JNA support for GBL Package

PF, Omar

07/01/2020

U.S. DEPARTMENT OF ENERGY | Stanford University

SLAC NATIONAL ACCELERATOR LABORATORY

Introduction

- Our current track reconstruction software is separated in two parts:
  - Track Finding and Fitting using seedTracker from LCSIM package
  - Track Refitting using General Broken Lines (GBL) using a java translation of (part) of the GBL cpp library

  GBL Repository
- Historically ported by Per and others.
- **The GBL java port (GBLJava) is only a partial implementation of the GBLCpp library** and, historically led to several questions whether if it was fully correct or not
- I've been maintaining the package since I joined HPS. Among other things I've:
  - Implemented a test example to validate the port
  - Fixed a bug in measurement without scatters GBLPoints
  - Ported unbiased residuals computation, treatment of holes-on-tracks as scatters
  - …
- Lot of things missing:
  - Refitting of trajectories from common vertex
  - Refitting with external constraints and measurement
  - Outlier removal procedures
  - …

Introduction

- Clearly, the current way to use GBL is not efficient when it comes to include new features in our reconstruction code.
- For every addition, it takes lot of time for **translating the code and testing and validating** it against the original library.
- Additionally, GBL library evolves (last svn push is Dec 2019 and I reported one bug in the CPP version to Claus)

- In our opinion, the current approach is **not sustainable** in the long run.

Full Port of GBL using JNA

- We should realise that, while moving forward, porting by hand every single feature of the GBL external library is not sustainable.
  - GBL moves forward (last release Dec '19), we need to update manually every-time
  - Error prone, requires validation and only partial functionalities are available.
- **I've decided to stop maintaing GBLJava and, together with Omar, we ported the GBL library using [Java Native Access (JNA)]()**
- JNA permits us to load an external C library and use it within hps-java
- It is supported by maven repository so it's easy to add it to the pom.xml file

```
       <dependency>
          <groupId>net.java.dev.jna</groupId>
            <artifactId>jna</artifactId>
            <version>5.5.0</version>
        </dependency>
```

tracking/pom.xml

Full Port of GBL using JNA

SLAC

- Since GBL is a C++ library, it's necessary to wrap the classes under C functions
- Together we wrote wrappers, around the latest GBL repository (see https://github.com/pbutti/GeneralBrokenLines) to call GBL from java using native language. We have validated the port against hps-java GBL and the GBLC++ code, see [hps-java jna-dev branch]()
- In hps-java one interface per class need to be made to call the C++ instance: for the moment support for GBLPoint and GBLTrajectory
- **The port fully support current hps-java calls to GBL.** Few adjustments need to be done to interface them to current refitting interfaces

```
extern "C" {

    GblTrajectory* GblTrajectoryCtor(int flagCurv, int flagU1dir, int flagU2dir) {

        return new GblTrajectory(flagCurv, flagU1dir, flagU2dir);

    }

    //Simple trajectory constructor wrapper
    GblTrajectory* GblTrajectoryCtorPtrArray(GblPoint* points[], int npoints,
                                             int flagCurv, int flagU1dir, int flagU2dir) {


        std::vector<GblPoint> aPointList;

        for (int i=0; i<npoints; i++) {
            //get the point pointer
            GblPoint* gblpoint = points[i];

            //add it to the vector
            aPointList.push_back(*(gblpoint));
        }

        return new GblTrajectory(aPointList, flagCurv, flagU1dir, flagU2dir);
    }

    //Simple trajectory constructor with seed wrapper

    GblTrajectory* GblTrajectoryCtorPtrArraySeed(GblPoint* points[], int npoints,
                                                 int aLabel, double seedArray[],
                                                 int flagCurv, int flagU1dir, int flagU2dir) {
```

The jan-dev branch have been tested on SLAC machines *without* a C++ installation of the GBL library and runs just fine as it is:
- JNA is used at run-time: if the JNA classes aren't called, no external library is needed
- We can rely on the *old* port of GBLJava for reconstruction, and things work as usual

Pros and Cons

PROS:
- Full Real GBL C++ library port
- No need for validation of every development
- Full and complete GBL functionality including outlier removals, external constraints,
proper computation of derivatives and support for additional local derivatives

CONS:
- Native Access comes with intrinsic overhead and our interface is not optimised: so it's slower (15-20%) on 100k tracks

- Bottom line:
  - **JNA includes a validated, maintained and largely used library with minimal work** (took us couple of days to implement)
  - It's slower than translating into Java, but remember that **GBL refitting *is not* where most the reconstruction time is lost** (that's the current seedTracker based track finding).
  - **If we pass to Kalman Filter, GBL is only needed for computing alignment derivatives:** in that case we care mostly about correctness and all the useful features.
- More modern alternatives to JNA exist:
  - https://github.com/bytedeco/javacpp

Summary

- Ported the *full* GBL C++ library to hps-java via JNA.
- **I will stop supporting and maintain GBLJava**
- JNA GBL C++ port is bit slower than the Java implementation. This is due to :

  - Intrinsic overhead by JNA

  - We didn't write a fully optimised interface
- HOWEVER:

  - GBL only take small amount of time in the event reconstruction

  - If we pass to KF tracks we don't need to refit them with GBL

  - We only need it for computing the local/global derivatives for MPII. Pede takes care of the fitting

  - The advantage in having a validated, complete and supported library I think overcomes speed.

  - Nonetheless there are alternatives to JNA: https://github.com/bytedeco/javacpp that claim to be overhead free.

  - Learning how to do a JNA/JAVACPP implementation in hps-java can be used to call other libraries that we might need in the future.

BACKUP

A real example - Track Parameters constrained alignment

- MPII refits tracks solving for df/dq at each p->p+Dp iteration
- If the local derivatives are “small” then Dq can be large to find the Chi2 minimum
- A track parameter un-constrained fit likely to result in a geometry which leads to biases.
- GBL Java port, doesn’t have a support for a refit with track parameters constraints, GBL C++ does.
- A seed-constrained fit is obtained adding a seed precision matrix to the X2.
- Easy to show that when computing dX2/dq that terms is added to the derivatives
- **In the case of the momentum, df/d(q/p) is inflated, which means that D(q/p) is smaller-> Dp is computed accordingly -> Momentum constrained aligment.**

track parameter derivatives

$$z_i = y_i - f(x_i, \boldsymbol{q}, \boldsymbol{p}) = \boxed{\sum_{j=1}^{\nu} \left(\frac{\partial f}{\partial q_j}\right) \Delta q_j} + \sum_{\ell \in \Omega} \left(\frac{\partial f}{\partial p_\ell}\right) \Delta p_\ell \,.$$

The dimension of the label set is arbitrary

$$n_{lc} = \text{number of local parameters} \qquad \text{array}: \left(\frac{\partial f}{\partial q_j}\right)$$

$$n_{gl} = \text{number of global parameters} \qquad \text{array}: \left(\frac{\partial f}{\partial p_\ell}\right); \text{ label-array } \ell$$

$$z = \text{residual} \quad (= y_i - f(x_i, \boldsymbol{q}, \boldsymbol{p})) \qquad \sigma = \text{standard deviation of the meas}$$

These need to get recomputed for each point and a new trajectory formed

$$\chi^2(\mathbf{x}) = \sum_{i=1}^{n_{\text{meas}}} (\mathbf{H}_{m,i}\mathbf{x} - \mathbf{m}_i)^T \mathbf{V}_{m,i}^{-1} (\mathbf{H}_{m,i}\mathbf{x} - \mathbf{m}_i) \quad \text{(from measurements)}$$

$$+ \sum_{i=2}^{n_{\text{scat}}-1} (\mathbf{H}_{k,i}\mathbf{x} + \mathbf{k}_{0,i})^T \mathbf{V}_{k,i}^{-1} (\mathbf{H}_{k,i}\mathbf{x} + \mathbf{k}_{0,i}) \quad \text{(from kinks)}$$

$$+ \boxed{(\mathbf{H}_s\mathbf{x})^T \mathbf{V}_s^{-1} (\mathbf{H}_s\mathbf{x}) \quad \text{(from external seed)} \quad (9)}$$

GBL Manual

Implementation of Momentum constrain in GBL Java

- I translated the code from GBL C++ to GBLJava for momentum constraint, tested it and seems like it's working in the right way (some checks on the derivatives should be done)
- Tested on MC-FEEs (thx Jeremy)
- Procedure:

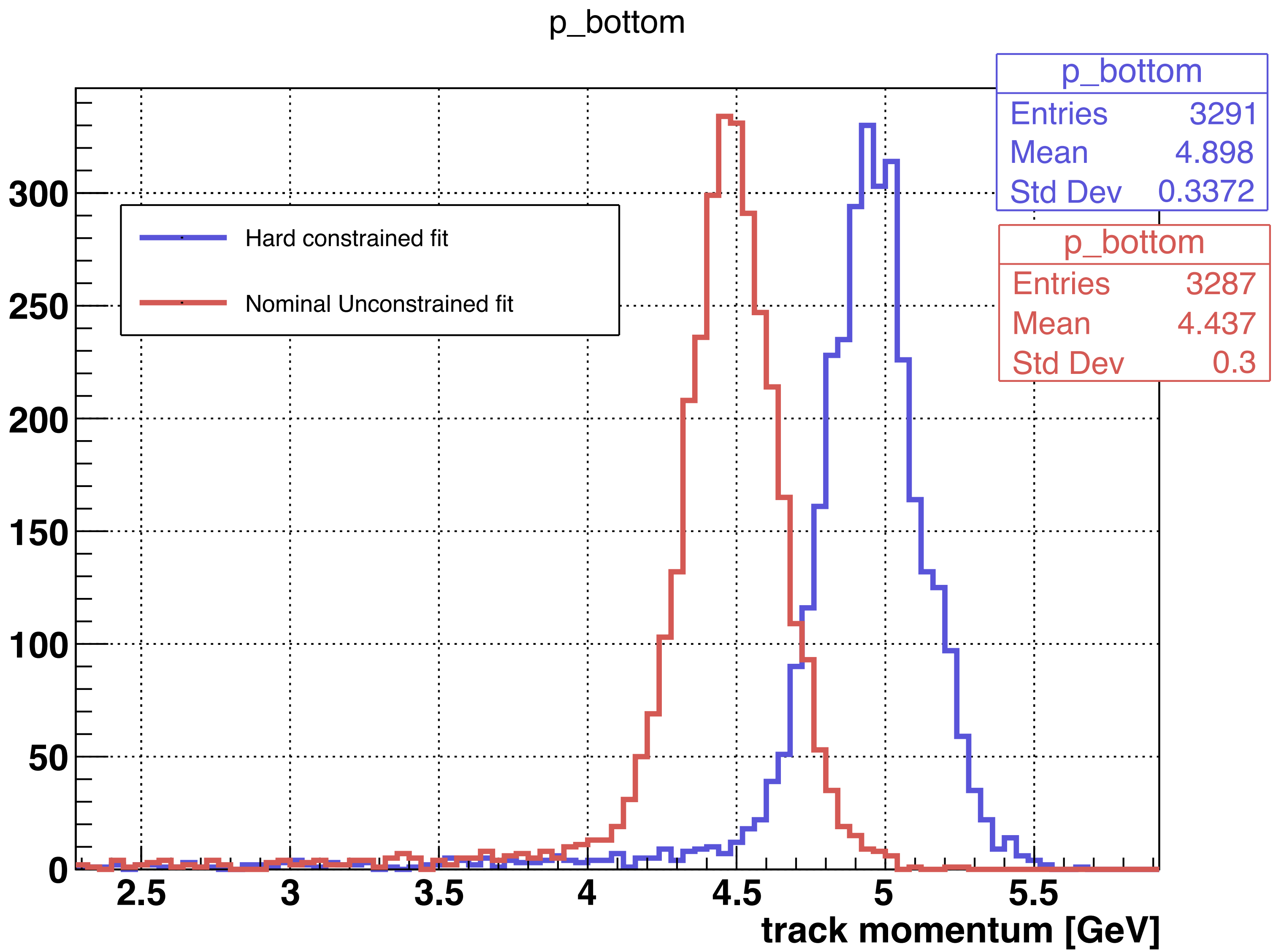
  - Take the initial helix

  - q/pT -> q/pT + d(q/pT) ==>

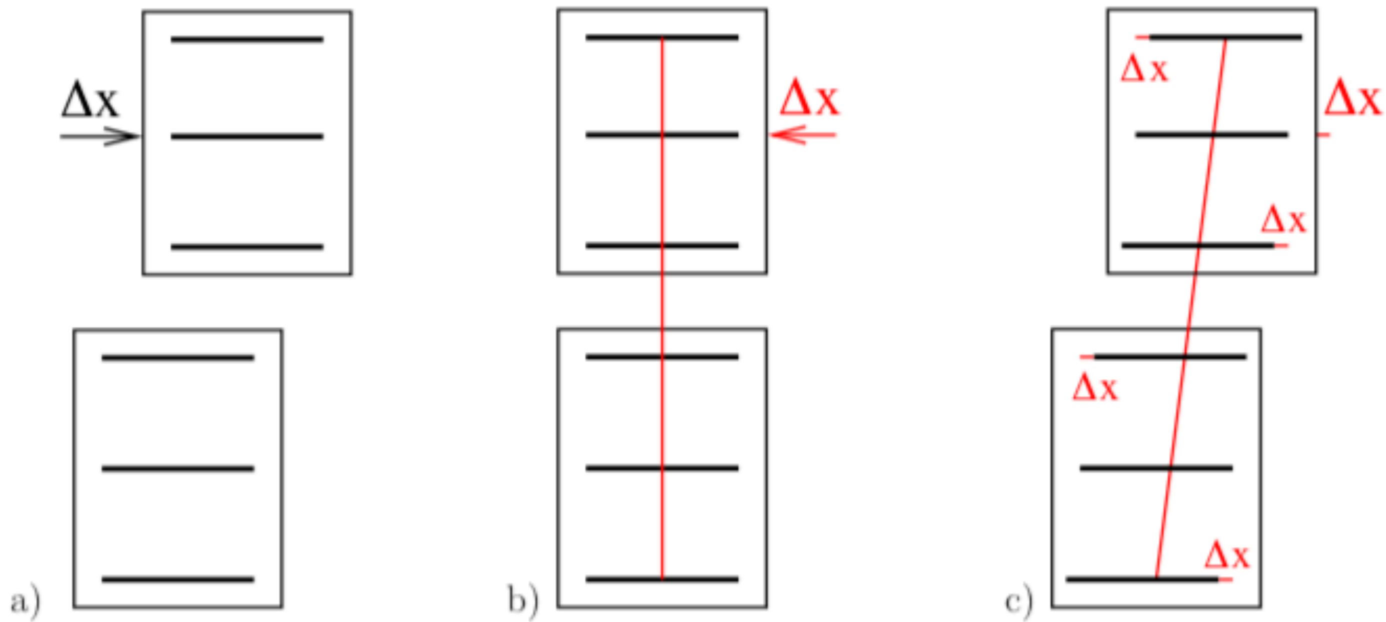
    w -> w + dw (curvature)

  - Refit with GBL nominally, with bias w/o contraint, with bias with constraint.
- Tested very large precision matrix [strong constraint]

Implementation of Momentum constrain in GBL Java



Why global structures first?



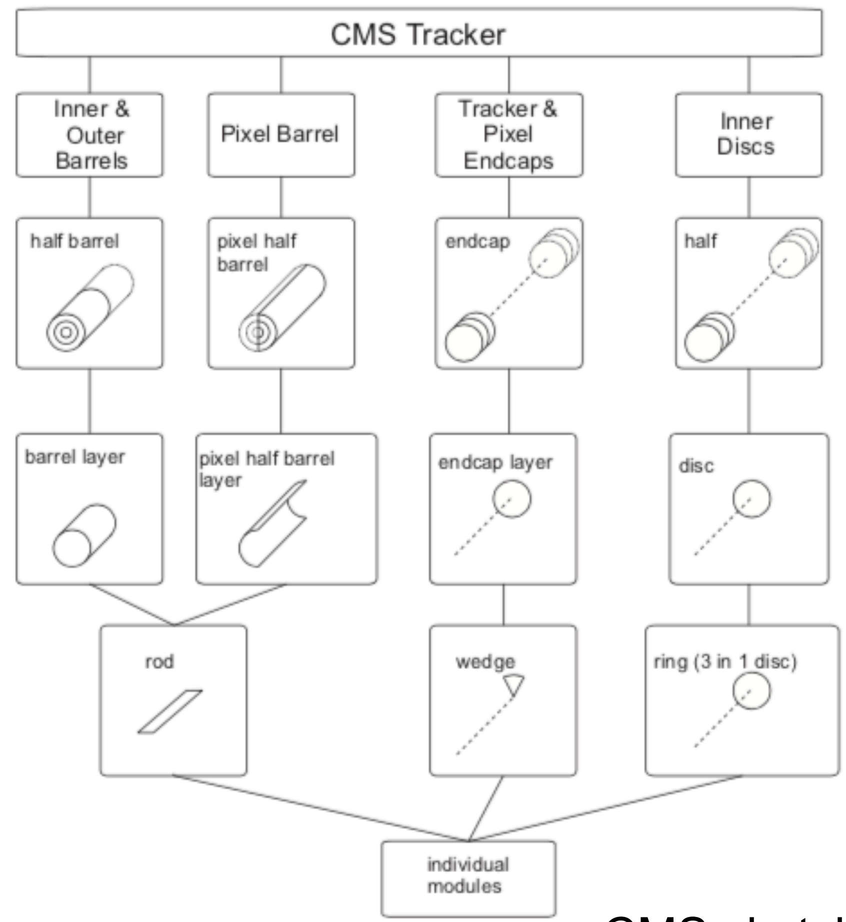
- Illustration of possible misalignment in a telescope.
- b is (a possible) solution if sub-telescopes are preferred
- c is (a possible) solution if single sensors are preferred
- In reality it depends of various factors including:

  - **Constraints** (what moves what not)

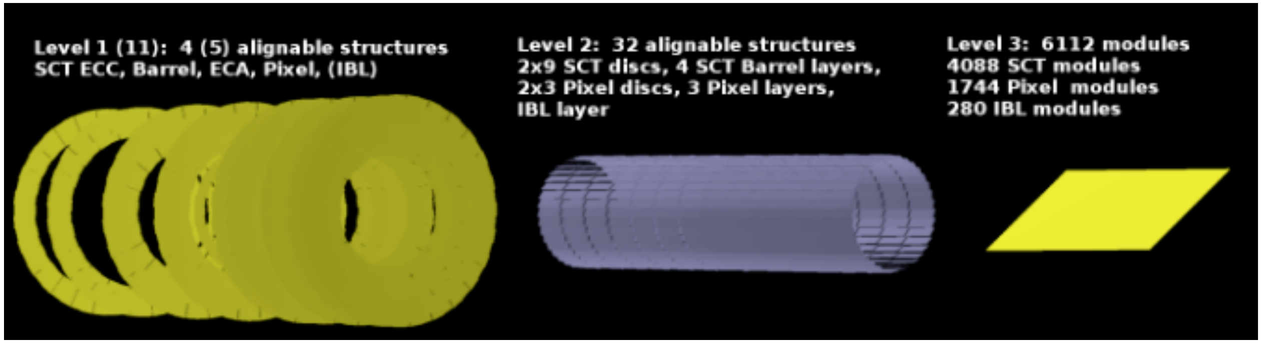
  - **Initial sensor position uncertainty** (we don't use any information on initial uncertainty in MPII solution)

Composite structure alignment

- What I would like to propose is to implement an hierarchical alignment procedure where we have alienable structures by MPII that aren't only sensors, but also sides, modules, UChannels and SvtBox.
- **This won't solve all of our problems outlined before, but should provide**:
  - **Same way to solve global and local misalignments**: just accumulate all information and decide which structure we want to align.
  - **Sensor positions and orientations will be relative to composite structures** and there is a **natural way to include constraints** to the solution.
  - **Composite structures will be aligned minimising the global χ^2** and correlations between DoF should be taken care of.
- This procedure is a standard in solving the alignment problem and has been used successfully by other experiments.



CMS sketch



ATLAS sketch

Math behind composite structures alignment

- Residuals are computed in the local coordinates (q) of a sensor and transformed to global frame (r) by

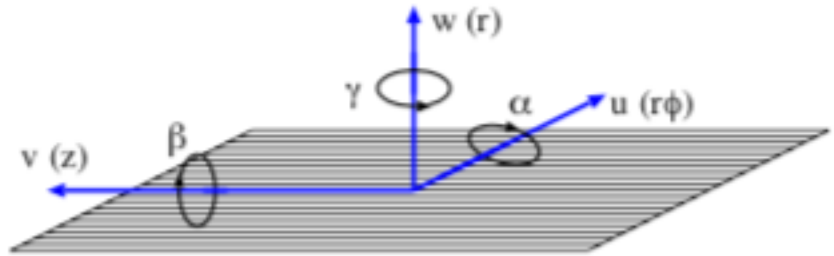
$$\mathbf{r} = \mathbf{R_s}^T\mathbf{q} + \mathbf{T_s}$$

- For individual sensors, alignment corrections are incremental rotations $\mathbf{\Delta R}$ and translations $\mathbf{\Delta q}$ which lead to

$$\mathbf{r} = \mathbf{R_s^T \Delta R_s (q + \Delta q_s) + T_s}$$

- Rotations can be reduced with respect to 3 angles. The alignment parameters become

$$a = (\Delta u \;\; \Delta v \;\; \Delta w \;\; \alpha \;\; \beta \;\; \gamma)$$



u: most sensitive direction
v: least sensitive direction
w: normal to the sensor plane

$$\zeta = \begin{pmatrix} u_r \\ v_r \end{pmatrix} = \begin{pmatrix} u_m \\ v_m \end{pmatrix} - \begin{pmatrix} u_p \\ v_p \end{pmatrix}$$

$$\frac{\partial\zeta}{\partial\mathbf{a}}\Big|_{\mathbf{a}=0} = \mathbf{P}\begin{pmatrix} -1 & 0 & \frac{du_p}{dw} & -v_r\frac{du_p}{dw} & u_r\frac{du_p}{dw} & -v_r \\ 0 & -1 & \frac{dv_p}{dw} & -v_r\frac{dv_p}{dw} & u_r\frac{dv_p}{dw} & u_r \end{pmatrix}$$

Stoye '07

Math behind composite structures alignment

- Each composite structure has an assigned local coordinate system defined by the orientation matrix $\mathbf{R_c}$ and origin $\mathbf{T_c}$
- The definitions of the composite structure alignment parameters $\mathbf{a_c}$ is the same of the sensor alignment parameters.
- The alignment relations between sub-component to composite structure is given by:

$$\mathbf{a}_c = \sum_{i=0}^{i=n} \mathbf{C}_i^{-1} \mathbf{a}_i$$

- We need to compute the C-matrices

TransC -> TransS

RotC -> TransS

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

TransC -> RotS => 0

RotC -> RotS => 0

$$\mathbf{a}_i = \mathbf{C}_i \mathbf{a}_c$$ ← relation between position/orientation corrections

$$\frac{\partial \mathbf{r}}{\partial \mathbf{a}_c} = \frac{\partial \mathbf{r}}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{a}_c} = \frac{\partial \mathbf{r}}{\partial \mathbf{a}_i} \mathbf{C}_i$$ ← relation between derivatives

Math behind composite structures alignment

TransC -> TransS

RotC -> TransS

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

TransC -> RotS => 0

RotC -> RotS => 0

$$\mathbf{C}_{11} = \mathbf{R}_s \mathbf{R}_c^T$$

lever arm

$$\mathbf{C}^1_{12} = \mathbf{R}_s(\mathbf{R}_c^T \frac{\partial \Delta \mathbf{R}}{\partial \alpha} \mathbf{R}_c(\mathbf{r}_{s0} - \mathbf{r}_{c0}))$$

$$\mathbf{C}^{21} = 0$$

$$R^{\alpha}_{22} = R_s(R_c^T \frac{\partial \Delta R}{\partial \alpha} R_c) R_s^T \tag{4}$$

The linear approximation euler angles of R^{α}_{22} gives the same column of the C_{22} matrix:

$$c_\alpha = M_{\alpha\beta} R^{\alpha}_{22} V_{\alpha\beta} + M_\gamma R^{\alpha}_{22} V_\gamma \tag{5}$$

$$M_{\alpha\beta} = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad V_{\alpha\beta} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$M_\gamma = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad V_\gamma = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Stoye's thesis

cmssw derivatives

Constrained alignment

An equality constraint is required to allow only linear combinations of the subcomponent alignment parameters, which leave the composite object alignment parameters invariant:

$$0 = \sum_{i=0}^{i=n} \mathbf{C}_i^{-1} \mathbf{a}_i \tag{5.17}$$

These constraints also change the interpretation of the subcomponent's alignment parameters. They do not represent anymore the absolute corrections, which are needed to be applied to a subcomponent. These parameters correct only the misplacement of the subcomponents on the composite structure. Composite structures can also be defined recursively. The corrections needed due to the misplacement of a composite structure can be calculated with the corresponding matrices $\mathbf{C}$. The total corrections applied to sensor i are then:

$$\mathbf{a} = \mathbf{a}_i + \mathbf{C}_{ij}\mathbf{a}_{cj} + \mathbf{C}_{jk}\mathbf{a}_{ck} + \ldots$$

where j,k, ... are the composite structures indices.

Stoye's thesis

A possible scenario of HPS Alignable structures

SLAC

- Here is reported the set of orientations R and origins T **(*)** for possible alignable structures as it is implemented in the current HPS geometry code
- **Notice:**
  - The 30.5mrad at module level in our geometry structure
  - The modules are located **far** from the sensors and from the support rings (large rot-to-trans cross terms in the C-matrices)
- An alignable structure is just a container of a Rotation and a translation
- C matrices can be computed in a recursive way.
- Tracking volume can be made alienable with identity rotation and null translation

(*) local to global is $R^T q + T$

Alignable Support Ring Top (aka SVT-front)

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \quad T = [-117.33, 56.857, 417.79]$$

UChannel46 top (aka SVT-back) - check this

$$R = \begin{bmatrix} 0.9995 & 0.0 & -0.0305 \\ 0.0305 & 0 & 0.9995 \\ 0 & -1 & 0.0 \end{bmatrix} \quad T = [14.995, 8.4230, 491.84]$$

Alignable Module Top L1

$$R = \begin{bmatrix} 0 & 1 & 0 \\ 0.9995 & 0 & -0.0304 \\ -0.0304 & 0 & -0.9995 \end{bmatrix} \quad \boxed{T = [-122.61, 59.820, 36.284]}$$

Alignable Sensor Axial L1

$$R = \begin{bmatrix} 0 & 1 & 0 \\ 0.9995 & 0 & -0.0304 \\ -0.0304 & 0 & -0.9995 \end{bmatrix} \quad \boxed{T = [1.1566, 7.8106, 38.366]}$$

Alignable Sensor Stereo L1

$$R = \begin{bmatrix} 0.0998 & 0.995 & -0.0031 \\ -0.995 & 0.0998 & 0.0303 \\ 0.0304 & 0 & 0.9995 \end{bmatrix} \quad \boxed{T = [2.1622, 7.7995, 45.934]}$$

Module to side C-Matrices examples

SLAC

$$C_{L1}^{M\rightarrow A} = \begin{bmatrix} \boxed{\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}} & \boxed{\begin{matrix} 0 & -5.8 & -123.6 \\ 5.8 & 0 & -52.0 \\ 123.6 & 52.0 & 0 \end{matrix}} \\ \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} & \boxed{\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}} \end{bmatrix}$$

module to axial side

module to stereo side

$$C_{L1T}^{M\rightarrow S} = \begin{bmatrix} 0.995 & 0.0998 & 0 & 1.34 & -13.4 & -129.0 \\ 0.0998 & -0.995 & 0 & -13.4 & -1.34 & 39.3 \\ 0 & 0 & -1 & -124.4 & -52.0 & 0 \\ 0 & 0 & 0 & 0.995 & 0.998 & 0 \\ 0 & 0 & 0 & 0.0998 & -0.995 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

- As example, the matrix for the L1 top between the module (as composition of Axial and Stereo sides) and the Axial side
- Notice for axial:
  - Module translations are the same of axial side translations (they have the same orientation)
  - Module rotations imply the same side rotation (same reason)
  - Module rotations imply large sensors translations (due to the offset in constructing the geometry discussed in previous slide)
- Notice for stereo the different orientation of the sensor local axes and the stereo angle.

How I implemented this, why I sucked in doing that and how I interfaced it to MPII

- First implementation in: cAli_dev
- Created AlignableDetectorElement class:
  - Way to pass the SurveyVolume transforms down to the Driver level, but mother-daughter is lost **(can be re-implemented by there must be a better way without duplicating information)**
- I compute the C-Matrices for each hit-on-track in the GBLRefitterDriver **(sucks because it's useless matrix multiplications for every hit. Transforms are known after geometry building)**
- The interface to MPII is very simple: just add the derivatives to the GBLPoint, form a new trajectory and call milleOut. Each mille binary entry will have 6 + 6*n derivatives where n is the number of the global structures depending on that hit.
- **I still don't compute the constrains automatically but with pen and paper.**

labels set

$$z_i = y_i - f(x_i, \boldsymbol{q}, \boldsymbol{p}) = \sum_{j=1}^{\nu} \left(\frac{\partial f}{\partial q_j}\right) \Delta q_j + \sum_{\ell \in \Omega} \left(\frac{\partial f}{\partial p_\ell}\right) \Delta p_\ell \, .$$

The dimension of the label set is arbitrary

n_{lc} = number of local parameters   array : $\left(\frac{\partial f}{\partial q_j}\right)$

$\boxed{n_{gl} = \text{number of global parameters}}$ $\boxed{\text{array} : \left(\frac{\partial f}{\partial p_\ell}\right) ; \text{ label-array } \ell}$

z = residual $(= y_i - f(x_i, \boldsymbol{q}, \boldsymbol{p}))$ σ = standard deviation of the measu

These need to get recomputed for each point and a new trajectory formed

$$c = \sum_{\ell \in \Omega} f_\ell \cdot p_\ell$$

The constraint value c is usually zero. The format is:

```
Constraint    value
label         factor
 ...
label         factor
```

$$0 = \sum_{i=0}^{i=n} \mathbf{C}_i^{-1} \mathbf{a}_i$$

MPII manual