Design Notes on Van der Meer Scan Analysis Framework

February 17, 2014

Contents

1 Motivation & Goals

We seek to develop a user-friendly software framework for the analysis of Van der Meer (VdM) scans. Overarching goals include reducing the time between when scan data are acquired and when calibrations are available, making it easy for new members of the BRIL group to participate in scan analysis work, and facilitating standardization of VdM analyses and error estimates.

Specific design goals for the system include:

1. The system should be simple and easy to use so that individuals new to VdM analysis can get up to speed quickly.
2. The system should encapsulate the knowledge gained in previous scan campaigns.
3. The system should be modular, making it easy to incorporate alternate inputs—e.g., data from different luminometers—and to implement different analysis schemes—e.g., different fitting functions.
4. The system should shield the user as much as possible from having to hand-collect information and corrections necessary for the analysis.
5. The code should be easily readable and should avoid complicated structures. It should be easy to modify.

6. The system should allow one to capture and archive a particular analysis, including all of the input data and analysis algorithms (analysis code).
7. Comparing results from different analyses should be simple. It should be straightforward to understand the differences between the analyses.
8. It should be possible to download all needed code and files to a laptop for analysis offline.
9. The system should be centrally maintained using standard version management methods.

2 Language and File Format

The main language will be `python-2`, but the code should be fully compatible with `python-3`. Lines will be added to the code to ensure eventual python-3 compatibility.

ROOT is used from python via its interface module `PyRoot`. Data will be stored in the form of root files for visualization and plotting and pickle files to have a quick access to the data. Other formats (CSV, plain text) could be produced from pickle files for a convenience of the user.

3 Structure

3.1 Overall

The framework consists of a collection of scripts, some of which are used to prepare the various input files and other of which are used to carry out the actual analysis. As a general rule, only the latter group of files are archived in connection with individual analyses. A block diagram of the process is shown in Fig. **??**.

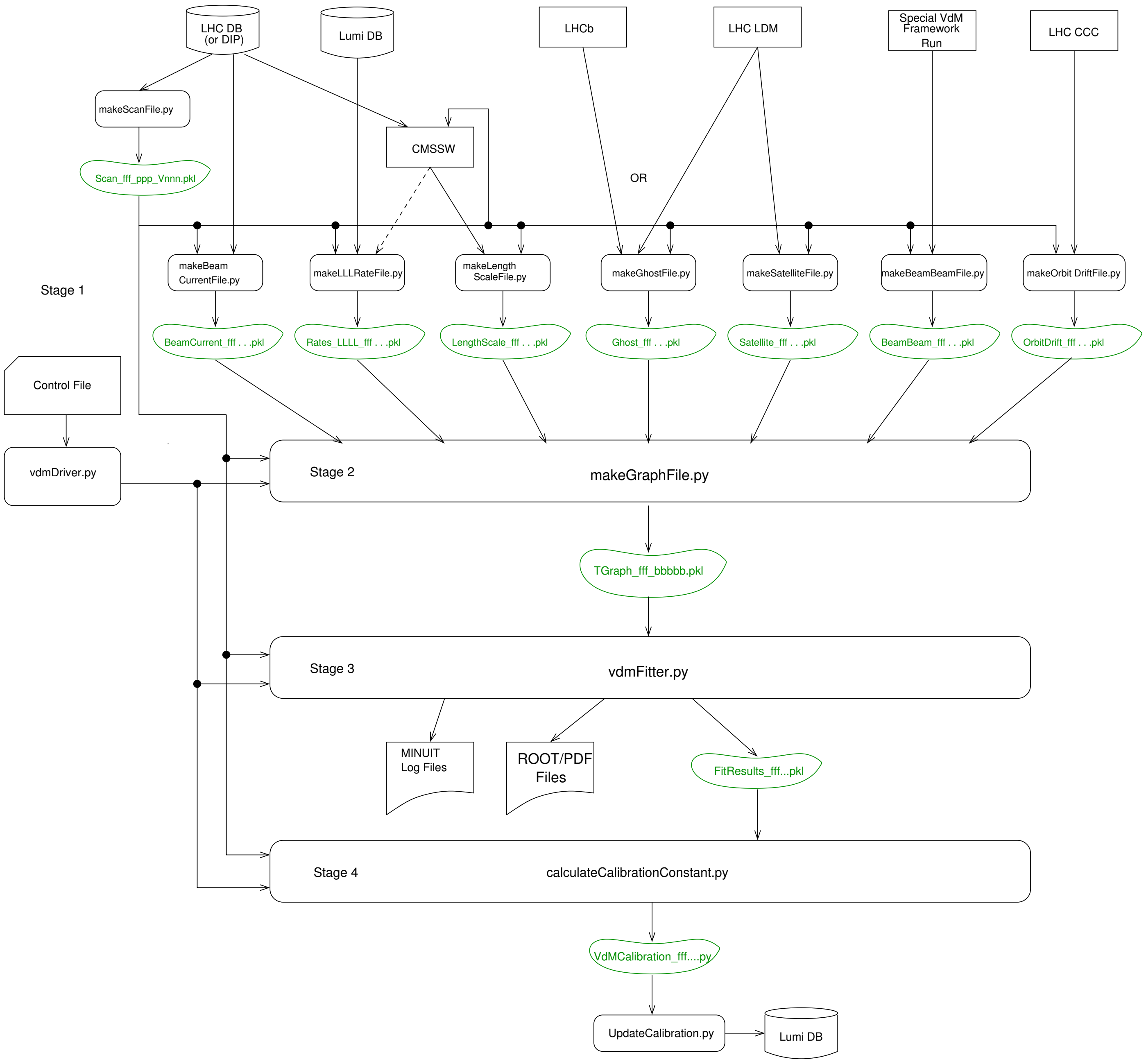

Figure 1: Overall structure of the VdM analysis framework.

3.2 Stages

An analysis is run in four stages (or layers):

Stage 1 Data preparation. This layer comprises a number of modules that will read information from various sources (the Lumi database, the LHC database(s), the output of CMSSW jobs, etc.) and convert them to standardized pickle files that comprise the input to the analysis layer. Most of these procedures will be implemented using standard scripts that will be officially supported as part of the framework.

Stage 2 Graph production. In this layer, the standardized input files from Stage 1 are used to produce ROOT TGraphErrors objects, which serve as input to the fitting stage. Various corrections to the input data from the luminometers are applied. The main script here is called `makeGraphFile.py`.

Stage 3 Fitting. This stage takes input from the previous stage(s) and uses it to perform fits to the VdM scan data. Results are stored in the form of ROOT files and `python` pickle files. The main script here is called `vdmFitter.py`.

Stage 4 Calibration constant calculation. This stage takes the results from the fits and uses them to arrive at an overall calibration constant for the luminometer.

3.3 General File Structure of an Analysis and Archival

An analysis consists of the data and results from the stages of Section **??**. Specifically it includes:

1. input files in pickle format that were produced by the Stage 1 data preparation layer
2. the code modules used to analyze the input files.
3. a control (or driver) file that specifies the input files and the code modules.
4. output files in pickle format that encapsulate the results

All files are stored in a central repository using standardized names (see Section **??** below). The driver file is enough to fully specify the analysis. The archived files will in general correspond to the fully corrected result. During the course of the analysis, however, corrections will be removed one-by-one so as to produce a results table that gives the size of each of the corrections that are applied.

4 Implementation Details

4.1 Input Data Files (Stage 1 Output)

The files described in this subsection are the outputs of Stage 1.

4.1.1 Scan File

Source The script that produces the pickle file is called `makeScanFile.py`. We anticipate that this will be done using calls to the LHC database, although the use of DIP files produced during the scan is another possibility.

Naming These files have a name of the form: `scan_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents This file contains basic information about a scan, including:

- A list of scans in time order—e.g., `[X1, Y1, X2, Y2, LSX1, LSY2]`.
- machine parameters: center-of-mass energy, beam 1 particle, beam 2 particle, β^*, crossing angle (Are there others?)
- list of nominally filled bunches
- list of nominally colliding bunches
- list of fill(s)
- list of run(s)
- scan point data table, where the format of a row is:
  - Scan#: equal to "1" for the first scan in the fill, "2" for the next, etc.
  - Scan Type: is x, y, or LS (length scale), etc.
  - Scan Point#: equal to "1" for the first point in the scan, "2" for the next, etc.

- t_{start} = time stamp in UTC. In python this will take the form of a `datetime.datetime` object, which has a resolution of μs.
- t_{stop} = time stamp in UTC. Same as above.
- x_1 = nominal horizontal offset of first beam. Units: μm
- y_1 = nominal vertical offset of first beam. Units: μm
- x_2 = nominal horizontal offset of second beam. Units: μm
- y_2 =nominal vertical offset of second beam. Units: μm

The precise form of the objects stored in the scan file is defined in the `python` code that reads and writes the module. Note that since many other scripts in the VdM framework depend on time-stamp information, the scan file plays a special role and must be prepared before many other parts of the analysis can proceed.

4.1.2 Beam Current File

Source The script that produces the pickle file is called `makeBeamCurrentFile.py`

Naming These files have a name of the form: `BeamCurrent_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents This file contains the beam-current values for each scan point. All currents are given as the number of protons. For heavy ions the reading is still in terms of the number of unit charges circulating—e.g., for lead, a reading of 8.2×10^7 means that the bunch contains 10^6 ions.

- Scan#: equal to "1" for the first scan in the fill, "2" for the next, etc.
- Scan Point#: equal to "1" for the first point in the scan, "2" for the next, etc.
- DCCT value for beam 1
- DCCT value for beam 2
- Raw FBCT values for beam 1 summed over all active bunches.

- Raw FBCT values for beam 2 summed over all active bunches.
- `python` dictionary where the key is the BCID and the the value is the beam 1 current for that BCID.
- `python` dictionary where the key is the BCID and the the value is the beam 2 current for that BCID.

The bunch currents in the last two items are adjusted such that the sum over active bunches equals the DCCT value. They may therefore be somewhat different from the raw FBCT values. The size of the difference can be gauged by comparing the DDCT values to the sums of the raw FBCT values. Ghost and satellite corrections are *not* applied at this stage.

In general, the time resolution of the beam current measurements will be smaller than the length of a scan point. In such cases, the value to be used will be the average of the beam current measurements falling between t_{start} and t_{stop} for the points.

4.1.3 Luminometer Data File

Source Any luminometer. In general making this file is the responsibility of the particular luminometer group. The framework will include a set of scripts used to create luminometer data files for the standard cases. There should be a companion CSV format for users who would rather not write pickle files directly.

Name These files have a name of the form: `Rates_llll_ffff_ppp_Vnnn.pkl`, where `llll` is the name of the luminometer (HF, PLT, BCM1F, PCC, etc.), `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents This file contains the luminosity values for each scan point.

- Scan#: equal to "1" for the first scan in the fill, "2" for the next, etc.
- Scan Point#: equal to "1" for the first point in the scan, "2" for the next, etc.
- `python` dictionary of luminosity values where the key is the BCID and the the value is a doublet with the luminosity and error for that BCID.

4.1.4 Length Scale File

Source This file is created in a separate analysis of zero-bias data acquired during special length-scale scans.

Naming These files have a name of the form: `LengthScale_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = “BRIL” for centrally produced files), and `nnn` is a version number.[1] A `.csv` file format is also implemented, along with scripts to convert between the pickle and `.csv` formats.

Contents This file contains length-scale correction information about a scan. In particular,

- k_{x1} = beam 1 horizontal scale correction.
- k_{y_1} = beam 1 vertical scale correction.
- k_{x2} = beam 2 horizontal scale correction.
- k_{y_2} = beam 2 vertical scale correction.

4.1.5 Ghost File

Source: This information is obtained either from LHCb or the LHC LDM group. In 2012/13 they provided python code and pickle files (LHCb) or csv files (LDM). The raw data in their original format, plus any documentation provided, will be kept in a dedicated directory.

The script that produces the pickle file is called `makeGhostFile.py`

Naming These files have a name of the form: `Ghost_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = “BRIL” for centrally produced files), and `nnn` is a version number.

Contents Contents to be defined.

[1]For “null corrections”, special files having `ffff`=0 and `nnn`=0 will be available.

4.1.6 Satellite File

Source: This information is obtained from the LHC LDM group. In 2012/13 they provided csv files. The raw data in their original format, plus any documentation provided, will be kept in a dedicated directory.

The script that produces the pickle file is called `makeSatelliteFile.py`

Naming These files have a name of the form: `Satellite_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents Contents to be defined.

4.1.7 Beam-Beam File

Source: Source is a python script that, given number of particles per beam, Σ values and uncorrected x and y coordinates, calculates a correction Δx and Δy. This implies that one has to run a fit on the uncorrected scan data and use the Σ values obtained from that fit as input data for the script. Hence fitting in stage 3 (see Section **??**) has to be done in two steps.

The script that produces the pickle file is called `makeBeamBeamFile.py`

Naming: These files have a name of the form: `BeamBeam_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents Contents to be defined.

4.1.8 Orbit Drift File

Source: The information for this correction must be obtained manually from the CCC. The effect is typically very small, so for now a place-holder null correction will be used. The script that produces the pickle file is called `makeOrbitDriftFile.py`

Naming These files have a name of the form: `OrbitDrift_ffff_ppp_Vnnn.pkl`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

Contents Contents to be defined.

4.2 Stage 2 Output File

This stage uses a script called `makeGraphFile.py` to produce `python` pickle files containing ROOT `TGraphError` objects. There is one `TGraphError` object for each scan-BCID combination. The `TGraphError` objects contain all of the corrections used for the particular run of the analysis, as determined by the `vdmDriver.py` file.

4.3 Stage 3 Output Files

These files are the output from Stages 2 & 3, which should produce two output files: a `python` pickle file containing a summary of the results, and the ROOT files containing the TGraphError objects described in Section **??**.

4.3.1 Stage 3 Output Files

These files, which are produced by the python script `vdmFitter.py`, have a name of the form: `FitResults_ffff_ccc_Vnnn.pkl`, where `ffff` is the fill number, `ccc` is bit pattern, expressed in hex, that indicates which corrections have been applied for this particular run of the fitter, and `nnn` is a version number.

The contents of the results file include:

- a string that specifies the type of fit
- a `python` dictionary, where the key is a concatenation of the scan-number and the BCID—e.g. 2X:1671—and the value is a tuple of values consisting of the values listed below. For each result value, there will be an associated statistical error (except for the fit quality variables).
  - Fully corrected calibration constant (cross section) for this scan

- Calibration constant after all corrections except for length scale.
- Calibration constant after all corrections except for ghosts.
- Calibration constant after all corrections except for satellites.
- Calibration constant after all corrections except for beam-beam.
- Calibration constant after all corrections except for dynamic-β.
- Uncorrected Σ ("cap sigma").
- Uncorrected μ_1[2]
- Uncorrected μ_2
- Uncorrected σ_1
- Uncorrected σ_2
- Uncorrected r parameter
- Uncorrected B_1 (first background)
- Uncorrected B_2 (second background)
- Uncorrected peak value
- Uncorrected area
- Current-product at peak
- Boolean indicating failed/successful fit
- χ^2 for fit
- $ndof$ for fit
- p-value for fit

Contents to be defined in code. Discussion needed here.

4.3.2 Plots File

These ROOT files have a name of the form: `Plots_ffff_ppp_Vnnn.pkl`, OD: `Plots_ffff_ppp_Vnnn.root`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a version number.

For each of the scan-number BCID pairs listed above there will be a TGraphErrors, which shows the data plotted with the fit overlaid.

[2]The variables μ_1, μ_2, σ_1, σ_2, r, B_1, and B_2 having meanings that depend on the nature of the fit model.

4.3.3 MINUIT log file

MG The minuit output in simple text format should be kept in a file with a name of the form: `Log_ffff_ppp_Vnnn.txt`, where `ffff` is the fill number, `ppp` is the nickname of the person creating the file (`ppp` = "BRIL" for centrally produced files), and `nnn` is a sequence number—e.g., "001" for the first instance.

OD: It is not possible to intercept TMinuit neither in root nor in python. One can do it, however, from the command line in bash and redirect the analysis output into a file.

4.4 Driver File

There is a lot of room for discussion here. I provide a specific model as an "existence proof." Other models are possible and should be considered.

One simple model for the driver file is to use a python script, which creates a `python` dictionary of file names that is passed to the main VdM analysis script. Typically users would start with a standard template script and modify it to suit the particular analysis. The template script would be generated by another python script that is part of the framework and takes various parameters, such as the fill number, luminometer, and fit type as inputs. The driver file thus generated would invoke fitting routines, with a standard configuration of parameters for the fits. Users wishing to change the way the parameters are configured—e.g., a user might want to fix a parameter that is normally free—would edit the code.

OD: It is handy to have a prepared config file either in python or in txt, rather then generate it. (See comments above in Sc. 4.1.3).

4.5 Analysis Script

Each analysis would start with a standard script that would fit data according to a user specified fit model (single Gaussian, double Gaussian, super Gaussian, etc.). OD: all fit models should be run in one shot. There will be two different classes, which will be run by two scripts: 1D models and 2D models.

The standard scripts would include standard prescriptions for determining starting values for parameters and for which parameters should be fixed. Users would be able to edit these scripts to customize them to their particular situation.

4.6 Post Analysis

4.7 Appendix

- Presently, HLXs work asynchronomous with LHC (with precision about 1 lumi section). That is why we recover start and stop of the scan by timestamp. If we introduce a scan point, will we be able to make a one-to-one correspondence between rates and beam parameters?

- Presently, we use the information only from one the most external ring (according to the rawlumi.py file). There is also information about the second ring. There is also info about HF+ and HF- separately. Question: do we want to store in addition to information per ring, per HF side, some useful combinations of it (i.e. Ring 1 HF+ plus HF- info)?

- Statistical error computation. In rawlumi.py the error is treated as $mathlog(below/active)$, where below are non-firead cells below the threshold and active are all cells (!) I.e. active is a constant. So we treat the error as $\frac{1}{below} - \frac{1}{active}$, considering that active has statistical fluctuations. So the statistical error is underestimated.