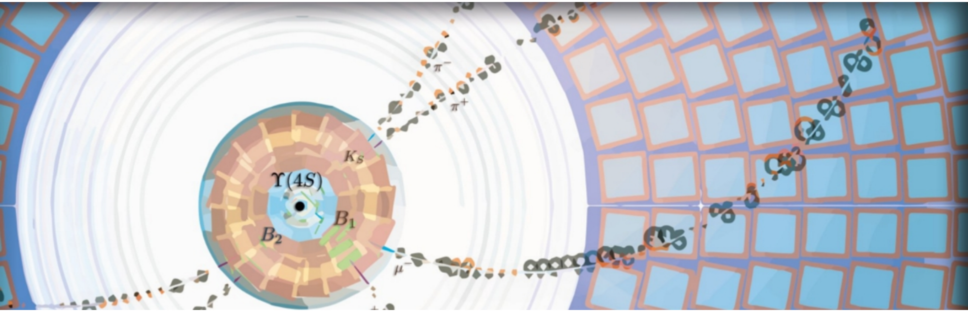
CDC cellular automaton track finding.

Various topics



Oliver Frost
Deutsches Elektronensynchrotron
2015-01-20

Overview

Changes to source code structure

Placement of tests

Place test, where you like

You can now place your `tests` folders inside subfolders of your package and they are still compiled into the toplevel test executables.

Suggestion

Keep your tests close to the implementation code like

```
/include
/src
/tests
```

Benefits

- Test do not pile them up in the package directory.
- Avoids the ‘out of sight – out of mind’ problem.

Test executables

Test executables include all tests in the folder hierarchy below.
E.g. all tests in

```
/tracking/trackFindingCDC/numerics/tests
```

end up in the following executables

```
test_tracking_trackFindingCDC_numerics
test_tracking_trackFindingCDC
test_tracking
test_all
```

such that one can select the detail of test execution.

Python packages

Structure your python code

The `scripts` folder now supports placement of Python packages in addition to simple python modules (like simulation.py).

Python packages = Folder hierarchies marked with `__init__.py` files.

Note

The `scripts` folder can be placed inside subfolders as well,
for example `tracking/trackFindingCDC/scripts`

Benefits

- Serves as a place for common python code.
- Place your analysis script or general purpose code here in a subfolder!
  - Reduce the size of your shadow repositories.
- Take advantage of rapid prototyping in python and gradually move to C++ using pyROOT

Nice application

Standard runs

- Do not write the same steering files over and over.
- Use prepared scripts and feed them with command line parameters (`--options` to view the parameters)

Prepare 1000 simulated events with EvtGen

```
python -m tracking.run.event_generation -n 1000 evtgen.root
```

Prepare 1000 simulated events with Cosmics generator

```
python -m tracking.run.event_generation -g cosmics -n 1000 cosmics.root
```

Prepare 1000 simple simulated events using the ParticleGun plus background

```
python -m tracking.run.event_generation -g simple_gun \
    -n 1000 -b /my/bkg/folder check_bkg.root
```

Run the primitive cdc xy-display on a file

```
python -m trackfindingcdc.run.display -i check_bkg.root
```

Increasing demand for python

Push to include more python libraries into the framework?

- Numpy - efficient storing and access to homogeneous data (already included)
- IPython (notebook) - interactive / graphical session over http
- Matplotlib - for plotting
- Scipy - statistics, numerical optimization
- Scikit learn - multivariate analysis
- Pandas - multivariate data handling

Note

You may easily install these libraries into the basf2 python installation with `pip`.
After sourcing the tools, install additional libraries like
`pip install matplotlib`
even if you are not root on your machine.

Validation

Presented plans : Extend general validation

Rational

I am looking at a lot of control plots like

- distributions of residuals
- pulls
- scatter plots
- at various stages

which should to be checked regularly.

Questions

- How do we provide them to the framework in a structured manner?
- Do you want to see plots on the validation framework showing bad performance?

General approach

Design considerations

- Validation is seen as an **error detection** tool as opposed to an **investigative** tool.
  - Limited data to a couple of thousands of events
  - Performance of the validation is negligible as it consumes little time in comparison to the finders/fitters.
- Setting up a BASF2 module from Python is quick, easy and fun.
- Very tight refactor $\leftrightarrow$ run feedback loop.
- Display key plots immediately to validate changes to the tested code.
- Let it run unchanged on the build server.
- Run on pre-generated events or generate them on the fly.

Technical details

- Use numpy for data handling and vectorised function application.
- pyROOT exposes everything to read and write back to ROOT files

Implementation units in Python

- Plot
  - Encapsulates ROOT histogram.
  - Only allows plots that are useful on the validation framework.
  - Access special attributes contact, description and check easily.
- Plot bundles
  - Group of plots that belong together
  - E.g. residuals, pulls, p-values
- Validation module
  - Pull data from DataStore and resolve relations into flat numpy.arrays
  - Produce ROOT plots on terminate, which are compatible for display on the validation page.
- Validation runs
  - Command line interface to feed input files, generator setup, finder and fitter options.
  - Specialisation to validation runs on the build servers.

Plot

Standardized ValidationPlot

- Abstracted view to a ROOT histogram with pythonic interface (inspired by matplotlib)
- Inputs are generally numpy.arrays, but TBranches are possible with a little work, if desired.
- Can handle
  - Histograms
  - Profiles
  - Scatterplots
  - $+$ standard fits to gaus, constant, diagonal, linear, (cauchy?)
  - $+$ **counts and displays non-finite values (*NaN* and $\pm$*inf*)**
  - $+$ manuell or automatic binning and outlier detection with robust (trimmed) estimators.
  - easy setting of special attributes contact, description, check, title, xlabel, . . .

Bundles of plots

Standardized PullAnalysis

Compares a single estimated quantity (with variance) to its truth with the following plots

- Distribution of truths
- Distribution of estimates
- Truth versus estimate profile + diagonal fit
- Truth versus estimate scatter plot
- Residuals
- Distribution of sigmas
- Pull + gaus fit
- P-Values + constant fit

Note

`residuals = truths - estimates` (+1) for Eugenio.

Basf2 validation module

ValidationModule

- Pulls data for each
  - event
  - Monte Carlo track
  - pattern recognition track

and stores them in working memory numpy.arrays

- Generate and write the plots to ROOT output
- Soon: optional write out of numpy.arrays into trees to file as well.

Plot organisation

- TDirectory subfolders like “expert” seem appropriate.
- Timothy is working on that subfolders can be shown on the validation page.

Validation run

ValidationModule

- Exhibit parameters
  - Number of events
  - Generator
  - Finder
  - Fitting parameters
  - . . .
- Makes parameters overridable from Python subclasses or command line.
- Existing specialisation
  - Cosmics runs
  - VXDTF
  - CDCLegendre
  - CDCAutomaton

Validate 1000 simulated events with Cosmics generator show results immediately

```
python -m tracking.validation.run -g cosmics -n 1000 -s
```

Cosmics finding

Status

Current performance is reasonably high.

Efficiency	0.9998
Hit efficiency	0.7954
Clone rate	0.2345
Fake rate	0.0016 (includes ghosts and background)

> Run the cellular automaton track finder as `TrackFinderCDCCosmicsModule`

Work to be done

- Forward the covariance matrix of the fast fits to Genfit.
  - Involves translation of the perigee covariance to Cartesian coordinates
- Adjust the track orientation from inside-out to top-down for cosmics.
- Enable a purity measurement with multiple cosmic tracks in one event.
  - Adjustment of the Comics generators is needed here
  - Abolish the `cosmicsHelix` on the same go.

Helix

Helix implementation

- Implementation in framework/dataobjects/src/Helix.cc
- Designed as an ideal geometrical object
- Perigee parametrisation is d_0, ϕ_0, ω, z_0, $\tan\lambda$ in that order.
- Perigee point is always the closest approach to the z axis.
- Points on the helix are addressed with the circle **arc length** s

$$h_{xy}(s) = \begin{pmatrix} \cos\phi_0 & -\sin\phi_0 \\ \sin\phi_0 & \cos\phi_0 \end{pmatrix} \cdot \begin{pmatrix} -\frac{\sin\chi}{\omega} \\ -\frac{1-\cos\chi}{\omega} - d_0 \end{pmatrix} \quad \text{where} \quad \chi = -s \cdot \omega$$

$$h_z(s) = \tan\lambda \cdot s + z_0$$

- To extract momenta the magnetic field has to be provided as a function parameter.
- Provides basic extrapolations to **cylinders** and **axial wires**.
  - Port HelixHelper class for more extrapolations?
- Features no uncertainty matrix yet.
  - Plan is to subclass it as UncertainHelix.

Adjust the TrackFitResult

Refactor to use UncertainHelix as member

- Removed duplicated code.
- Block some of the unwanted methods of the helix.

Discussion

Should the TrackFitResult contain a start point different from the origin?

Benefits

- Capture the start point of cosmics correctly ?
- Capture secondary particles, hard scattering ?

Merging the two track finders

Merge the two track finders

Rational

- Many things are implemented twice like
  - Hit objects
  - Track objects
  - Fits
  - Trajectory representations
  - Reconstruction of the z coordinate
  - A basic event display
  - Sorting of hits in a track
  - Merging of tracklets
- Combining the efforts may lead to synergies.
  - Exchange of ideas, experiences and problems
  - Mutual review of code

Kick off in Karlsruhe

Strategy

1. Synchronise the basic data objects such as hits and tracks.
2. Clean and generalise algorithms to work on common data structures.

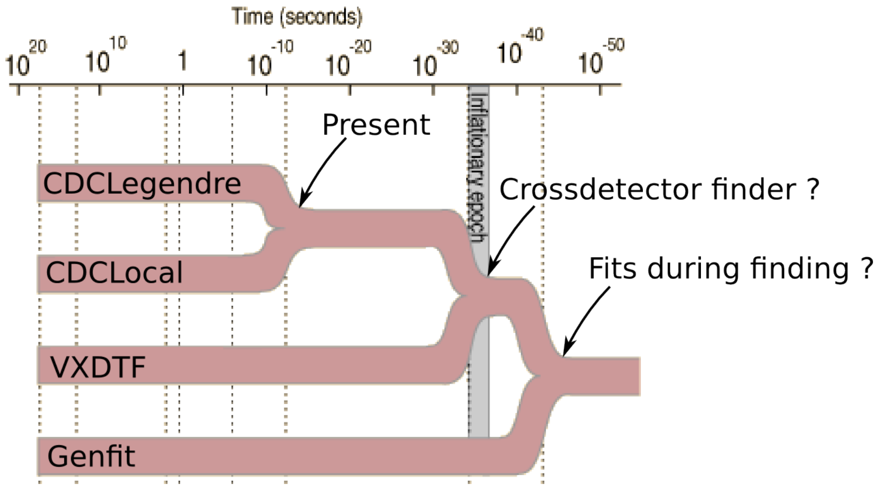
Use cases - brainstorming

- Easy transfer of rich data between the track finders, not involving the DataStore.
- Common merging approach to 2D-tracks (segments) and 3D-tracks
- Change execution order:
    1. Clean event from background hits using cellular automaton before Legendre run.
    2. Run Legendre part for the high momentum tracks
    3. Mark hits as use
    4. Run cellular automaton on the rest.

Changes so far

- Adopted long agreed naming scheme
- Namespace `TrackFindingCDC`, finder modules start with `TrackFinderCDC`
- Start of gradual refactoring

GUT - Grand unified tracking



Time (seconds)
10^{20}
10^{10}
1
10^{-10}
10^{-20}
10^{-30}
10^{-40}
10^{-50}
Inflationary epoch
Present
Crossdetector finder ?
Fits during finding ?
CDCLegendre
CDCLocal
VXDTF
Genfit