**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 Information on the internet was scarce (and a lot of it was outdated and no longer correct), but I eventually managed to find the correct instructions on 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.

**March 28th**

Since I have started working on the project again, I have mainly been working on reading in the data properly. While I was working on getting back into the project again, I managed to find some bugs which had caused the program to give erroneous results. Since then, I have been concerned with have to efficiently read in the data. First, I attempted to implement the C++ standard library functionality of #include <threads> to see if I could split up the reading of an individual file and parallelise that. However, these efforts were in vein by what is probably related to ROOT. I tried to get it to work in various ways, i.e by looking into so called ‘mutex locks’, but ROOT has several problems. The trees that I read in seem to have their copy constructor set to delete, and opening the same file twice did not seem to be possible. Furthermore, although I tried making multiple copies the file I was testing the program on to trick ROOT into thinking that I was reading different files, this was also unsuccessful. I believe this is due to that ROOT is throwing around a lot of raw pointers which causes the separate threads to interfere with each other and break through the respective scopes. Upon further research, that is to say googling, this seems to be a common problem and the solution is to use ROOT’s own implementation of multithreading. However, I decided to not do this due to two reasons. The primary reason was that a much simpler solution (which I then implemented) was to write the program to read in only one file at the time and then launch multiple copies of this program for different files. Since these were entirely separate programs executed in the terminal, ROOT’s pointers could not cause more issues this way. Secondly, there were some comments about needing to compile ROOT with a separate flag to use its multithreading functionality, I decided that trying to compile ROOT myself on every machine I was going to use would be a waste of time.

As for this week, I have been working on getting the read-in to work more fluently. I have implemented functionality such that I can simply give a file path as input to the program from the terminal so that I do not need a separate compilation process every time I want to change the directory. I have also separated out the plotting process to another file so that I can simply replot everything instead of having to re-run everything. I have also implemented functionality to just open a pre-processed file and just load in the histogram from it in case I want to access it later without having to re-run the entire program.

I have also implemented functionality to separate the data more precisely in the way that was originally planned. What I mean with this is that my implementation of the code just correlated TPC-tracks with the FMD-tracks. The new version separates out correlations between the TPC-tracks and the separate back and forward FMD:s’ tracks as well as also saving the correlation between the two FMD:s. Furthermore, I have also implemented some removal features for η-data (I forgot to mention that I also turned the histograms into 2D histograms with both Δη and Δφ data for the counts) at the edges of the detectors which I was told was bad due to decreasing resolution. Since I do not know the details of what I am supposed to cut away on the FMD which consists of two smaller FMD:s, I will have to talk to my supervisor about that during our next meeting. Finally, I have also implemented cutting functionality for the different cut quality markers which are stored in the data.

**April 28th**

Since the last entry in this log, I have mostly been concerned with getting the read in of the data to work as wanted. The code has grown sort of ‘organically’, with more and more features being implemented over time. The full event mixing, and all of the various cuts has been implemented. Furthermore, it is now possible to tell the program what regions of centrality and transverse momentum (pT) which should be saved. This helps in saving a lot of time since the program does not have to be rerun every time for every different category. The cost of all of this is that the program is now quite involved and complicated, with the source file being over 1000 lines of code, roughly half of which is dedicated to the constructor. I have also implemented plotting for debugging during this time in combine.cpp which also took some time to get working. Another thing that has changed is that I decided upon implementing multithreading anyways. Since ROOT is very complicated, this is done by launching multiple instance of the same programming with a bash script feeding in what range of events the program should read over. Properly implementing this, along with features like reading the smallest file sizes first took both a lot of time and several iterations. For example, the bash script originally run i.e 10 files at 1x speed on 10 cores, but was later changed to run 1 file at i.e 10x speed on 10 cores.

While the code is running (which is very time consuming) I have also started working on the fitting algorithm with some of the finished data sets, and the order of magnitude of the acquired v2 seems correct, but I still need to check with my supervisor so that there isn’t anything which has gone wrong with my background data.

Additionally, I have also written same pages in the Bachelor’s thesis and made some Figures since there has been a lot of time just waiting for the code to finish. The fact that large amounts of data is necessary to actually see the signal means that I have not been able to short cut the debugging to see if things are obviously wrong and have had to spend quite some time on it.

Another thing I spent quite some time on was seeing if I could manage to do the event mixing analytically. Unfortunately, this was to no avail. While getting the combinatorial background to seemingly be predicted theoretically, additional considerations after finishing that implementation made me realise that I would not have enough information to be able to deduce the detector efficiencies through other means than event mixing (I had originally thought that I could just average over unrelated events (which is a single loop instead of a nested loop so n vs n^2 in time) and theoretically expect a flat distribution, but I did not keep in mind that since I do not know the spatial dimension of the detector to do this).