**Project Log**

**Note:** I have not proofread these notes.

**January 17th**

Today, the general aim of the project and the intended procedure was gone over again to ensure that everything was clear. I was also given access to a sample ROOT tree so that I may start working.

I also went through some of the old ROOT tutorials from MNXB01 to refresh my memory and started watching a YouTube tutorial (<https://www.youtube.com/watch?v=KPz-dNjdx40&list=PLLybgCU6QCGWLdDO4ZDaB0kLrO3maeYAe&index=1>).

**January 19th**

I have now gone through the information on the Canvas page for ‘Writing in English…’. Furthermore, I have also finished watching the first 18 videos in the YouTube tutorial to get reacquainted with ROOT. I also got some reintroduction to ROOT through reading first the three chapters in the primer at ROOT’s website. The main reason for this was chapter 3 of that primer, which went over important information about how to use ROOT in conjugation with g++, which I have had some issues with on previous attempts. Although I successfully got it to work with g++, I still need to get it to work with Make. Make seems somewhat peculiar in that most of the Bash syntax works, but sometimes it does not play nice with text substitution with backticks ‘`’, and sometimes it does.

My reason for spending this time on getting ROOT to work with g++ and Make is that this project will likely be much larger than programming projects I have worked on before. Hence, I want to be able to use encapsulation with object files instead of just having one giant ROOT macro or stringing multiple ROOT macros together in some chain that needs to be memorised.

**January 25th**

Since last time I have worked on a couple of things. The first I worked on was the getting ROOT to work with g++ and the Make-file, which turned out to be a lot more complicated than I had anticipated. It turned out that using the normal ROOT-packages was quite simple, and only a matter of running a command to get the necessary flags for ROOT to work. However, to be able to use custom classes (which I need to do since custom classes were used in the ROOT-tree I was going to read) needed a lot more of an involved process. Information on the internet was scarce (and a lot of it was outdated and no longer correct), but I eventually managed to find the correct instructions on <https://root.cern/manual/io_custom_classes/>. (While it might seem trivial to check *the* manual, ROOT seems to have several different manuals and several different primers so finding the correct one was a non trivial matter, especially since google works quite poorly since most results with the keyword ‘root’ a referring to the top of the Linux directory (windows equivalent of C:\). It was also quite hard to find the error since the error messages were from the linker and these were quite uninformative. The solution turned out to be that I needed to create a shared library through a so called ‘dictionary’ (which the ROOT console would normally create automatically if running the interpreter) as well as a file called ‘LinkDef.h’ which contained so called “pragma”-pre-processor statements. I think all of these extra complications are due to that the ROOT-compatible classes need to inherit from TObject (and also have the ClassDef() and ClassImp() statements).

After I finished this, I started working on reading the root tree. This also presented some challenges as I was unfamiliar with how exactly the data was stored. I had assumed that everything was just stored as separate leaves on separate branches, with which tracks belonging to which event being kept track of by a separate leaf (this is also what TBrowser lead me to believe). However, it turned out that the data-format for two of the three branches was a TCloneArray containing the custom classes I received. This of course makes a lot of sense in hind-sight since each branch could then have one set of leaves per event, but I didn’t think of that at the time. One thing I am really missing from python is the type() function to quickly check what type of object a function is. I have tested some methods listed online before, but I did not get those to work then. If I have more issues debugging things I will probably attempt to get that to work. While on the topic of debugging, another thing that I might have to consider is to try to get the debugger in the integrated development environment (IDE) VScode to work while using Makefile, although I do not know if it would play nicely with me use the shared object files for ROOT.

After finally getting the classes to work, I tested out reading in some of the data. I’ve included two screenshots below (it took a lot of time to generate them so I didn’t regenerate them again only to add in labels). The first one shows the number of counts of a specific azimuthal angle difference between an event in the time projection chamber (TPC) and forward multiplicity detector (FMD) and the second plot (in red) shows the derivative of said plot. If we a cylindrically symmetric TPC and the FMD:s at the end of the beamlines, the average azimuthal angle detected in the FMD:s would correspond do the being parallel to the beamline while those in the TPC would be perpendicular to the beamline. This then explains why the most common difference in azimuthal angle is close to .

Chart, line chart

Description automatically generatedGraphical user interface, application

Description automatically generated

One thing that was obvious when producing these figures was that reading in data took a lot of time. Since this was only around 70 MB, and some of the files were over a gigabyte large, I think I will need to work with optimising my code. One thing I intend to do is to create a class for reading in data and storing it in histograms (which I may then store inside of a .root-file) so that the file size is reduced. Considering that the counts are in the order of magnitude of a million, I could probably get the data size down by a lot if a million stored -values can be stored as a just a count and a bin number.