{a}_{ij}^T\mathbf{d}_i\right) - \left(\alpha_{ij}^T\delta_j\right)\right]_k\right)^2}{\sigma_{ijk}^2} + \lambda\left(f^T\cdot\mathbf{d} - f_0\right). \tag{21}$$

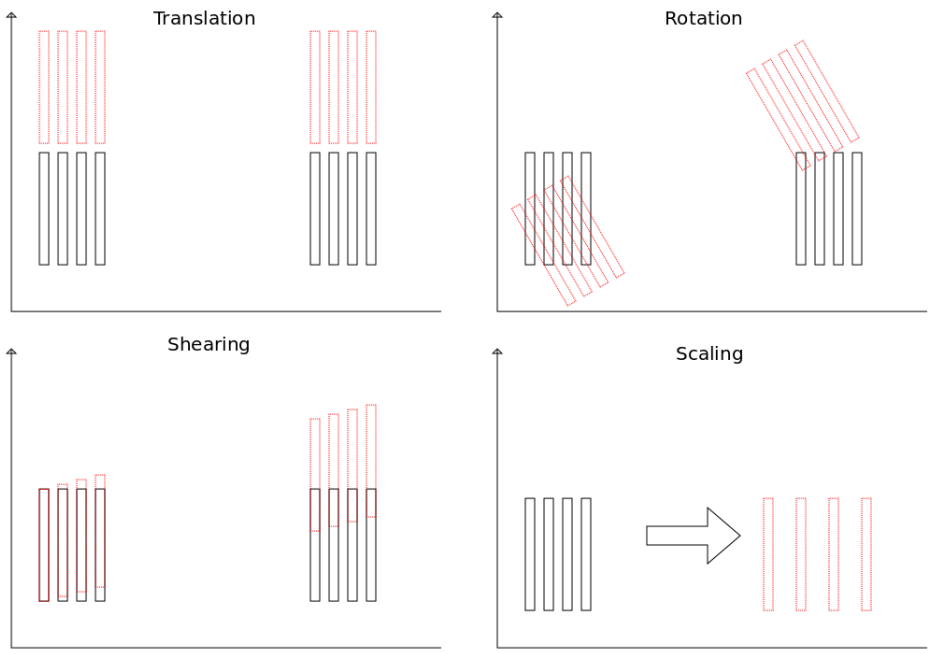

Figure 2: The four basic types of linear transformations for a telescope-like detector

Differentiating not just with respect to the global and local parameter, but also with respect to the Lagrange multiplier λ leads to:

$$\left(\begin{array}{c|c} C & f \\ \hline f^T & 0 \end{array}\right)\left(\begin{array}{c} d \\ \hline \lambda \end{array}\right) = \left(\begin{array}{c} b \\ \hline f_0 \end{array}\right). \tag{22}$$

Appropriate constraints are very important to align the detector. It is important to define the global reference frame. Since the modules are aligned with respect to each other, but not with respect to a predefined coordinate system, it is possible to introduce translation or rotation deformations. Therefore the position of one detector element is fixed. It defines the axes of the global frame and the other modules are aligned with respect to this one.
The tracks in the testbeam telescope are almost parallel to the z direction. Thus the alignment procedure is not sensitive to Δ_z. To avoid wrong scaling the z position of the modules is fixed.
A shearing can be introduced, if the survey alignment is set up in a way that it reconstructs an average slope to the track other than zero. The alignment procedure finds a solution, that keeps this slope and so leads to shearing. Figure (3) illustrates this scenario. To avoid such an effect, the constraint, that the tracks-slopes are on average zero need to be introduced in the process by a constraint equation as described in Ref. 20. The following linear constraint-equations are introduced:

$$\sum_{k=0}^{n} \Delta_x \frac{z_k - \bar{z}}{\sigma_z^2} = -n_{glo}\bar{a} \tag{23}$$

$$\sum_{k=0}^{n} \Delta_y \frac{z_k - \bar{z}}{\sigma_z^2} = -n_{glo}\bar{c}, \tag{24}$$

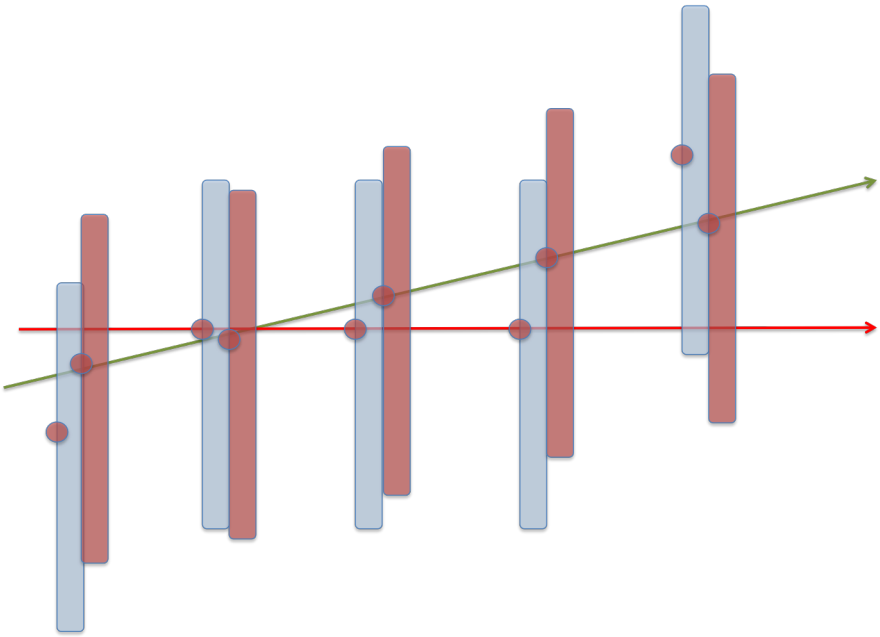

Figure 3: Track-Slopes and detector shearing introduced by misalignment. Blue boxes represent the initial alignment, red boxes the alignment result. A track parallel to the z-axis (red line) will be reconstructed as a sloped track (green line) and hence a detector shearing will be introduced.

where $\bar{a}$ and $\bar{c}$ are the average slopes of the tracks and

$$\bar{z} = \frac{1}{n_{glo}} \sum_{k=1}^{n_{glo}} z_k \tag{25}$$

$$\sigma_z^2 = \frac{1}{n_{glo}} \sum_{k=1}^{n_{glo}} (z_k - \bar{z})^2 \,. \tag{26}$$

n is the number of detector elements and z_k is the z-position of the k-th detector element. Similar equations need to be introduced to constrain the rotation around the z-axis. The average deviation of rotation around the z-axis per z-unit is evaluated to be:

$$\left(\frac{d\bar{\Delta}_\gamma}{dz}\right) = \frac{1}{n_{tracks}} \sum_{k=1}^{n_{tracks}} \pm\sqrt{\frac{a_k^2 + c_k^2}{b_k^2 + d_{y,k}^2}}, \tag{27}$$

where the sign depends on the quadrant of the x-y-plane in which the intercept lies (positive in the upper-left and the lower-right quadrant, negative in the other two). The equation to constrain the rotations around the z-axis is:

$$\sum_{k=1}^{n_{glo}} \Delta_\gamma \frac{z_k - \bar{z}}{\sigma_z^2} = -\left(\frac{d\bar{\Delta}_\gamma}{dz}\right). \tag{28}$$

3 BACH software description

The alignment algorithm is included in the software package BACH that contains the full reconstruction chain: clustering algorithm, pattern recognition, track fit, the alignment algorithm and analysis tools to fill histograms. This package allows the user to perform the

typical reconstruction and alignment of test beam data. For testing purpose at the beginning of the chain a set of data are simulated.
The detector description is stored in a .xml-file. The geometry is defined by reading in the number of modules, their survey positions and the alignment constants. The software is written for pixel sensors, so typical properties, like pixel number, pixel size or sensor thicknesses can be specified.
In the simulation tracks are created and their intercept point with the sensor surface are evaluated. The pixels traversed by the track are determined and a certain charge (ADC) is associated to that pixel, according to the distance of the track within that pixel. This gives a realistic set of data, which can be used as input for the clustering algorithm.
The clustering algorithm looks for adjacent pixel-hits and evaluates its cluster position. This is determined by the centre of gravity-method (CoG). The position of the cluster is the average pixel-positions weighted with the corresponding ADC value:

$$\bar{x} = \frac{\sum_i x_i ADC_i}{\sum_i ADC_i}. \tag{29}$$

If the cluster consists of only one pixel, the centre of the pixel corresponds to the cluster position. For silicon detectors the precision of the position is the best, if the track passes many pixel cells, since the CoG-method gives an ADC-weighted and therefore more precise result. But the more pixels are hit the less charge is deposited in the single pixel. It becomes more possible that pixels are excluded from the clustering, due to thresholds applied to reduce noise. In practice one obtains the best result if two pixels are hit. Figure 4 shows the difference between the true and the reconstructed x-position, for tracks perpendicular to a module, in which case the track hits mostly one pixel, and angled by 9°, where the best resolution for a 300 μm thick sensor is expected.
An example of an simple pattern recognition algorithm, which is sufficient for a testbeam analysis, is used in the program. The pattern recognition looks for clusters from the different sensors to form a collection of clusters. Since in this testbeam setup no magnetic field is applied and the tracks are straight lines mostly parallel to the z-axis, this algorithm is fairly simple. Starting from one cluster on a specified reference module, e.g. the first one of the telescope, the algorithm looks for cluster on the next modules that lie within a specified search window defined as a circle. If there are more than one cluster in this area, the one with the closest distance in the $x - y$-plane with respect to the reference cluster is chosen. Only the track candidates with clusters on every module of the telescope are kept to perform the track fit.
The track fit is performed using a linear least square technique. After the track fit a χ^2/ndof cut can be applied, to select only tracks of a certain quality.
Figure 5 shows the ratio of simulated over reconstructed track slope in x and y-direction. This shows the good agreement between simulated and reconstructed tracks.
Finally the alignment-procedure is performed. The clusters associated to a track are given to the alignment-algorithm as input. The alignment constants are determined as described in section 2. Fig 6 shows an example residual distribution before and after the alignment.
The BACH package includes all the elements to perform the common reconstruction of test beam data. It is based on different classes that could be to be adapted to different test beam setup and to replace one or more algorithm in a easy and flexible way. For instance the pattern recognition and the track fit could be replaced by more advanced algorithms that take into account effects due to magnetic field and apply correction due to multiple

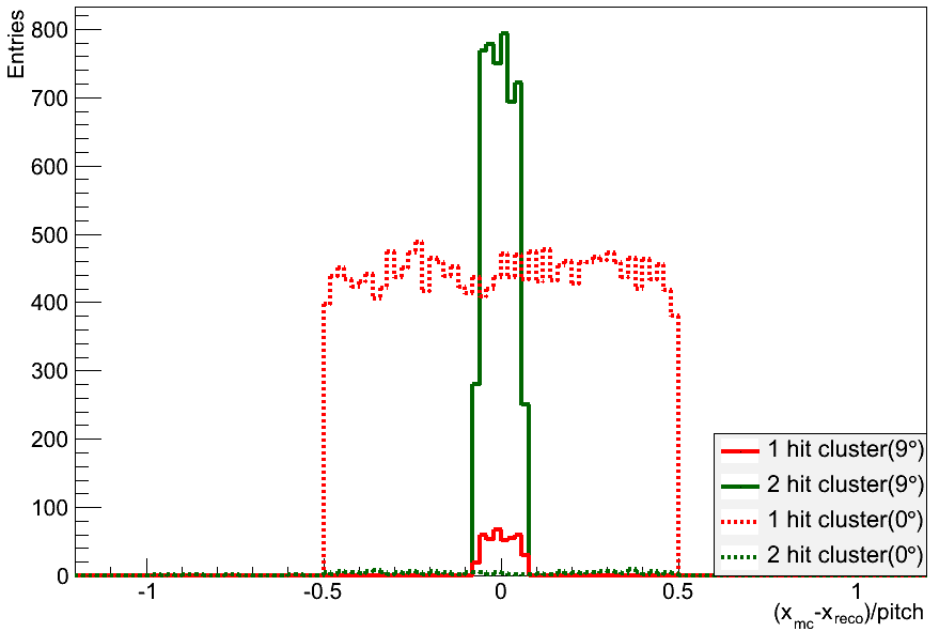

Figure 4: Difference between reconstructed (x_{reco}) and simulated (X_{mc}) cluster-position divided by the pixel size for a module perpendicular to the tracks and with a projected angle of 9°.

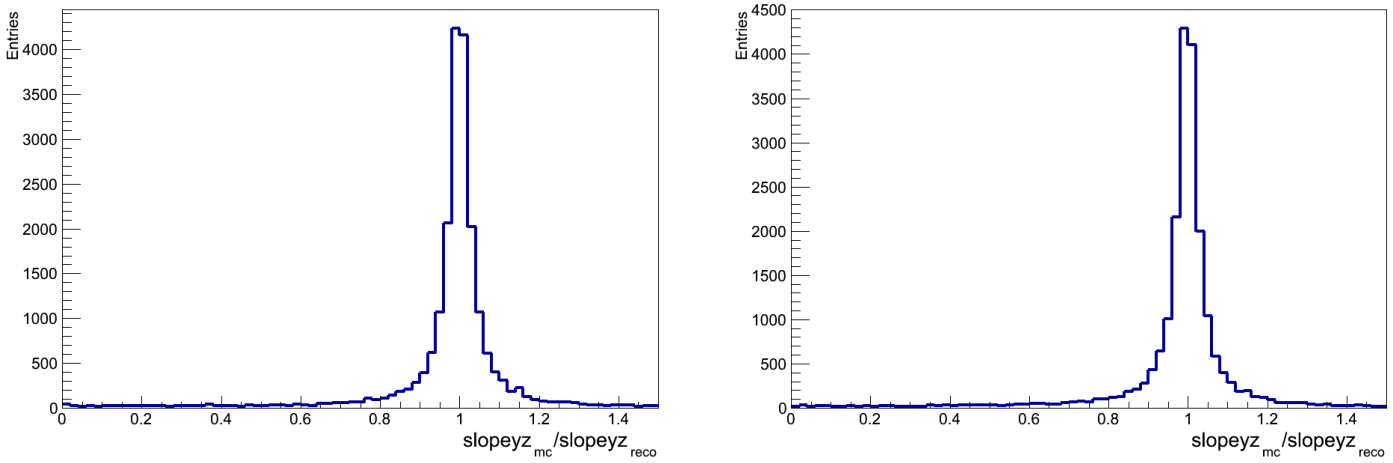

Figure 5: Ratio of simulated over reconstructed track slope in x and y-direction.

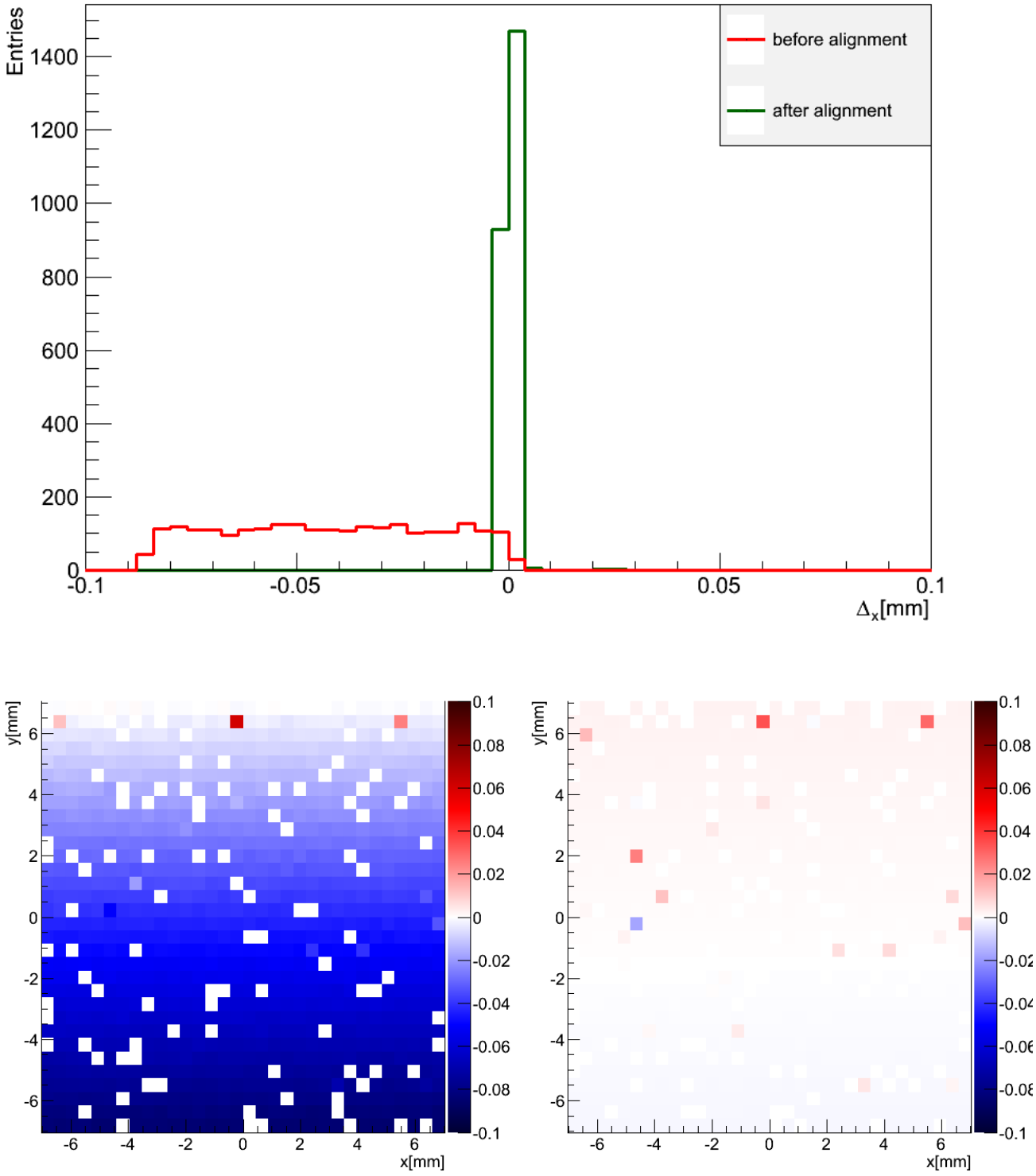

Figure 6: Example of a residual distributions before and after alignment. The upper graph shows the residual distribution in x for a test module. The lower graph shows the residual distribution for the complete sensor before (left) and after (right) alignment.

scattering with the detector material. The next sections describe the configuration and the software classes.

3.1 Setup

BACH requires a minimal set of external programs: namely Boost [12], a set of widely used C++ libraries; ROOT [13], a software package for particle physics analyses.
The source-code is available in the AIDA svn repository:

```
svn co https://svnsrv.desy.de/public/aidasoft/AIDAAlign/trunk/Tb Tb
```

The $ROOTSYS-variable has to be linked to the ROOT home-directory, and $PATH has to be defined as:

```
$ROOTSYS=<ROOT home-directory>
export PATH=$ROOTSYS/bin:$PATH
```

The BACH package is compiled by:

```
cd Tb/Bach/
make
```

3.2 Basic example

The geometry of the telescope is defined in a .xml-file, for example `geom/Telescope_geom.xml`. Each `<Module>`-tag defines one module, with its unique name, its position (`X, Y, Z, RX, RY, RZ`) and alignment constants (`dX, dY, dZ, dRX, dRY, dRZ`).

```
<Module name='G08-W0087' X='0' Y='0' Z='357' RX='0.157' RY='0.157' RZ='0'
   dX='0.0' dY='0.0' dZ='0' dRX='0.0' dRY='0.0' dRZ='0.0' />
```

The `xml/Configuration.xml` file defines the options of the different algorithms.
Firstly the number of events can be defined with the `NoOfEvt`-option. The Toy data options allow to define the number of tracks that are generated per event (`NoOfTracks`) and the detector geometry used for the generation of the data (`GeometryFile`). The generated slopes of the tracks follow a Gaussian distribution and can be defined by (`MeanX, MeanY, SigmaX, SigmaY`).
The characteristic of the pixel sensors, i.e. pitch (`PitchX, PitchY`), number of pixels (`NoOfPixelX, NoOfPixelY`), thickness of the sensor (`Thick`), and the geometry xml file used by the hit and track reconstruction, are defined in the GeometrySvc.
The pattern recognition options allow to define from which module the pattern recognition starts to look for the track candidates (`ReferenceModule`), the radius of the search window (`Distance`) and the cut on χ^2 quality (`Chi2ndof-cut`).

The Alignment options define the module constraint to the initial position (`FixedModule`). To reach a converging result it is often helpful to run the alignment algorithm for more than one time. The number of iterations can be defined with the Iterations-option.
The PlotTool algorithm creates a set of histograms to monitor the procedure. The output-file is defined (`OutputFile`) is defined here.

```
<?xml version="1.0" encoding="utf-8"?>
<Algorithms NoOfEvt="10" >



  <TbGeometrySvc>
    <constant name='GeometryFile' value="geom/Misalign_geom.xml" type="S"/>
    <constant name='PitchX'     value="0.055"               type="D"/>
    <constant name='PitchY'     value="0.055"               type="D"/>
    <constant name='NoOfPixelX' value="256"                 type="I"/>
    <constant name='NoOfPixelY' value="256"                 type="I"/>
    <constant name='Thick'      value="0.3"                 type="D"/>

  </TbGeometrySvc>

  <TbToyData>
    <constant name="NoOfTracks" value="25"             type="I"/>
    <constant name="GeometryFile" value="geom/Telescope_geom.xml" type="S"/>
    <constant name='MeanX'      value="0."                  type="D"/>
    <constant name='MeanY'      value="0."                  type="D"/>
    <constant name='SigmaX'     value="0.0001"              type="D"/>
    <constant name='SigmaY'     value="0.0001"              type="D"/>
  </TbToyData>

  <TbClustering/>

  <TbPatternRecognition>
    <constant name="ReferenceModule" value="C09-W0108"     type="S"/>
    <constant name="Distance"        value="1."            type="D"/>
    <constant name="Chi2ndof-cut"    value="50.0"          type="D"/>

  </TbPatternRecognition>

  <TbAlignment>
    <constant name="Iterations" value="2"                  type="I"/>
    <constant name="FixedModule" value="D09-W0108"         type="S"/>
  </TbAlignment>
  <TbPlotTool>
    <constant name="OutputFile"  value="out/Histograms.root" type="S"/>
  </TbPlotTool>
</Algorithms>
```

In this example the hit and track reconstruction runs over 10 events and first initialises the geometry defined in `geom/Misalign_geom.xml`. Toy-data are created with 25 tracks per event, using the geometry defined in `geom/Telescope_geom.xml` in order to perform a misalignment study. The full reconstruction chain and the alignment algorithm is run over these data. The new alignment constants are saved in `geom/Misalign_geom.xml` and a set of histograms are stored in `out/Histograms.root`.
To create a misalignment, a simple python script can be executed.

```
python scripts/MakeMisalign.py geom/Telescope_geom.xml
```

This creates a misaligned detector description based on the detector defined in `geom/Telescope_geom.xml`. The misalignment constants are randomly generated following a Gaussian distribution with a width of 100μm for x- and y-translation, 1 mrad for the rotation around the z-axis and 0.1 mrad for the rotations around the x- and y-axis. These settings can be changed in `scripts/MakeMisalign.py`.
The architecture of the software and the algorithms provided are discussed in the next section.

3.3 Software description

Figure 7 shows a schematic overview of the software layout. There are three directories, where classes and tools are stored. `BACH` is the main directory from where the program is run and compiled. The required geometry and configuration `.xml`-files are stored here, as well as scripts to create misaligned geometries, setup new algorithms and produce plots. `bach.cpp` initialises and executes all algorithms specified in the configuration `.xml`.
All algorithms inherit from one base class called `TbBaseClass`, since they all share certain common methods (configuration, initialize, execute, end event, finalize, see Sect. 3.5.1). This classes including the object definitions used by all algorithms, like geometry, hits, clusters, tracks are stored in `TbKernel`. As well, the `MILLEPEDE`-class (see Sect. 3.4) and a class to have an easy access to ROOT-objects can be found here.
`TbAlgorithm` contains the algorithms: TotData, Decoder, Clustering, Pattern Recognition, Alignment, PlotTool and Track Algorithms.
The setup of the alignment algorithm is discussed in the following section.

3.4 TbAlignment

In the execute-part `TbAlignment` picks up all tracks reconstructed in `TbPatterRecognition` and stores them in a container. The actual alignment procedure is applied after having run over all events.

For each track the function

```
bool TbAlignment::PutTrack( TbTrack *track, int nglo, int nloc, bool
    m_DOF[])
```

adds a row to `MILLEPEDE'S` matrix, according to equation 9. The following lines demonstrate the way it is implemented for the x coordinate.

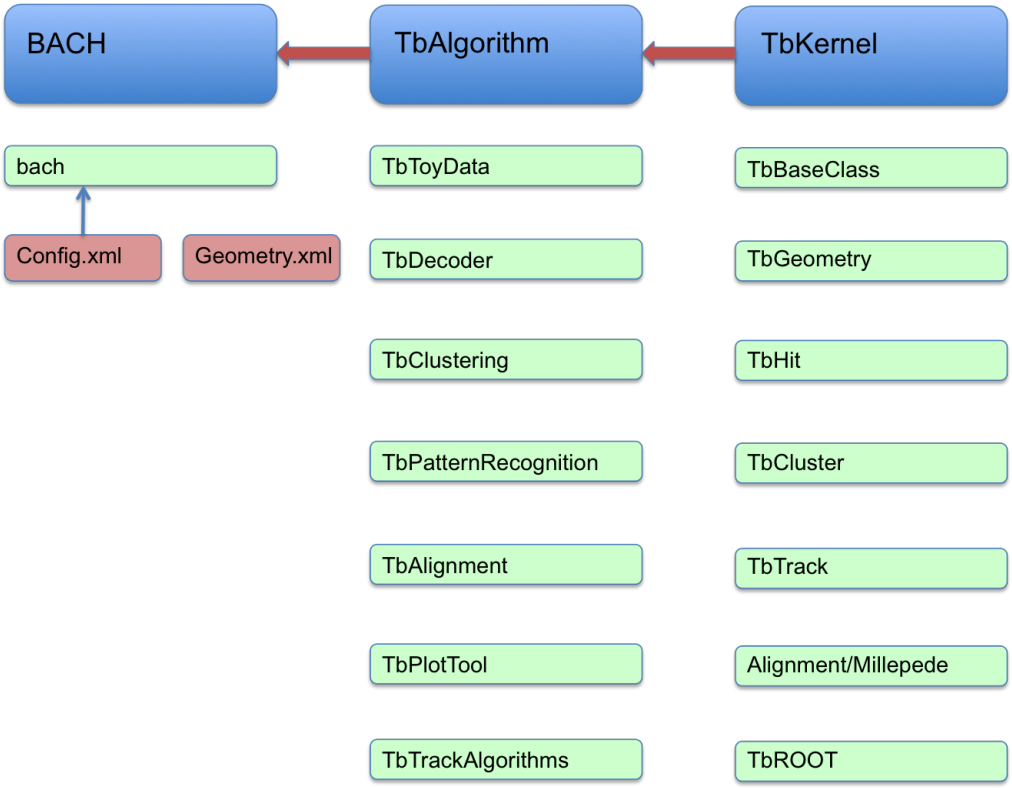

Figure 7: Schematic overview of software description

```
m_millepede->ZerLoc(&derGB[0],&derLC[0],&derNonLin[0],&derNonLin_i[0]);

// LOCAL 1st derivatives for the X equation

derLC[0] = 1.0;
derLC[1] = z_cor;
derLC[2] = 0.0;
derLC[3] = 0.0;
derLC[4] = 0.0;
derLC[5] = 0.0;
// GLOBAL 1st derivatives (see LHCbnote-2005-101 for definition)

if (m_DOF[0]) derGB[n_station]         = -1.0;   // dX
if (m_DOF[1]) derGB[Nmodules+n_station] = 0.0;  // dY
if (m_DOF[2]) derGB[2*Nmodules+n_station] = 0.0; // dZ
if (m_DOF[3]) derGB[3*Nmodules+n_station] = 0.0; // d_alpha
if (m_DOF[4]) derGB[4*Nmodules+n_station] = z_loc; // d_beta
if (m_DOF[5]) derGB[5*Nmodules+n_station] = y_cor; // d_gamma

sc = m_millepede->EquLoc(&derGB[0], &derLC[0], &derNonLin[0], &derNonLin_i[0],
                x_cor, err_x); // Store hits parameters
if (! sc) {break;}
```

Before evaluating the alignment constants, the constraint equations (eqs. 23 and 25) are defined in the following way.

```
for (int j=0; j<TelescopeMap.size(); j++)
  {

    double z_station = TelescopeMap[j];
    float z_0 = 1;
    int total = TelescopeMap.size();


    if (z_station >= 0){


      ftx[CorrectTelescopeMap[j]]             = 1.0;
      fty[CorrectTelescopeMap[j]+Nstations]   = 1.0;}

    ftz[CorrectTelescopeMap[j]+2*Nstations] = 1.0;
    frotx[CorrectTelescopeMap[j]+3*Nstations] = 1.0;
    froty[CorrectTelescopeMap[j]+4*Nstations] = 1.0;
    frotz[CorrectTelescopeMap[j]+5*Nstations] = (z_station-zmoy)/s_zmoy;
    shearx[CorrectTelescopeMap[j]]          = (z_station-zmoy)/s_zmoy;
    sheary[CorrectTelescopeMap[j]+Nstations] = (z_station-zmoy)/s_zmoy;


    fscaz[CorrectTelescopeMap[j]+2*Nstations] = (z_station-zmoy)/s_zmoy;



  }
// Here we put the constraints information in the basket


if (Cons[0] && DOF[0]) m_millepede->ConstF(&ftx[0], 0.0);
if (Cons[1] && DOF[0]) m_millepede->ConstF(&shearx[0], -(nglo)*(m_slopex));
if (Cons[2] && DOF[1]) m_millepede->ConstF(&fty[0], 0.0);
if (Cons[3] && DOF[1]) m_millepede->ConstF(&sheary[0], -(nglo)*(m_slopey));
if (Cons[4] && DOF[2]) m_millepede->ConstF(&ftz[0], 0.0);
if (Cons[5] && DOF[2]) m_millepede->ConstF(&fscaz[0], 0.0);
if (Cons[6] && DOF[3]) m_millepede->ConstF(&frotx[0], 0.0);
if (Cons[7] && DOF[4]) m_millepede->ConstF(&froty[0], 0.0);
if (Cons[8] && DOF[5]) m_millepede->ConstF(&frotz[0], -2.5*(m_alpha));
```

Now the alignment algorithm is ready to perform the global fit calling the following method.

```
      m_millepede->MakeGlobalFit(m_millepede->mis_const,
                                 m_millepede->mis_error,m_millepede->mis_pull);
```

To reach a better convergence, it is possible to run more than one iterations of the alignment algorithm. The detector's geometry is updated with the alignment constants evaluated by the alignment procedure. `TbGeometrySvc` then updates the geometry-xml file, which can be used for further analyses.

3.5 TbKernel

In `TbKernel` the core software components are stored. A brief description of each class is given in the following sections.

3.5.1 TbBaseClass

All algorithms, that are used in BACH inherit from `TbBaseClass` in order to provide functionality, which apply to all algorithm-classes. In one analysing circle all algorithms undergo the same steps:

```
virtual bool configuration();
virtual bool initialize(std::vector< std::pair< std::string, TbBaseClass* > >);
virtual bool execute(std::vector< std::pair< std::string, TbBaseClass* > >);
virtual bool end_event();
virtual bool finalize();
```

All parameters that can be defined by the user, are declared in `configuration()`. The functions `Const_(I,D,S,B)` provide this functionality, depending whether it is an integer (I), a double (D) a string(S) or a bool(B)-value. It can be initialised for example by

```
Const_I("Test_Int",5);
```

Introducing an integer with the name `"Test_Int"` and the default value 5. This value can be changed in the `.xml`-config by adding the line

```
<constant name="Test_Int"  value="3"      type="I"/>
```

which sets this value to 3. There is no recompiling necessary to change these values. Within the program, the parameter can be called by

```
Const_I("Test_Int");
```

In `inititalize()` all necessary variables, vectors, etc. can be initialised. If the algorithm requires any other algorithm-class (e.g. the pattern recognition needs information from the clustering-class), they should be called here.
The actual execution of the algorithm is done in `execute()`. BACH loops over the number of events that are specified in the configuration-xml and executes this part for each event.
After all algorithms are executed within one event, `end_event()` is called, to perform operations that need to be called at the end of each event.
After all events are proceeded, the `finalize()`-method finishes the algorithm.

3.5.2 TbGeometry

The geometry class first reads in the geometry specified in the geometry-xml file. `readConditionsXML()` creates for every module a TbModule object, in which its name and position is stored.
A new geometry-file (e.g. after the alignment procedure) can be written with `writeConditionsXML()`. The transformation of a point in the local module frame to the global detector frame and vice versa can be done with these methods:

```
virtual XYZPoint localToGlobal(const XYZPoint& p,
                               const std::string& id);
virtual XYZPoint globalToLocal(const XYZPoint& p,
                               const std::string& id);
```

3.5.3 TbHit

All information of a hit on the sensor are stored in TbHit, i.e the name of the module, the row and column and its ADC value. It also can contain the true position and track properties from the simulation.

3.5.4 TbCluster

All hits, that form the clusters, and the cluster positions are stored in TbCluster.

3.5.5 TbTrack

The clusters and track parameters (slope in xz- and yz-direction, the first state) are stored in TbTrack.

3.5.6 Alignment/Millepede

A c++ version of `MILLEPEDE` is stored here. The details of its functionality should be discussed in the following.

```
bool Millepede::InitMille(bool DOF[], double Sigm[], int nglo
                          ,int nloc, double startfact, int nstd
                          ,double res_cut, double res_cut_init, int n_fits)
```

This method initialises `MILLEPEDE`, which need certain information from some variables.

- `DOF[]` is an array with Boolean values, indicating whether a degree of freedom is aligned or not. In the example of a telescope detector, there are six degrees of freedom, the transitions in x, y and z and the rotations around these axes. So in this case `DOF[]` would be an array of the size six.

- `Sigm[]` is an array of the same size as `DOF[]`. It indicates the range in which `Millepede` looks for misalignment for each degree of freedom. So it should be filled with the approximate error on the alignment. It should not be chosen too small, since `MILLEPEDE`

will not be sensitive to the misalignment, but also not too big, as the error would become too big.

- `nglo` is the number of global parameter, e.g. the number of modules in the detector.
- `MILLEPEDE` can be run in an iterative mode, meaning that for large misalignment it might be necessary to lose the internal χ^2/ndof cut on tracks for the first iteration. `startfact` defines that value.
- The number of standard deviations for the χ^2/ndof-cut in the local fit is defined with `nstd`. If this is set to 0, no χ^2/ndof-cut is applied.
- To reject outliers, tracks with large residuals, and increase the quality of the fit, one can apply residual cuts. This value is defined with `res_cut`. For the first iteration, it is defined by `res_cut_init`.
- The number of fits is defined by `n_fits`. This is the number of tracks that are given to `MILLEPEDE`.

```
bool Millepede::ConstF(double dercs[], double rhs)
```

With this method the constraint equations are defined. `dercs` is the row containing the constraint equation derivatives, which is put into the final matrix. `rhs` is the Lagrange multiplier (see equation 22).

```
bool Millepede::ParSig(int index, double sigma)
```

The range within the global parameter is encouraged to vary is set here. `sigma` is the parameter given by `Sigm[]`, `index` the index of the array.

```
bool Millepede::ZerLoc(double dergb[], double derlc[],double dernl[],
    double dernl_i[])
```

The derivative vectors are reset. `dergb[]` are the global parameter derivatives, `dernl` the global non linear derivatives, `dernl_i` is the array linking the non-linear derivatives and their corresponding local parameters. `derlc` are the local parameter derivatives.

```
bool Millepede::EquLoc(double dergb[], double derlc[], double dernl[],
    double dernl_i[],
                    double rmeas, double sigma)
```

The equations are set. `rmeas` is the measured value, `sigma` is its error.

```
bool Millepede::FitLoc(int n, double track_params[], int single_fit)
```

This method performs the track fit to determine the local parameters and stores them for iterations. `n` is the number of the fit. `track_params[]` gives back the fitted track parameters and related errors. `single_fit`, if set to 1, performs the track fit and updates the track parameters without modifying the global matrix.

```
int  Millepede::SpmInv(std::vector<std::vector<double> >*
    v,std::vector<double>* b, int n,
                    std::vector<double>* diag,std::vector<bool>* flag)
```

This method solves the equation $V \cdot a = b$ (17) by inverting the matrix V by partioning.

```
bool  Millepede::MakeGlobalFit(std::vector<double>* par,
    std::vector<double>* error, std::vector<double>* pull)
```

In this method the global parameter fit is performed. `par` contains the global constants (misalignment constants), `error` the corresponding error.
For a more detailed look about theimplementation `MILLEPEDE` in an analysis-chain (see section 3.4).

3.5.7 TbROOT

This class gives an interface to commonly used ROOT-objects, like histograms.

4 Summary and outlook

In this report, the alignment software package, BACH, is described. The package includes the full reconstruction chain and the alignment algorithm for a typical telescope test beam setup. The alignment is based on MILLEPEDE. A complete description of the alignment method and its implementation is provided. This alignment method has been used in LHCb upgrade test beam with the LHCb-AIDA TimePix telescope proving the validity and its performance. The software package, presented in this report, can be easily adapted to use different pattern recognition or fit algorithm. An example on simulated data is included in the package and it can be easily adapted to any user case.

References

[1] V. Blobel and C. Kleinwort, *A New method for the high precision alignment of track detectors*, `arXiv:hep-ex/0208021`.

[2] K. Akiba *et al.*, *The Timepix Telescope for High Performance Particle Tracking*, arXiv:1304. 5175 (2013).

[3] *3rd LHC detector alignment workshop*, `http://indico.cern.ch/event/50502/`.

[4] D. J. Jackson, D. Su, and F. J. Wickens, *Internal alignment of the SLD vertex detector using a matrix singular value decomposition technique*, Nuclear Instruments and

Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **510** (2003), no. 3 233 .

[5] R. Frühwirth, T. Todorov, and M. Winkler, *Estimation of detector alignment parameters using the Kalman filter with annealing*, Journal of Physics G: Nuclear and Particle Physics **29** (2003), no. 3 561.

[6] W. Hulsbergen, *The global covariance matrix of tracks fitted with a Kalman filter and an application in detector alignment*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **600** (2009), no. 2 471 .

[7] M. Gersabeck *et al.*, *Performance of the LHCb vertex detector alignment algorithm determined with beam test data*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **596** (2008), no. 2 164 .

[8] C. Collaboration, *Alignment of the CMS tracker with LHC and cosmic ray data*, Submitted to JINST (2014), no. CMS-TRK-11-002, CERN-PH-EP-2014-028.

[9] ATLAS Collaboration, *Alignment of the ATLAS Inner Detector Tracking System with 2010 LHC proton-proton collisions at* $\sqrt{s} = 7$ *TeV*, Tech. Rep. ATLAS-CONF-2011-012, CERN, Geneva, Mar, 2011.

[10] A. collaboration, *Alignment of the alice inner tracking system with cosmic-ray tracks*, Journal of Instrumentation **5** (2010), no. 03 P03003.

[11] E. L. Volker Blobel, *Statistische und numerische Methoden der Datenanalyse*, Verlag Teubner Stuttgart, 1998.

[12] *Boost C++ librairies*, `http://www.boost.org/`.

[13] I. Antcheva *et al.*, *Root – a C++ framework for petabyte data storage, statistical analysis and visualization*, Computer Physics Communications **180** (2009), no. 12 2499 , 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.