UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE

"ACCELERATION OF THE KALMAN FILTER ALGORITHM FOR THE CLAS12 EVENT RECONSTRUCTION SOFTWARE"

BRUNO BENKEL

UNDERGRADUATE THESIS TO OPT FOR THE TITLE OF
CIVIL INFORMATICS ENGINEER

Advisor: Claudio Torres, Ph.D.
Correferent Professor: Hayk Hakobyan, Ph.D.
Correferent Professor: Raquel Pezoa, Ph.D.

September - 2019

DEDICATION

Dedicated to professor Milan Derpich, for teaching me the real value of knowledge.

ACKNOWLEDGMENTS

I thank professor Claudio Torres for helping me through the development of this document and the project evidenced in it, and professors Hayk Hakobyan and Raquel Pezoa for assisting me during its conclusion. I also thank professors Javier Cañas, Horst Von Brand, Silvio Olate, Chiu Hui Sun Lin and Moisés Cañas for the knowledge they passed onto me through my academic years.

I thank the Jefferson Laboratories institution for allowing me to work with them, and in particular Veronique Ziegler and Vardan Gyurjyan for personally assisting me during the development of this project.

I thank my mother Isabel Balbontin and my father Walter Benkel for doing everything in their power to ensure I had the best possible education up until and including my university years, while also thanking my two sisters Paula and Natalia for their love and support. In the same way, I thank my grandparents "Opa" Klaus Benkel and "Oma" Hedi Opitz for providing me with excellent role models in both a professional and personal sense.

I deeply thank my girlfriend Eli Stensberg without whose love and care I wouldn't have been able to finish this project and remain sane in the process, and I also take the opportunity to apologize for my constant cranky mood during it. I also thank the other students with whom I shared the pain and glory of higher education, with the special mention of Francisco Casas, who personally helped me through the development of this project and even more so during its conclusion.

ABSTRACT

Abstract— Particle accelerators and detectors are one of the main sources of data for High Energy Physics studies and analysis in general, and thus are tools that require work and maintenance to keep running in optimal conditions. In light of this, special care must be put into the software dedicated to the reconstruction of the detected events, and thus its optimization is essential to keep the production chain running efficiently.
This document presents this exact optimization, which is done by focusing on different elements associated with the components that take the most time in the offline reconstruction software of the CLAS12 detector.
The results obtained in the project are highly favorable, with a reduction of half the running time of the entire software. These results are relevant because they allow for a much faster reconstruction of the available data, and thus accelerating the whole production chain. A favorable part of the optimizations applied is that they can be useful for other institutions working with particle detectors.

Keywords— Particle detectors; Jefferson Laboratories; CLAS12; Drift chambers; Event reconstruction software.

RESUMEN

Resumen— Los aceleradores y detectores de partículas son una de las principales fuentes de data para el estudio y análisis de la física de partículas de alta energía, y por tanto son herramientas que requieren trabajo y mantenimiento para correr en condiciones óptimas. En vista de esto, especial atención se debe poner en el *software* dedicado a la reconstrucción de los eventos detectados, y por tanto su optimización es esencial para mantener la línea de producción corriendo eficientemente.
Este documento presenta esta optimización, la cual se realiza enfocándose en los distintos elementos asociados a los componentes que más tiempo toman en el *software* de reconstrucción *offline* del detector CLAS12.
Los resultados obtenidos en el proyecto son altamente favorables, viendo una reducción de la mitad del tiempo de la totalidad del *software*. Estos resultados son relevantes porque permiten una reconstrucción mucho más rápidos de los datos disponibles, y por tanto aceleran la línea de producción en su totalidad. Una parte favorable de las optimizaciones aplicadas es que pueden ser útiles para otras instituciones que trabajan con detectores de partículas.

Palabras Clave— Detectores de partículas; Jefferson Laboratories; CLAS12; *Drift Chambers*; *Software* de reconstrucción de eventos.

GLOSSARY

AB4: Fourth Order Adams-Bashforth method
ABM4: Fourth Order Adams-Bashforth-Moulton method
AM4: Fourth Order Adams-Moulton method
ASIC: Application-Specific Integrated Circuit
BMT: Barrel Micromegas Tracker
CEBAF: Continuous Electron beam Accelerator Facility
CERN: European Council for Nuclear Research (*Counseil Européen pour la Recherche Nuclaire*)
CLARA: CLAs12 Reconstruction and Analysis framework
CLAS12: CEBAF Large Acceptance Spectrometer for 12GeV
CND: Central Neutron Detector
CPU: Central Processing Unit
CTOF: Central Time-Of-Flight detector
DC: Drift Chambers
DCHB: Drift Chambers Hit-Based tracking engine
DCTB: Drift Chambers Time-Based tracking engine
DOCA: Distance Of Closest Approach
DPE: Data Processing Environment
EBHB: Hit-Based Event Builder
EBTB: Time-Based Event Builder
EC: Electric Calorimeter
EKF: Extended Kalman Filter
eV: electron Volt
FD: Forward Detector
FLOPS: Floating point operations per second
FPGA: Field Programmable Gate Array
FT: Forward Tagger
FTCAL: Forward Tagger CALorimenter
FTOFHB: Hit-Based Forward Time-Of-Flight detector
FTOFTB: Time-Based Forward Time-Of-Flight detector
GeV: Giga electron Volt
GPGPU: General Purpose Graphical Processing Unit
GPU: Graphical Processing Unit
HEP: High Energy Physics
HIPO: HIgh Performance Output (data format)
HPC: High Performance Computing
HTCC: High Threshold Cherenkov Counter
JLab: Jefferson Laboratories
JVM: Java Virtual Machine
KF: Kalman Filter
MINCHI2: `HITBASEDTRKGMINFITHI2PROB` Variable

MPC: Multiwire Proportional Chamber
MSE: Mean Squared Error
LHC: Large Hadron Collider
linac: linear (particle) accelerator
LQE: Linear Quadratic Estimator
LTCC: Low Threshold Cherenkov Counter
OOT: Out of Timers (Hits)
RF: Radiofrequency
RK4: 4th Order Runge Kutta
SV: State Vector
RICH: Ring Imaging CHerenkov detectors
SOA: Service Oriented Architecture
SVT: Silicon Vertex Tracker
TeV: Tera electron Volt
TJNAF: Thomas Jefferson National Accelerator Facility
ZMQ: Zero Message Queueing

Contents

List of Figures

INTRODUCTION

For much of the first half of the last century, it was believed that there were just three fundamental particles: protons, neutrons and electrons. These three particles however were only able to describe a small portion of the phenomena that defines particles physics, and improvements in particle accelerator and detector technologies gave way to the discovery of a plethora of new particles. This led to a richer understanding of particle physics and to a simple, unified theory to explain these particles on a fundamental level.

This unified theory eventually became known as the Standard Model of Particle Physics, and it explained the known particles with incredible accuracy. The theory describes two fundamental types of particles: **fermions** and **bosons**. **Fermions** make up all the "stuff" in the world, and **bosons** describe the way fermions interact with one another. **Fermions** are then further divided into **quarks**, which make up protons and neutrons, and **leptons**, which include electrons in addition to more elusive particles like muons, taus and neutrinos [Perkins and Perkins, 2000].

While the Standard Model could predict the properties of particles with incredible precision, it faced one major problem: it could not explain why particles have mass, and thus could not predict the masses of individual particles. To solve this problem, [Higgs, 1964] theorized an extension to the Standard Model, which involved the existence of another fundamental field which gives mass to all fundamental particles. To prove the existence of this field, a particle that could only come to be from the excitation of this field had to be observed: the famed **Higgs Boson**.

This particle remained a theory for nearly fifty years, until the European Organization for Nuclear Research (CERN) confirmed that the elusive particle had been observed in 2012 [Aad et al., 2012, Collaboration et al., 2008]. This observation was achieved by analyzing the particles produced in the collision of a beam of particles and a target, which is commonly named event in the field of High Energy Physics (HEP). The experiment was performed in the Large Hadron Collider (LHC) at CERN, which is a type of particle accelerator. The process of figuring out what particles in which conditions cause each detection or hit is called **event reconstruction** [Le Duff, 2005], and is usually done in software due to the complexity of analyzing the large amounts of available data [Demchenko et al., 2013].

To reconstruct an event from the available data, it is required to use both an understanding of the physics behind it and a plethora of mathematical tools to estimate the particles' characteristics and trajectories. Naturally, this process relies on the heavy use of many different hardware components to detect and categorize all the gathered data and of computing units to filter it and reconstruct the event. Thus, valuable information is obtained that can be used by physicists to further understand the underlying structure of the universe [Sirunyan et al., 2017]. An example of a common process the

data goes through from the detector to the physics analysis can be seen in figure 1.

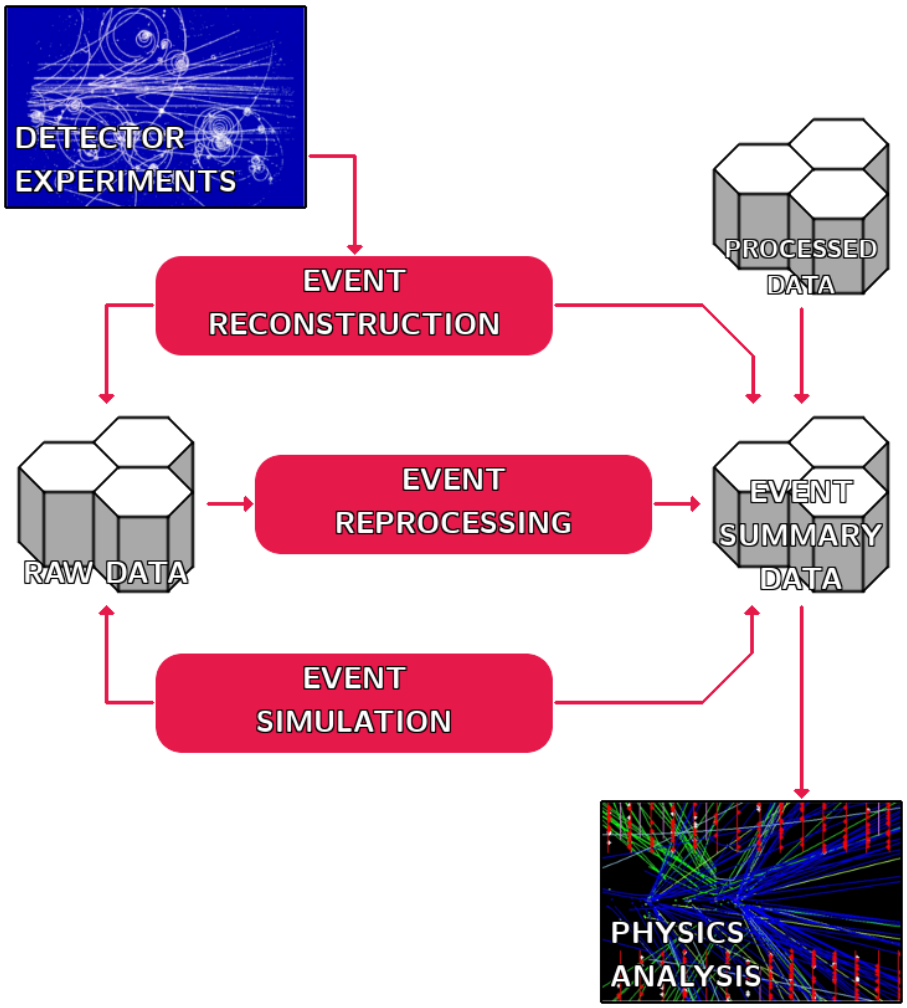

Figure 1: Example data process found in particle detector software. Source: Own elaboration. Images used are freely available at `https://home.cern/`.

A common problem that arises from this structure is the fact that, as technology progresses and particle detectors become more advanced, the produced data becomes more detailed and thus more expensive to process. This issue is seen too at the Thomas Jefferson National Accelerator Facility (TJNAF) in Virginia, USA, where the linear accelerator and its paired detectors produce large amounts of data faster than what the available hardware can cope with. Two natural solutions can be seen to this problem: Upscale the hardware to deal with the ever-increasing amounts of data to be processed; or improve the software to process more information without using new resources. While the first option can't always be avoided, it's easy to see that the second is usually preferable since it helps alleviate the expenses of upgrading hardware. In the

present document, this second solution is explored by thoroughly going through each factor that slows down the data processing speed and assessing them separately.

While the problem analyzed in this document belongs to the field of High Energy Physics, the effort done is more closely related to algorithms and complexity analysis. This is attributed to the fact that the nature of the work lies in the underlying framework of the event reconstruction process, which in its core is about minimizing the error in a fit. This by no means is a problem unique to physics. Due to this, the contribution of this work has multiple objectives: It directly benefits the software development team at TJNAF in that it accelerates the software used, and it also provides a framework for other projects that look into speeding up the computation of the event reconstruction software for other particle accelerators or systems in general.

Figure 2 shows a detailed tree diagram of the CLAS12 software components. CLAS12 is the reconstruction software used at TJNAF, and is described in detail in Section 3.2.

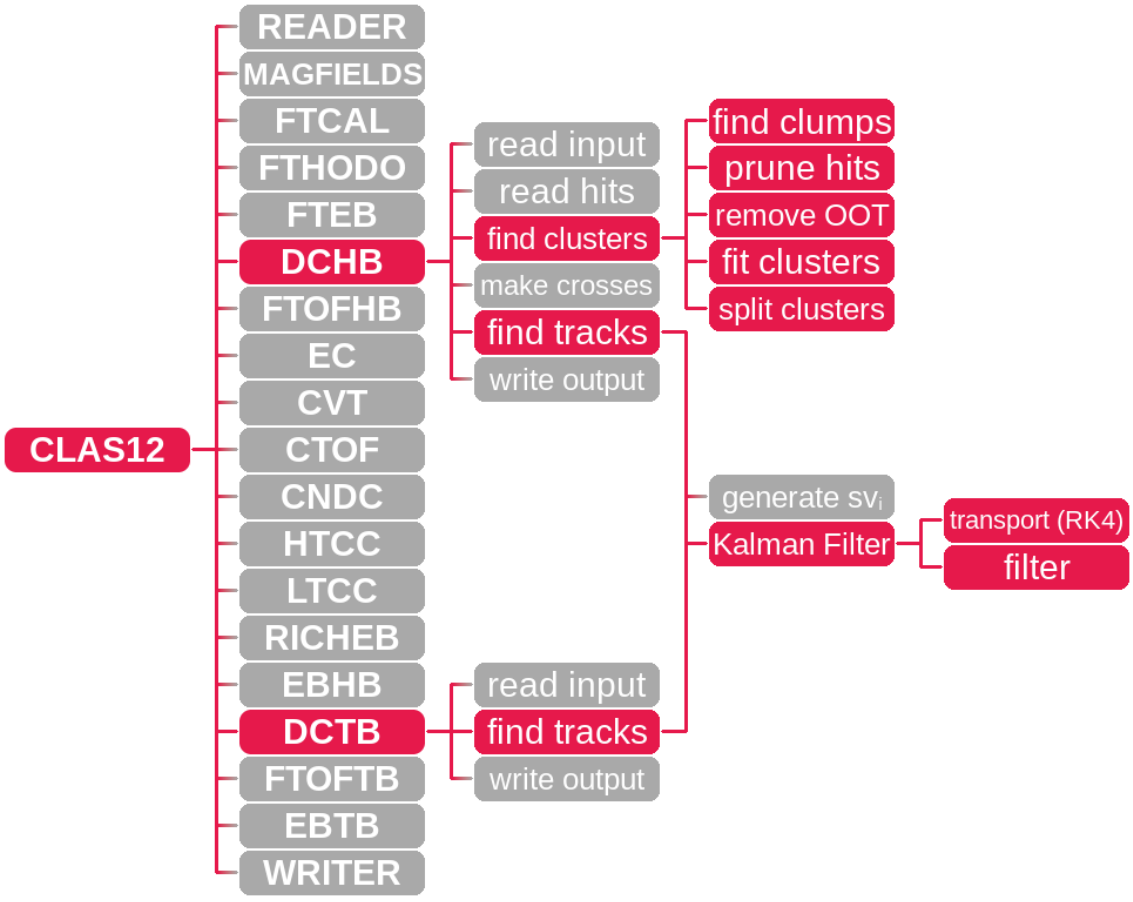

Figure 2: Tree diagram with the CLAS12 software components. The components where the project's work is focused are highlighted.

Chapter 1 presents a part of the current state of the art, along with a short description of each publication listed. In Chapter 2 the problem is formally defined, along with its context and the goals of the project. In Chapter 3 most of the large concepts that are related to the project and the document are explained, including the hardware and the software components. Chapter 4 elucidates in detail the profiling process described previously and presents the results of the profiling sessions. Chapter 5 explains the solutions implemented to each different bottleneck found in Chapter 4 and proposes an algorithm for future work. Then in Chapter 6 the validity of the results produced after each change is asseverated and the computing times are compared. Finally, Chapter 7 provides with the concluding remarks of the work, including the setbacks faced and future work is proposed.

CHAPTER 1
STATE OF THE ART

A list of state-of-the-art methods aiming at similar objectives as the one described in this work follows:

GPGPU Tracking in the COMET Phase-I Cylindrical Drift Chamber [Yeo et al., 2019]: The COMET Phase-I experiment is a project in Japan that is aiming at discovering neutrinoless coherent transition of a muon to an electron in the field of an aluminum nucleus. Just as in CLAS12, a Kalman filtering method is implemented to find the particle tracks, whose seeds are obtained by iterating through different combinations of hits using a fourth order Runge Kutta algorithm. A seed is initial state vector that is fed to the filter, as is described in Section 3.5.

This algorithm is implemented with the objective of finding the seeds with the minimum possible Distance Of Closest Approach (DOCA) to the wires in the detector, assigning only the seeds with a DOCA below a specific threshold to be investigated further with the Kalman Filter. Due to the fact that the algorithm iterates through many combinations of seeds and that the operations that need to be done on them are constant, it is implemented as a parallel algorithm for a General Purpose Graphical Processing Unit (GPGPU).

This work is useful for the study of the event reconstruction software related to Drift Chambers (DC) because it shows that a commonly used algorithm for DC can be leveraged to a GPU, potentially reducing the computing times.

Discriminative Training of Kalman Filters [Abbeel et al., 2005]: While useful in a plethora of fields, it's undeniable that the Kalman Filter is an essential tool in robotics due to its fine precision for state-estimation problems. Considering this, scientists and engineers from the Department of Computer Science at the Stanford University brought up the idea of using machine learning algorithms to fine-tune the parameters in the filter and minimize the effect of perturbations or noise in the estimation of state vectors.

In view of this objective, various training algorithms with different objective functions in mind were developed, such as maximizing the joint likelihood of all the data, maximizing the measurement likelihood or the prediction likelihood, among others. The way in which these algorithms help is in the fact that their output can help adjust the covariances of the EKF, in a way such that maximizes their predictive accuracy.

Accelerating the Kalman Filter on a GPU [Huang et al., 2011]: As is described in Section 2.2, the usual cause for high computing times for the EKF are related to the matrix operations performed by the filter. To counter this, the matrix computations can be leveraged to a GPU in order to accelerate the general computing time of the

filter.

[Huang et al., 2011] implemented this approach and proved that the Kalman filter time on CPU is approximately linear with the state dimension while on GPU it can be sublinear due to its ease for computing matrix inversions and multiplications. It's worth noting that while this publication is essential to most projects aiming to accelerate the KF, in the specific case of the CLAS12 software is not appropiate for because of the small size of the state vector being used [Moravánszky, 2003].

GPU Enhancement of the Trigger to Extend Physics Reach at the LHC [Halyo et al., 2013]: The idea of using computer vision and pattern recognition on detector data is explored by scientists at CERN, and considering that problems in this field were the origin of the GPU technology [Pharr and Fernando, 2005], they propose the idea of using these methods on detector data.

The main idea proposed in the publication is that of using the Hough Transform (described in Appendix 8.5) on a list of hits to find clusters following straight or curved lines. Although this publication makes good use of the Hough Transform, it does not provide a framework for using it, it rather is proposed as a complement to the conventional combinational track finder algorithms commonly used in High Energy Physics.

Kalman-Filter-Based Particle Tracking on Parallel Architectures at Hadron Colliders [Cerati et al., 2016]: While not directly looking to reduce the total computing time of the particle tracking algorithm, this publication looks into parallelizing the algorithms used at the Large Hadron Collider (LHC) to make use of lower-power, multi-core processor units such as GPGPUs.

These improvements begin with the implementation of a custom matrix library named **Matriplex** which is optimized for dealing with small matrices (no larger than 6×6), like the ones used for particle tracking at the LHC. To optimize the loading a small matrices into a GPU's memory, the library supports special vector registers for loading sets of matrices at once. **Matriplex** also includes a code generator for generation of optimized matrix operations supporting symmetric matrices and on-the-fly matrix transposition.

To optimize memory management, [Cerati et al., 2016] also try offloading the memory management work to a worker process or to another thread, thus grouping all memory operations together and removing redundant operations.

Continuing on the work with memory management, the size of the data structures used on the algorithm are also optimized in size to fit into the fastest cache memory. This is done by replacing the hit data, the track data and the covariance matrices from object-oriented data structures to C-style arrays, along with the optimization of the contents of the data structures.

CHAPTER 2
PROBLEM DEFINITION

2.1 Context

A particle accelerator is a machine that can accelerate charged particles to very high speeds and contains them in well-defined beams via an electromagnetic field, providing an environment in which controlled collisions can occur so that particle physicists can study the small particles that result from these impacts [Le Duff, 2005]. It is important to note that due to the fact that the special theory of relativity requires that matter always travels slower than the speed of light in a vacuum, a particle will never reach this speed. Due to this, the particle's speed is not thought of in traditional terms, but rather in terms of its energy or momentum, usually measured in electron Volts (eV).

As of the time of writing of this work, two types of accelerators are mainly used to study particle physics: the circular or cyclic radiofrequency (RF) and the linear accelerators. The former is, as the name suggests, a ring, where the particles' track is bent into a circle, allowing them to re-use the same track as much as possible so that they can reach energies up to the order of Tera eV (TeV) in some cases, like the well-known Large Hadron Collider (LHC), at CERN [Le Duff, 2005]. The latter, usually denominated linear particle accelerators (linacs), accelerate particles by attracting them onto charged plates and switching their charge after the particles pass the plate to repel them, pushing them to the next plate. While the particles accelerated by linacs generally achieve lower momenta than the ones pushed by their circular counterparts, they offer the advantage that they can produce a continuous stream of particles, whereas a cyclic accelerator can only periodically raise the particles to sufficient energy to merit a "shot" at the target [Pinchoff, 2005].

At the laboratory for which this work is done, TJNAF or Jefferson Laboratories (JLab) research is assisted by the use of one such linac, the Continuous Electron Beam Accelerator Facility (CEBAF). CEBAF is composed of a polarized electron source and a pair of superconducting radiofrequency linear accelerators connected to each other by two arc sections that contain steering magnets [Leemann et al., 2001]. The peculiar design of CEBAF allows for the continuous beam characteristic of other linacs, while also being able to reuse the tracks, effectively reaching energies found in an accelerator ten times its size [Leemann et al., 2001]. A picture of CEBAF can be seen in 3.

The produced beam ends at one of the four available experimental halls, labeled Hall A, B, C and D, each containing specialized spectrometers to record the products of collisions between the electron beam with itself or with a stationary target. Particularly, in experimental Hall B, the detector utilized to study the results of these collisions is the CEBAF Large Accelerator Spectrometer for 12 GeV (CLAS12), where each

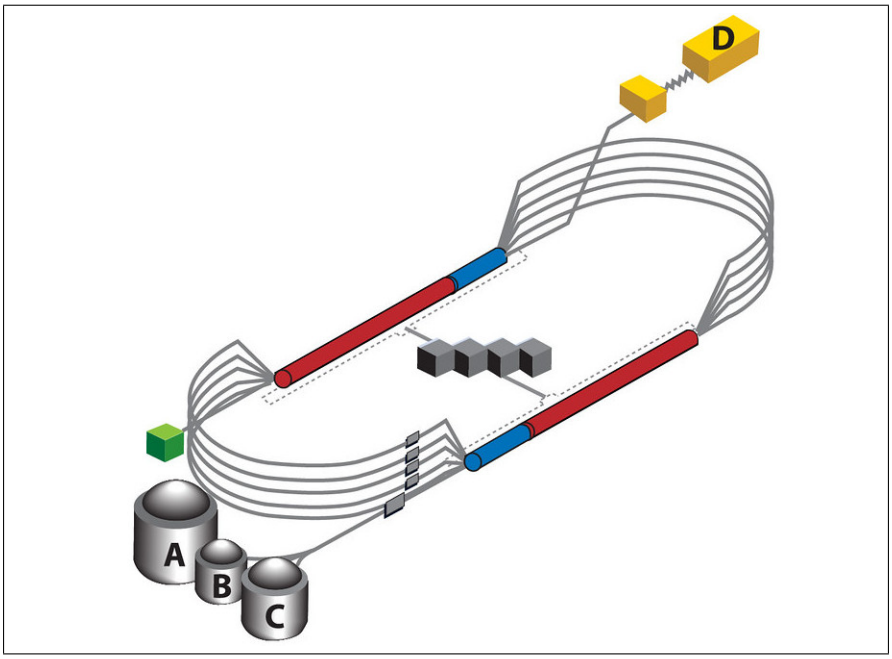

Figure 3: The Continuous Electron Beam Accelerator Facility (CEBAF). Source: `https://www.flickr.com/photos/jeffersonlab/12599705145`

particle-target collision, commonly denominated "event", is captured. This is done in an order of up to several thousand events per second, and the recorded data is later transferred to a farm of computing processors for teams of physicists to analyze them and look for new kinds of particles or information related to the structure of different particles [Mecking et al., 2003].

Most particle detectors deal with enormous amounts of data, usually too large to be analyzed "by hand" by the physicists trying to understand it, so it's common to build software that aids in the analysis process by filtering the useful data from the background noise [Demchenko et al., 2013]. In the case of CLAS12, this software follows a Service Oriented Architecture (SOA) and consists of the framework, event reconstruction, visualization, calibration, monitoring services as well as detector and event simulator. The scope of this work is centered mostly with the framework and the event reconstruction software, which are the CLAs12 Reconstruction and Analysis framework (CLARA) and the event reconstruction software itself, specifically with its components associated to the Drift Chambers (DC).

Roughly speaking, the CLARA framework acts as a fairly simple to use system to separate different sections of the software into services which run in so-called Data Processing Environments (DPEs). The DPEs allow for easy scaling of the software by leaving the distribution of the hardware resources to CLARA [Gyurjyan et al., 2013, Gyurjyan et al., 2015] using the ZeroMQ message transfer protocol [Hintjens, 2013]. The event reconstruction software works by analyzing the data obtained from each of

the different components of the CLAS12 detector and tries to reconstruct the event, as its name suggests [Ziegler, 2013].

Like any particle detector, CLAS12 is divided into a series of many different components that detect different types of particles utilizing a variety of detection methods [Pinchoff, 2005]. One such kind of detectors are the Drift Chambers (DC), which work by measuring the fluctuations in the charge of the matter inside the chambers and in turn try to "guess" what passing particle caused such difference in charge. This measurement is obtained via a set of wires acting as cathodes which, in the case of CLAS12, are strung inside 18 wire chambers inside 3 hexagonal shapes, each with 2 superlayers of 6 layers with 112 wires each, thus having a total of 24,192 wires [Mestayer et al., 2017]. These measurements are obtained with the hope that a set or cluster of hits in different wires will allow to estimate the trajectory of the particle that caused these hits and in turn understand its properties. A diagram of the CLAS12 detector can be seen in Figure 4, where two particle trajectories are marked in a pointed line. The wires hit by each trajectory are marked in red in the image to the left and the area that is detected by each wire is denoted by the hexagons in the image to the right.

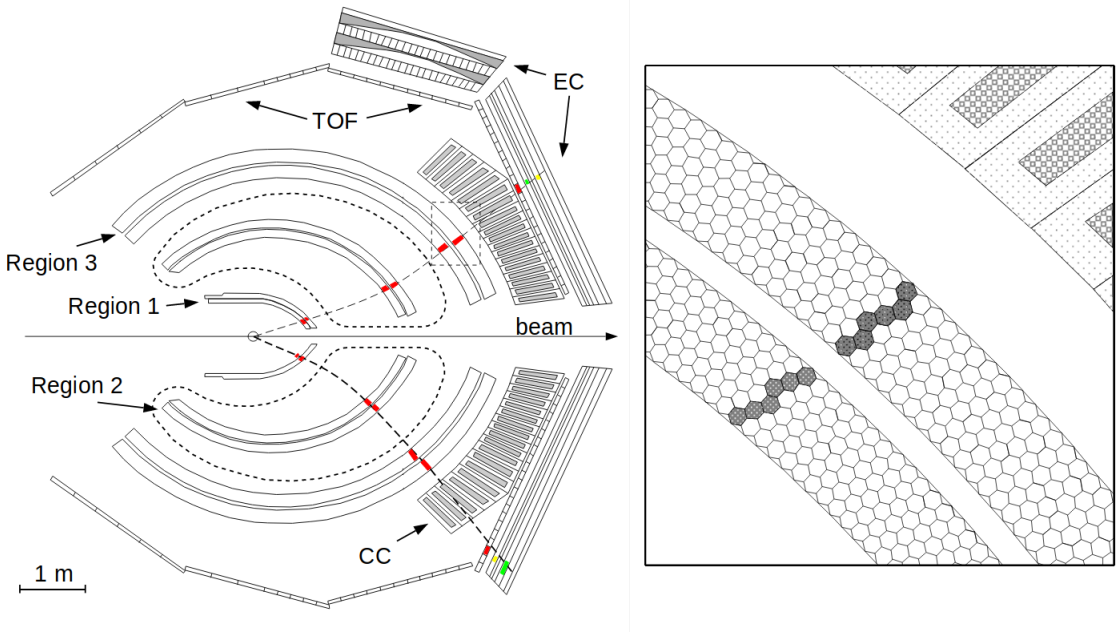

Figure 4: Horizontal cut through the CLAS detector at beam line elevation showing two charged particles traversing the 18 superlayers in the 3 regions of the drift chambers in opposite sectors. The picture to the right shows an enlargement of the boxed area in the picture to the left. Source: The CLAS Drift Chamber System [Mestayer et al., 2000].

2.2 The Problem

Different algorithms are used to estimate the trajectory or track of some of the particles that are ejected in each event, and these algorithms belong in a field named particle tracking. In the specific case of the DC in CLAS12, particle tracking is done via the Extended Kalman Filter (EKF), which is a popular algorithm for estimating the position of a moving object with a good precision given that the path followed by it can be modeled by deterministic movement equations and that measurements (i.e: the ones taken by the wires) can be obtained for this object's position over time. This makes it a good estimator for the charged particles ejected by the target after each event [Kalman, 1960].

The EKF shows great performance at its task in the reconstruction software, but it suffers from the disadvantage that it is a fairly slow algorithm. To reach the previously mentioned precision, both the Kalman Filter (KF) and its extended counterpart require matrix multiplications and inversions when computing states, as well as the calculation of Jacobian matrices, as is described in Section 3.5. These matrix operations mean that the computational complexity of the algorithm is quite high, which in big-O notation is:

$$O(\mathrm{C}(\mathbf{f}(\cdot,\cdot)) + \mathrm{C}(\mathbf{h}(\cdot)) + n^3 + n^2\,m + n\,m^2)\,,$$

where n and m are the state and measurement vector sizes and $C(\mathbf{f}(\cdot,\cdot))$ along with $C(\mathbf{h}(\cdot))$ are used to denote the costs of $\mathbf{f}(\cdot,\cdot)$ and $\mathbf{h}(\cdot)$ respectively, where $\mathbf{f}$ and $\mathbf{h}$ are the transition functions between two arbitrary timesteps k and $k+1$ for the state vector $\mathbf{x}_k$ and the measurement vector $\mathbf{z}_k$ respectively. The mathematical formulation of the model and the calculation of the computational complexity of each operation are described in detail in Section 3.5.

Something to note is that while most systems using the Kalman Filter are slowed down because they process large state and measurement vectors with large covariance matrices, the DC software suffers from a different issue. The n and m mentioned before are constants 5 and 6 respectively, but the state transition itself between states k and $k+1$ takes a large amount of time and is iterated over a large number of times as is described in Sections 3.5, 3.6 and 3.7. Even more, the Kalman Filter is used as a fixed point iteration to compute the state vector at the last measurement site K with enough precision, requiring many runs to obtain precise results.

This large computation time takes a heavy toll on the total running time of the two engines that process the DC data in the reconstruction software: the DC Hit-Based tracking engine (DCHB) and the DC Time-Based tracking engine (DCTB); taking a total of 49.3% of the total running time of the whole CLAS12 software. It's also worth noting that the second component in terms of processing time is the Cluster Finding used in DCHB and DCTB, which takes in total a 40.1% of the previously mentioned time. To better visualize the impact on the processing time of the DCHB and DCTB

components, see Figure 5. As all the other performance tests in this document, these measurements were obtained by running the CLAS12 software on 10.000 events with real experimental data. The test file used is named `clas_004013.0.hipo`, and can be obtained by contacting the Hall B team at Jefferson Lab (https://www.jlab.org/Hall-B/).

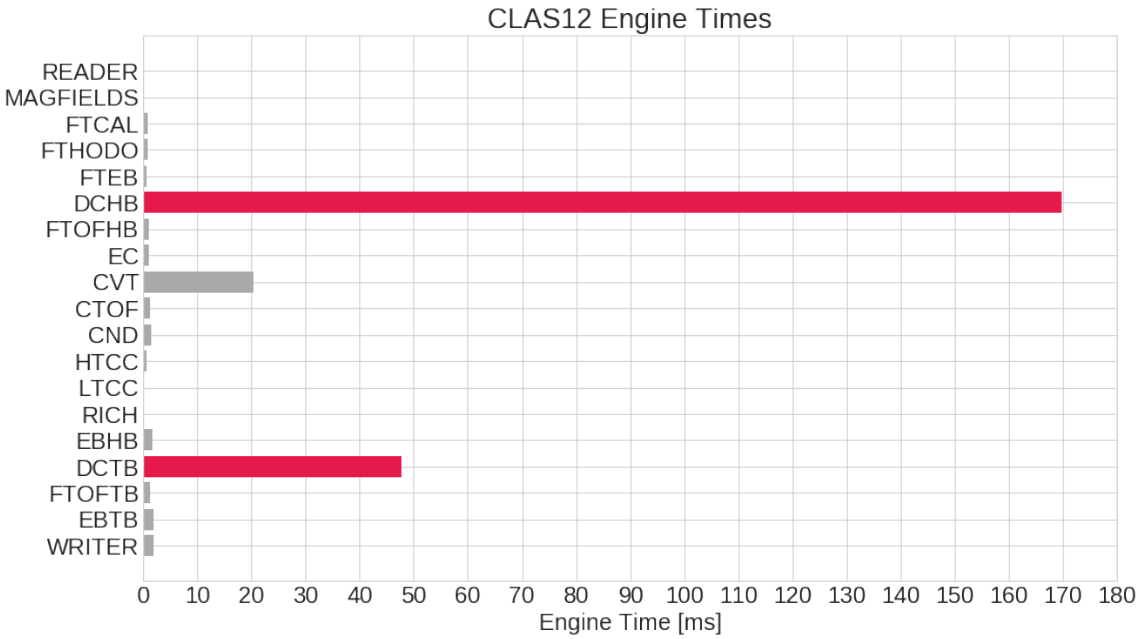

Figure 5: CLAS12 Engine times as of September 7th, 2018.

The problem that this project focuses on is that of reducing the total computation time of the whole CLAS12 offline software. To tackle the problem, the project focuses in the slowest components of CLAS12, DCHB and DCTB, and especifically to the acceleration of the Kalman Filter used in these two DC engines. Another smaller part of the project also focuses in the acceleration of the cluster finding algorithm, considering that it is the second largest bottleneck in the software's execution.

2.3 Goals

As can be seen in Figure 5, the combined time taken by the DCHB and DCTB engines is as much as 86.35% of the total execution time of the CLAS12 software, making them prime candidates for accelerating the reconstruction software. Therefore, the main goal of this project is to reduce the total running time of the CLAS12 software by specifically accelerating the KF algorithm currently implemented in the code while also trying to tackle the cluster finding algorithm to a lesser extent. The project's major goals are:

- Compare a set of the different Java profilers and analyze which one is the most useful to measure the CLAS12 software performance.
- Study which components of the KF and Cluster Finding algorithms implemented in CLAS12 can be accelerated via optimizations and/or parallelization.
- Study the way in which these optimizations and/or parallelizations could take place in the context of execution of the software, the CLARA framework.
- Effectively apply the optimizations and implement the algorithms that would allow a substantial improvement in the running time of the software.
- Profile and test the code continuously while applying changes, ensuring that the output data is consistently the same through the development process.
- Document the changes made to the code and describe in detail the algorithms and optimizations implemented in order to help future related work.

It's worth noting that these objectives differ slightly from the ones originally proposed for this project: Originally only parallelization techniques were going to be tested, but during development it was decided to apply general optimizations.

CHAPTER 3
CONCEPTUAL FRAMEWORK

3.1 The CLARA Framework

CLARA is a SOA platform designed to build streaming scientific-data analytics applications that allows users to process data as streams across data centers and clouds [Gyurjyan et al., 2011]. The framework is used both in JLab for CLAS12 and in NASA for the NASA Information and Data System (NAIADS) and Surface Radiation Budget (SRB) projects [Lukashin et al., 2015].

As defined in the Introduction, a data event is the collision of a beam of particles into a target which results in a shower of particles. A data processing engine or simply engine is a software component that processes the data event, usually reconstructing the data obtained in a specific detector component or providing a secondary functionality.

The CLARA framework consists of four core components: A Data Processing Station, a Pipe, a Data Processing Environment and an Orchestrator:

- **Data Processing Station**: Usually referred to as a CLARA station, the Data Processing Station acts as a container for a user-made data processing engine. In the context of the CLAS12 software, this acts as the container for each engine associated to a hardware component of the CLAS12 detector, with a simple relationship of one station for one engine. This station-engine coupling is denoted as a Data Processing Service or simply service, and can be seen in Figure 6.

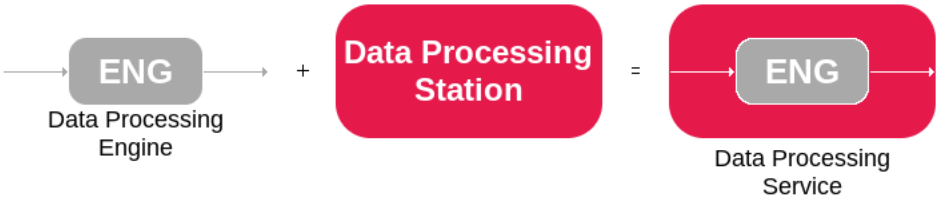

Figure 6: Data engine and CLARA station coupling.

- **Data Stream Pipe**: A data bus based on the xMsg messaging system, which in turn is based on the ZeroMQ protocol [Hintjens, 2013]. xMsg is a messaging protocol based on the publisher-subscriber pattern which allows for various agents to communicate based on topic subscriptions. In the context of the CLAS12 software, the data buses act as pipes between services, allowing one service to send Data Events to another. The interaction between stations and pipes can be seen in Figure 7.

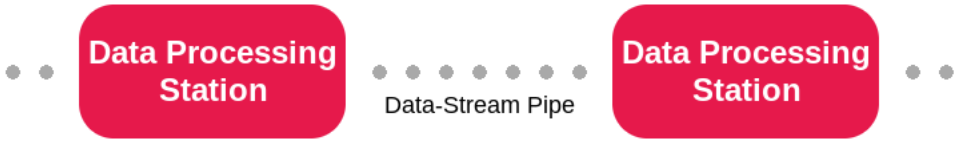

Figure 7: CLARA stations coupled with pipes.

- **Data Processing Environment (DPE)**: A DPE is a container unit for one or more services and their corresponding data pipes which acts as a communications hub between the services and the hardware. Communication between DPEs can be done using the same data stream pipes. Different DPEs can be placed in different computing environments, thus allowing the communication of different services between different machines.
- **Data Flow Orchestrator**: The Orchestrator is a system to coordinate the communication between the Data Processing Services via the Data-stream Pipes. In CLAS12, the Orchestrator simply forms a straight line between services which is called "Production Chain", and each service alters the Data Event as it sees fit to then have the engines following read and write it. To setup a production chain such as the one seen in Figure 8, the Orchestrator must be used to configure each service and pipe to allow the free flow of data.

Figure 8: CLARA production chain.

3.2 CLAS12 Offline Software

The CLAS12 Offline Software is the software associated to reconstructing the data detected at the CLAS12 detector from Hall B. It is separated into two components: **common tools** and **reconstruction** [Ziegler, 2013].

The **common tools** package is a set of libraries developed by the physicists at JLab that contain all the tools that are commonly reused by different engines, like the **detector geometry** or **magnetic fields** packages. The **reconstruction** package, which is where most of this work is focused, is composed by a set of semi-independent engines connected via the aforementioned CLARA pipes, where each detector component is represented by one or more CLARA Data Processing Services.

These services are separated into two categories, ones that "help" the data flow (Event Builders, IO, etc) and others that describe the hardware of the CLAS12 detector. A diagram of the detector can be seen in Figure 9.

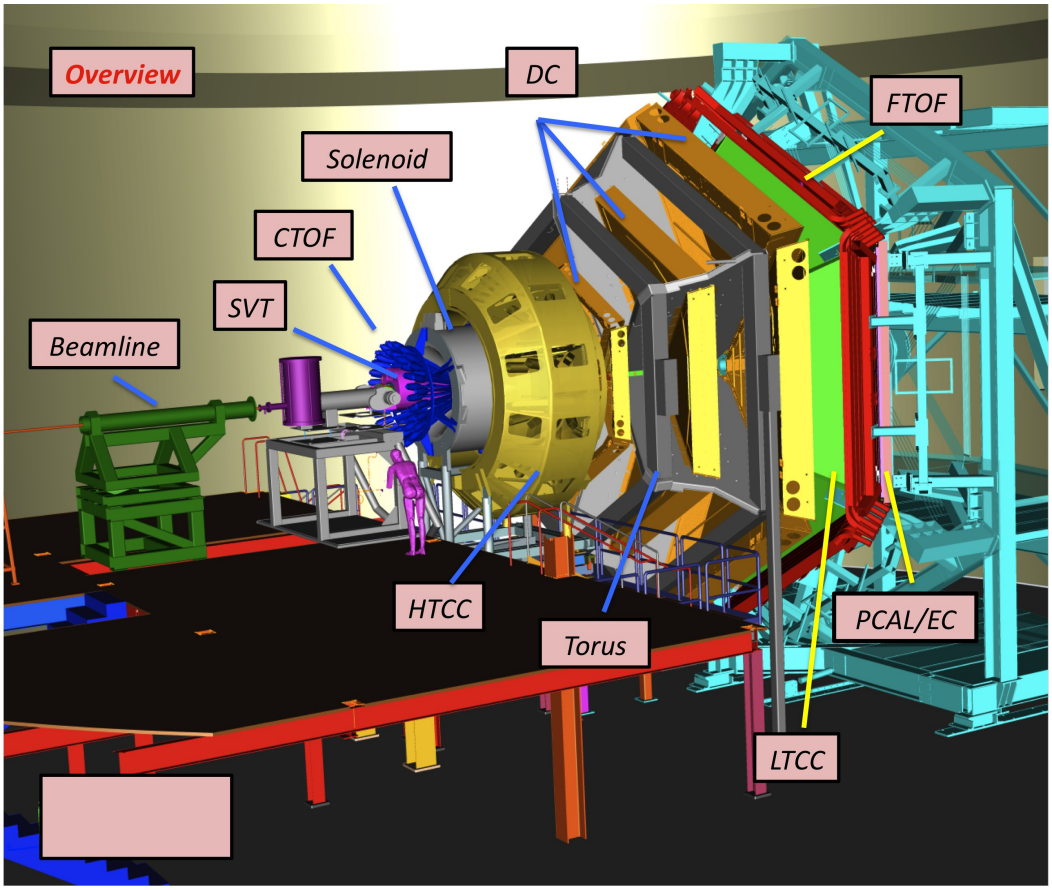

Figure 9: CLAS12 detector hardware design. Source: CLAS12 web (`https://www.jlab.org/Hall-B/clas12-web/`).

A comprehensive list of CLAS12 software engines and their associated physical components (when available) is given. It is worth noting that many engines run twice for each data event, first running a Hit-Based (HB) analysis and then a Time-Based (TB) analysis. HB analysis is done without using time-related data and is mainly used to filter through the data and eliminate noise, while TB analyzes the filtered data and finds the final particle tracks.

- **HIPO READER and WRITER**: Engines that provide the file reading and writing, based on the HIgh Performance Output (HIPO) data format [Gavalian, 2019].
- **Event Builder Hit-Based (EBHB) and Event Builder Time-Based (EBTB)**: Engines that build the data events from the HIPO data and pass them to the other engines.
- **MAGFIELDS**: Engine providing a model for the magnetic field inside the detector, used by components that need to estimate a particle's trajectory.
- **High Threshold Cherenkov Counter (HTCC) and Low Threshold Cherenkov Counter (LTCC)**: Engines detecting particles moving faster than the local speed of light in a medium utilizing the emitted cone of light. The HTCC detects pions with high momenta ($> 5\ GeV/c$), while the LTCC the ones with low momenta ($> 3\ GeV/c$), and both are part of the Forward Detector (FD), a superconducting Torus magnet that covers the range from 5° to 40° measured from the source of the event, as can be seen in Figure 4 from Section 2.1.
- **DC Hit-Based tracking (DCHB) and Time-Based tracking (DCTB)**: The engines where most of the work in this thesis is focused, and are explained in detail in Section 3.3. Part of the FD.
- **Forward Time-Of-Flight Hit-Based detector (FTOFHB) and Time-Based detector (FTOFTB)**: Plastic scintillators that provide precise time-of-flight measurements for charged particle identification up to 4.5 GeV/c. Scintillators are materials that show a scintillation or flash of light when excited by ionizing radiation. Part of the FD.
- **Ring Imaging Cherenkov (RICH) Detectors**: Similar in design to the FTOF detector, but capable of detecting charged particles with momenta of up to 8 GeV/c instead of 4.5. Part of the FD.
- **Electromagnetic Calorimeter (EC)**: Allows for the detection of single high-energy photons by using a pre-shower calorimeter, improving spatial resolution and the separation of two photos for momenta of up to 10 GeV/c. In particle physics, a shower refers to the cascade of secondary particles after the collision between the beam and the target. Part of the FD.

- **Central Time-Of Flight detector (CTOF)**: A scintillator array allowing for pion identification in a momentum range of up to 1.2 GeV/c. Part of the Central Detector (CD), which is a superconducting Solenoid magnet that covers the polar angles from 35° to 135°.
- **Central Neutron Detector (CND)**: A barrel with scintillator bars that detects high energy neutrons with an efficiency of up to 15%. The efficiency of a neutron detector refers to how many neutrons that pass through it are detected [Pinchoff, 2005]. Part of the DC.
- **Forward Tagger (FT) electromagnetic calorimeter (FTCAL), FT tracker (FTEB) and FT scintillation counter (FTHODO)**: An extension of the CLAS12 capabilities that allows it to detect electrons and photons at angles as low as 2.5°.
- **Silicon Vertex Tracker (SVT) and Barrel Micromegas Tracker (BMT)**: Reconstruct the trajectories of charged tracks in the angular region from 35° to 125° with a momenta up to 1 GeV/c. In the code, both components are contained inside the CVT engine.

The production chain integrating all of the engines that represent these detectors can be seen in Figure 10.

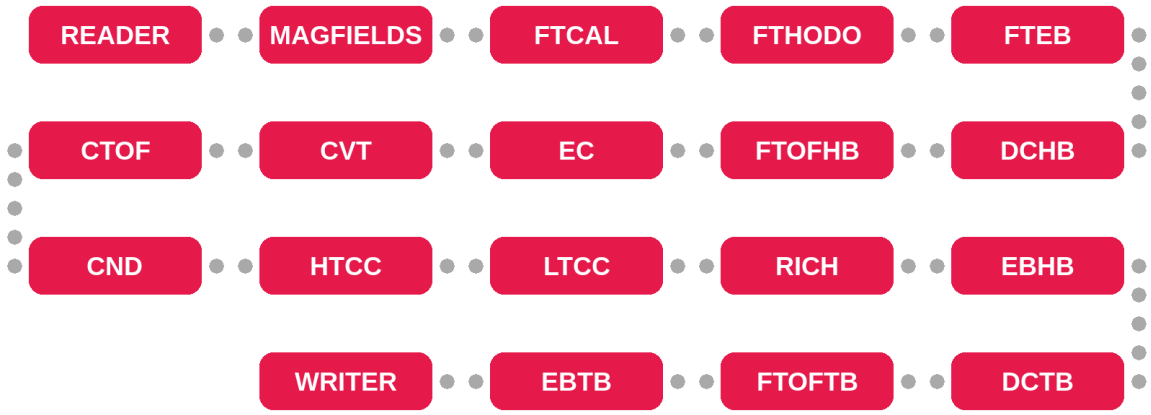

Figure 10: CLAS12 engines production chain.

3.3 Drift Chambers

Drift Chambers in general are a type of Multiwire Proportional Chambers (MPC) that use an array of wires at high voltage running through a chamber held at ground potential which is filled with gas. Any ionized particle passing through the chamber will ionize its surrounding atoms, accelerating them across the chamber so that they can be collected by the nearest wire. By computing the different pulses on each of these wires, the trajectory of the particle can then be estimated [Sauli, 1977].

DC differentiate themselves from MPC by the fact that they can obtain very precise measurements of the timing of the pulses from the wires along with the information commonly obtained by MPC. This allows a more precise estimation of the distance at which the particle passed the wire and greatly improves the accuracy of the path reconstruction [Blum et al., 2008].

As can be seen in Figures 4 and 11, the CLAS12 Drift Chamber system measures the momentum of charged particles emerging from the target using a set of 18 wire chambers distributed into 3 regions with 6 wire chambers each. Each wire chamber consists of 2 superlayers, while each superlayer of 6 layers and each layer of 112 wires, making up to a total of $24,192$ sense wires strung on the entire machine [Mestayer et al., 2000]

After obtaining the measurements, the DCHB and DCTB CLARA engines run to process them and deduce the tracks of the detected particles. Pattern recognition for the DC is first done on a hit-based basis, where a hit is defined as a wire with a recorded signal. Prior to searching for track segments (groups of hits) in each superlayer, a hit-pruning algorithm is employed to reject apparent noise, which is defined based on the number of contiguous hits in a layer.

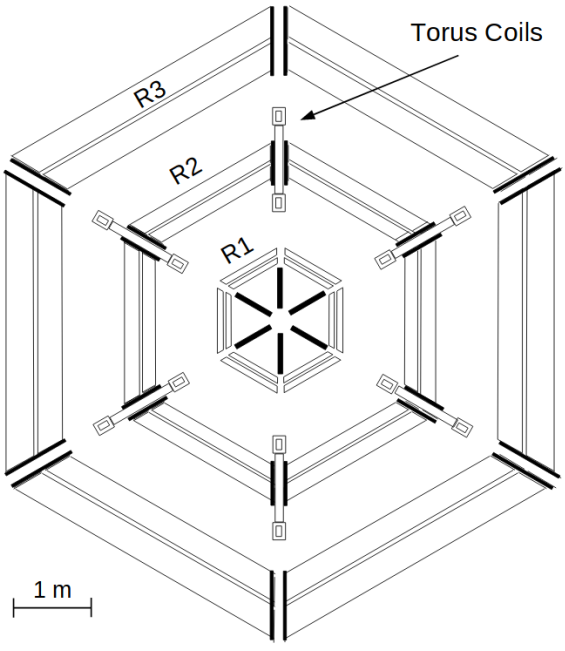

Figure 11: Vertical cut through the drift chambers transverse to the beam line at the target location. Source: The CLAS Drift Chamber System [Mestayer et al., 2000].

Since the particles should go through layers close to horizontally, if many hits are detected contiguously in one layer it usually means that either the wires are malfunctioning or various wires are detecting the same particle, so unnecessary hits are rejected.

The pruned hits are then grouped into clusters, defining a cluster as a contigu-

ous set of hits spanning through different layers in a superlayer. A more detailed description of the cluster finding algorithm is given in Section 3.4. The pairs of clusters in two adjacent superlayers are denoted as crosses. Linear fits to these crosses serve as cluster refiners and are a preliminary step to estimating a particle's trajectory or track.

Then, a fit is done to sets of three crosses which can be linear o curved depending on the strength of the local effect of the magnetic field in the particle's trajectory. This fit is commonly denoted as track fitting. Track parameters are estimated in the global coordinate system of the detector from this trajectory, and are then fed as input to the Kalman Filter, which refines an estimate of the parameters of a track.

A general description of the KF algorithm is explained in Section 3.5, while the specific implementation done for the track finding in CLAS12 can be read in Section 3.6.

3.4 Cluster Finding

The Cluster Fitting algorithm used at the DC can be described by dividing it into the five algorithms that conform it, which are **Clump Finding**, **Hit Pruning**, **Out of Timers Removal**, **Cluster Fitting** and **Cluster Splitting**.

Clump Finding: A clump is defined as a simple set of adjacent hits that spans at least `DC_MIN_NLAYERS` layers, which is a constant at CLAS12 usually set at 4. In the image to the left of Figure 12, the clusters formed from a group of hits in one superlayer is denoted. Each clump formed from the hits is marked using a different color and is also denoted using a simple numbering system. Running the clump finding algorithm on this group of hits, it can be seen that clumps (1) and (3) are valid since they span at least 4 layers while clump (2) is invalid for the opposite reason.

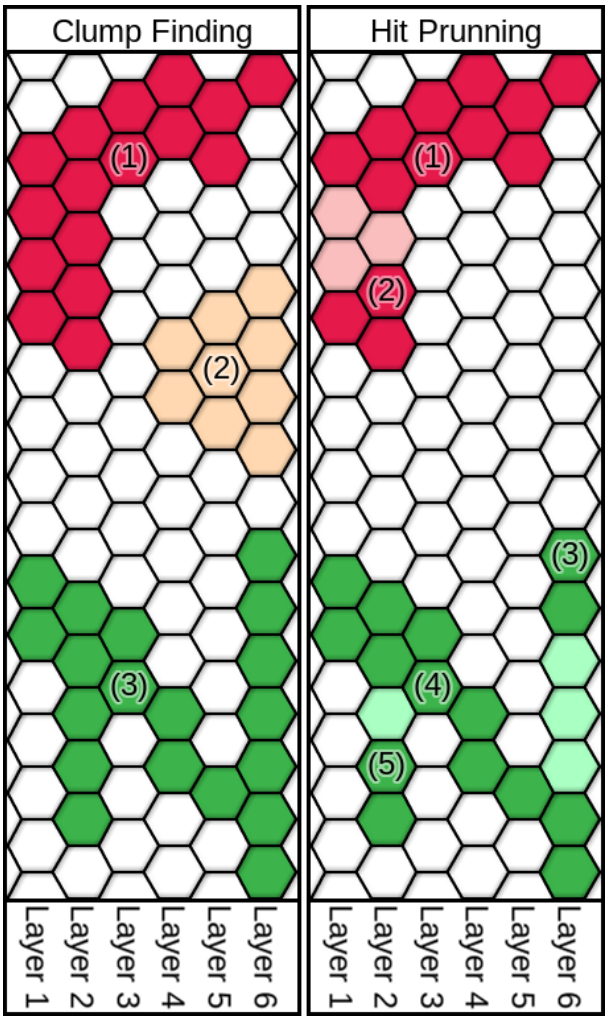

Figure 12: Clump Finding & Hit Pruning algorithms over the layers in a superlayer.

Hit Pruning: Hits that are considered noise are pruned in this step. Four cases are considered for each sequence of hits in a layer: **(I)** if the sequence contains more than 10 hits, all hits are considered noise and are removed. **(II)** If the sequence contains between 5 and 10 hits, only two at each end of the sequence are conserved. **(III)** If the sequence contains between 2 and 4 hits, only one hit from each end is conserved. **(IV)** If 2 or less hits are in the sequence, no hits are removed.

In the image to the right of Figure 12, the hits removed from each clump are denoted by a lighter color than the ones that stay. After running the pruning algorithm, the **Clump Finding** algorithm is ran again on the now pruned hits. The new clumps are denoted again in the image, with clumps (2), (3) and (5) considered invalid.

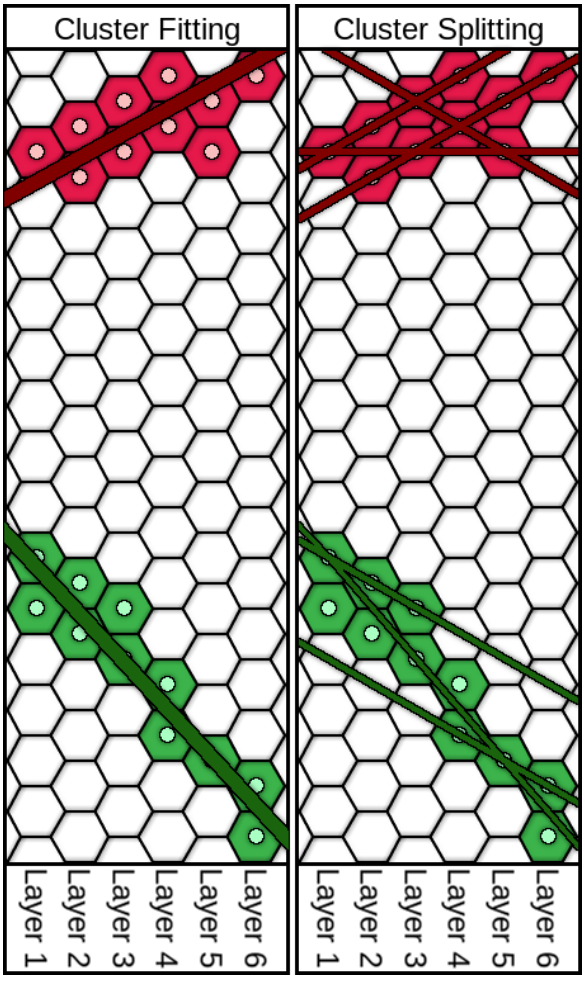

Figure 13: Cluster Fitting and Splitting algorithms over the layers in a superlayer.

Out of Timers (OOT) Removal: This method iterates through all the hits in each clump and removes the ones considered Out of Timers.

"Out of Timers" are hits that have a Distance Of Closest Approach (DOCA) higher than their cell size. A hit's DOCA refers to the closest distance between the charged particle and the center of the wire in the particle's trajectory, and is measured by analyzing the fluctuations in the wire's charge through time [Blum et al., 2008]. A wire's cell size simply refers to the size of the side of the hexagonal cell that contains the wire, as can be seen in Figures 12 and 13.

Cluster Fitting: After the clumps are pruned and cleaned from out of timers, a linear fit is attempted for each. The fit is considered successful if its associated χ^2_k error is lower than a pre-defined constant threshold named `HITBASEDTRKGMINFITHI2PROB` (denoted as χ^2_{min}), where k, or the number of degrees of freedom, is the number of hits minus 2. In the image to the left of Figure 13, a drawing of a possible fit line for each of the clusters obtained in the last step is shown. χ^2 error is described in Appendix 8.2.

Cluster Splitting: When the fitted cluster's χ^2_k error is above `MINCHI2`, it becomes necessary to split it and check for better fits using these split clusters. To do this, all the hits in a cluster are passed through a **Hough Transform**, which is described in Appendix 8.5. In simple terms, the Hough Transform is a method to detect lines from a set of points via representing these points in a special "Hough Space" [Duda and Hart, 1971]. After the new clusters are formed, only the ones with a χ^2_k error lower than `MINCHI2` are accepted.

In the image to the right of Figure 13, multiple lines that could be obtained via the Hough Transform are presented. It is worth noting that, in the current state of the software, a hit may end up being placed in more than one cluster if these clusters were produced by the Cluster Splitting procedure.

3.5 Extended Kalman Filter

The Kalman Filter (KF) is a useful algorithm to analyze a dynamical system that can be modeled regardless of how uncertain the information that is known about it is [Rauch et al., 1965]. Its utility is given by the fact that it can estimate the current state of a process while minimizing the average quadratic error (described in Appendix 8.2) using only the measurement system and the last state, without requiring access to the whole chain of events previous to the current one [Einicke, 2006]. Kalman filters are commonly used for control of vehicles, signal processing and econometrics [Zarchan and Musoff, 2013].

This filter, also known as the Linear Quadratic Estimator (LQE), is an algorithm that estimates the value and uncertainty of variables associated to a statistical noise using a series of measurements through time [Kalman, 1960], representing the variables as a state vector $\mathbf{x}_k$ and an observation (or measurement) vector $\mathbf{z}_k$ for an arbitrary moment on time or "timestep" denominated k.

The basic Kalman Filter, as useful as it is, is limited in the sense that it assumes that the model is linear, and thus is of little use when facing problems with non-linear processes or observation models [Simon, 2006]. For this type of problems, the so-called Extended Kalman Filter (EKF) is used [Karimipour and Dinavahi, 2015], which only requires the functions used to predict the next state and the next observation to be differentiable. It linealizes over an estimation of the current expected value and the covariance of the state vector. It's worth noting that the cited textbook explains the original work from $1959 - 1961$ since no publications were found explaining the EKF from that time by the author.

Different from its linear counterpart, the EKF is not an optimum estimator, and if fed an erroneous initial vector it can quickly diverge. The estimated covariance matrix can also underestimate the real covariance matrix, being sometimes inconsistent without the addition of stabilization noise [Julier and Uhlmann, 1997]. Due to this and the sometimes inefficient calculation of Jacobian matrices (described in addendum 8.4) necessary for the EKF, it is often preferred to use the Unscented Kalman Filter (UKF) [Julier and Uhlmann, 1997].

Regardless, for the specific case of CLAS12, it was decided during development that only the EKF was necessary since, as described in Section 3.3, the filter is fed preprocessed data with already some degree of precision. While not linear, the **predict** and **update** functions (explicitly defined in Section 3.6), are quasi-linear and their Jacobian matrices aren't complicated enough as to require the usage of the UKF, with them only being 5×5.

To describe the Extended Kalman Filter (EKF) and derive its computational complexity, it is necessary first to provide a mathematical formulation for the next state $\mathbf{x}_k$ and the obtained measurements $\mathbf{z}_k$ with transition functions $\mathbf{f}(\mathbf{x}, \mathbf{u})$ and $\mathbf{h}(\mathbf{x})$ respectively:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k$$
$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k \,.$$

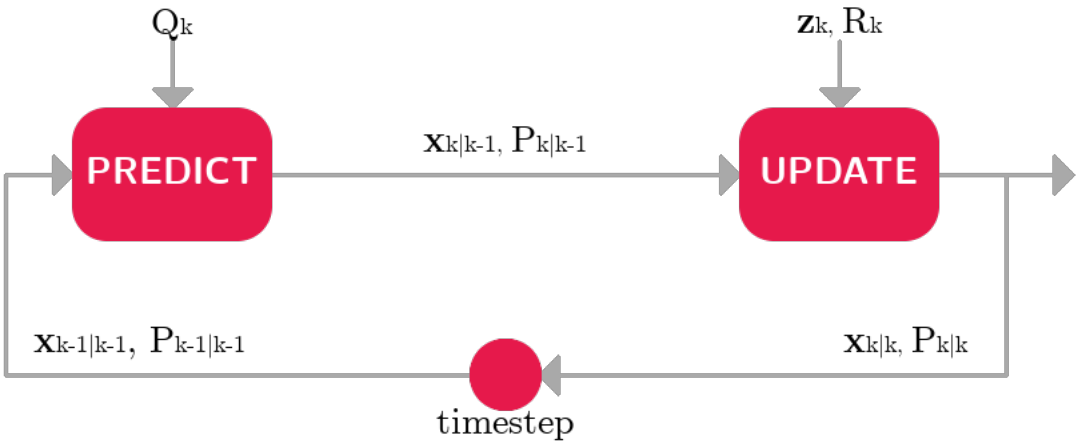

Figure 14: System diagram showing the EKF process to estimate a state vector $\mathbf{x}_{k|k}$ and its covariance matrix $P_{k|k}$ from $\mathbf{x}_{k-1|k-1}$, $P_{k-1|k-1}$ and $\mathbf{z}_k$ with noise matrices Q_k and R_k.

where $\mathbf{u}_k$ is the control vector and $\mathbf{w}_k$ and $\mathbf{v}_k$ are the errors in the measurement process, which are assumed to be Gaussian noises with an average of zero and with covariance matrices Q_k and R_k respectively. For the purposes of this project the control vector $\mathbf{u}_k$ is left as $\mathbf{0}$ considered that it denotes user input, which isn't present in this model.

A note on the notation: $k_1|k_2$ refers to state in step k_1 given the information known in step k_2, where k_1 and k_2 are arbitrary steps. Based on this, $\mathbf{x}_{k|k-1}$ is the state vector $\mathbf{x}$ in step k knowing the information from $k-1$. The exact logic on how this is updated is related to the measurement vector and is explained later in this section.

The transition and observation matrices $J_{\mathbf{f}(k)}$ and $J_{\mathbf{h}(k)}$ are also defined as the following Jacobian matrices:

$$J_{\mathbf{f}(k)} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Big|_{\hat{\mathbf{x}}_{k-1|k-1}}$$
$$J_{\mathbf{h}(k)} = \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\Big|_{\mathbf{z}_k} \,.$$

To calculate the computational complexity of the algorithm, the size of the state vector $\mathbf{x}_k$ is first defined as n and the size of the measurement vector $\mathbf{z}_k$ as m, while also defining the computational cost of applying the functions $\mathbf{f}(\mathbf{x}, \mathbf{u})$ and $\mathbf{h}(\mathbf{x})$ on an arbitrary input as $\mathrm{C}(\mathbf{f}(\cdot,\cdot))$ and $\mathrm{C}(\mathbf{h}(\cdot))$ respectively.

To obtain the next state two steps are required, **Predict** and **Update**. The former requires the following computations to obtain the estimate $\hat{\mathbf{x}}_{k|k-1}$ and the covariance matrix $P_{k|k-1}$ *a priori*:

$$\begin{aligned} \hat{\mathbf{x}}_{k|k-1} &= \mathbf{f}(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k) && \rightarrow \mathrm{C}(\mathbf{f}(\cdot,\cdot)) \\ P_{k|k-1} &= J_{\mathbf{f}(k)} P_{k-1|k-1} J_{\mathbf{f}(k)}^T + Q_k && \rightarrow 2n^3 + n^2\,, \end{aligned}$$

Then, to calculate the estimated state $\hat{\mathbf{x}}_{k|k}$ and the covariance matrix $P_{k|k}$ *a posteriori*, the following calculations are done in the latter step:

$$\begin{aligned} \tilde{\mathbf{y}}_k &= \mathbf{z}_k - \mathbf{h}(\hat{\mathbf{x}}_{k|k-1}) && \rightarrow m + \mathrm{C}(\mathbf{h}(\cdot)) && \\ S_k &= J_{\mathbf{h}(k)} P_{k|k-1} J_{\mathbf{h}(k)}^T + R_k && \rightarrow n^2 m + n m^2 + m^2 && (1) \\ K_k &= P_{k|k-1} J_{\mathbf{h}(k)}^T S_k^{-1} && \rightarrow n^2 m + n m^2 && (2) \\ \hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + K_k \tilde{\mathbf{y}}_k && \rightarrow nm + m && \\ P_{k|k} &= (I - K_k J_{\mathbf{h}(k)}) P_{k|k-1} && \rightarrow n^2 m + n^2\,, && (3) \end{aligned}$$

where $\tilde{\mathbf{y}}_k$ is the called residual measurement of the step k, S_k is its covariance matrix, named residual covariance matrix, and K_k is defined as the Kalman gain during that step. The Kalman gain is a matrix of gain from each measurement to each estimation, and if it's zero then $\mathbf{x}_{k|k} = \mathbf{x}_{k|k-1}$.

To calculate the computational complexity of each step, it is assumed that the complexity of multiplying two matrices of the same size $p \times p$ is $O(p^3)$, multiplying matrices of sizes $p \times q$ and $q \times r$ is $O(p\,q\,r)$ and inverting a matrix of size $p \times p$ has a complexity of $O(p^3)$. Considering that the computation of $P_{k|k-1}(J_h)_k^T$ in step (1) is reused in step (2), the total complexity at an arbitrary step k of the algorithm is the following:

$$\begin{aligned} &\mathrm{C}(\mathbf{f}(\cdot,\cdot)) + \mathrm{C}(\mathbf{h}(\cdot)) + 2n^3 + 2n^2(1+m) \\ &\qquad + n(1 + m + 2m^2) + m(1+m)\,, \end{aligned}$$

and in Big O notation:

$$\mathrm{O}(\mathrm{C}(\mathbf{f}(\cdot,\cdot)) + \mathrm{C}(\mathbf{h}(\cdot)) + n^3 + n^2 m + n m^2)\,. \tag{4}$$

3.6 Track Finding

In the Drift Chambers software, each particle's track is estimated independently via the EKF, process that is dubbed **Track Fitting**. The state vector estimated by the EKF is $\mathbf{x}_k(z)$, and is defined as:

$$\mathbf{x}_k(z) = \begin{pmatrix} x \\ y \\ \theta_x \\ \theta_y \\ q \end{pmatrix},$$

given that:

$$\theta_x = \frac{\mathrm{p}_x}{\mathrm{p}_z}, \theta_y = \frac{\mathrm{p}_y}{\mathrm{p}_z}, q = \frac{Q_e}{\|\mathbf{p}\|},$$

where (x, y) represents the coordinates of the particle measured in a specific plane z, commonly denominated **measurement plane**. A measurement plane z can be thought of as a plane perpendicular to the beam line seen in Figure 4 producing a view similar to the one seen in Figure 11. Unlike traditional methods, z is used as the "time" variable because the speed of each particle in z is constant for this purpose. $\mathbf{p} = (p_x, p_y, p_z)$ is the momentum of the particle in that measurement plane and Q_e is the electric charge of the particle. The equations of motion are:

$$\begin{aligned}
f_x(z, \mathbf{x}) &= \theta_x \\
f_y(z, \mathbf{x}) &= \theta_y \\
f_{\theta_x}(z, \mathbf{x}) &= q\, v \sqrt{1 + \theta_x^2 + \theta_y^2} \\
&\quad \big(\theta_y(\theta_x\, \mathrm{B}_1(x, y, z) + \mathrm{B}_3(x, y, z) - (1 + \theta_x^2)\, \mathrm{B}_2(x, y, z)\big) \\
f_{\theta_y}(z, \mathbf{x}) &= q\, v \sqrt{1 + \theta_x^2 + \theta_y^2} \\
&\quad \big(-\theta_x(\theta_y\ \mathrm{B}_2(x, y, z) + \mathrm{B}_3(x, y, z) + (1 + \theta_y^2)\, \mathrm{B}_1(x, y, z)\big) \\
f_q(z, \mathbf{x}) &= 0\,,
\end{aligned}$$

where $\mathbf{B}(x, y, z) = (\mathrm{B}_1(x, y, z), \mathrm{B}_2(x, y, z), \mathrm{B}_3(x, y, z))$ is the effect of the magnetic field in a particular (x, y, z) position inside the detector and v is the speed of light in the medium. The components of $\mathbf{B}$ are defined as B_1, B_2 and B_3 instead of $\mathrm{B_x}$, $\mathrm{B_y}$ and $\mathrm{B_z}$ to avoid confusion with the notation of partial derivatives. For the EKF to be able to be solved, the gradient of these equations of motion are also obtained:

$$\begin{aligned}
&\frac{\partial f_x}{\partial x} = 0,\ \frac{\partial f_x}{\partial y} = 0,\ \frac{\partial f_x}{\partial \theta_x} = 1,\ \frac{\partial f_x}{\partial \theta_y} = 0,\ \frac{\partial f_x}{\partial q} = 0 \\
&\frac{\partial f_y}{\partial x} = 0,\ \frac{\partial f_y}{\partial y} = 0,\ \frac{\partial f_y}{\partial \theta_x} = 0,\ \frac{\partial f_y}{\partial \theta_y} = 1,\ \frac{\partial f_y}{\partial q} = 0\,,
\end{aligned}$$

$$\frac{\partial f_{\theta_x}}{\partial x} = q\,v\sqrt{\theta_x^2+\theta_y^2+1}$$
$$\left(\theta_y\left(\theta_x\frac{\partial}{\partial x}\,\mathrm{B}_1\left(x,y,z\right)+\frac{\partial}{\partial x}\,\mathrm{B}_3\left(x,y,z\right)\right)+\left(-\theta_x^2-1\right)\frac{\partial}{\partial x}\,\mathrm{B}_2\left(x,y,z\right)\right) \tag{5}$$

$$\frac{\partial f_{\theta_x}}{\partial y} = q\,v\sqrt{\theta_x^2+\theta_y^2+1}$$
$$\left(\theta_y\left(\theta_x\frac{\partial}{\partial y}\,\mathrm{B}_1\left(x,y,z\right)+\frac{\partial}{\partial y}\,\mathrm{B}_3\left(x,y,z\right)\right)+\left(-\theta_x^2-1\right)\frac{\partial}{\partial y}\,\mathrm{B}_2\left(x,y,z\right)\right) \tag{6}$$

$$\frac{\partial f_{\theta_x}}{\partial \theta_x} = \frac{q\,\theta_x\,v\left(\theta_y\left(\theta_x\,\mathrm{B}_1\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)-\left(\theta_x^2+1\right)\mathrm{B}_2\left(x,y,z\right)\right)}{\sqrt{\theta_x^2+\theta_y^2+1}}$$
$$+\,q\,v\left(-2\theta_x\,\mathrm{B}_2\left(x,y,z\right)+\theta_y\,\mathrm{B}_1\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_{\theta_x}}{\partial \theta_y} = \frac{q\,\theta_y\,v\left(\theta_y\left(\theta_x\,\mathrm{B}_1\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)-\left(\theta_x^2+1\right)\mathrm{B}_2\left(x,y,z\right)\right)}{\sqrt{\theta_x^2+\theta_y^2+1}}$$
$$+\,q\,v\left(\theta_x\,\mathrm{B}_1\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_{\theta_x}}{\partial q} = v\left(\theta_y\left(\theta_x\,\mathrm{B}_1\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)-\left(\theta_x^2+1\right)\mathrm{B}_2\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_{\theta_y}}{\partial x} = q\,v\sqrt{\theta_x^2+\theta_y^2+1}$$
$$\left(-\theta_x\left(\theta_y\frac{\partial}{\partial x}\,\mathrm{B}_2\left(x,y,z\right)+\frac{\partial}{\partial x}\,\mathrm{B}_3\left(x,y,z\right)\right)+\left(\theta_y^2+1\right)\frac{\partial}{\partial x}\,\mathrm{B}_1\left(x,y,z\right)\right) \tag{7}$$

$$\frac{\partial f_{\theta_y}}{\partial y} = q\,v\sqrt{\theta_x^2+\theta_y^2+1}$$
$$\left(-\theta_x\left(\theta_y\frac{\partial}{\partial y}\,\mathrm{B}_2\left(x,y,z\right)+\frac{\partial}{\partial y}\,\mathrm{B}_3\left(x,y,z\right)\right)+\left(\theta_y^2+1\right)\frac{\partial}{\partial y}\,\mathrm{B}_1\left(x,y,z\right)\right) \tag{8}$$

$$\frac{\partial f_{\theta_y}}{\partial \theta_x} = \frac{q\,\theta_x\,v\left(-\theta_x\left(\theta_y\,\mathrm{B}_2\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)+\left(\theta_y^2+1\right)\mathrm{B}_1\left(x,y,z\right)\right)}{\sqrt{\theta_x^2+\theta_y^2+1}}$$
$$+\,q\,v\left(-\theta_y\,\mathrm{B}_2\left(x,y,z\right)-\mathrm{B}_3\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_{\theta_y}}{\partial \theta_y} = \frac{q\,\theta_y\,v\left(-\theta_x\left(\theta_y\,\mathrm{B}_2\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)+\left(\theta_y^2+1\right)\mathrm{B}_1\left(x,y,z\right)\right)}{\sqrt{\theta_x^2+\theta_y^2+1}}$$
$$+\,q\,v\left(-\theta_x\,\mathrm{B}_2\left(x,y,z\right)+2\theta_y\,\mathrm{B}_1\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_{\theta_y}}{\partial q} = v\left(-\theta_x\left(\theta_y\,\mathrm{B}_2\left(x,y,z\right)+\mathrm{B}_3\left(x,y,z\right)\right)+\left(\theta_y^2+1\right)\mathrm{B}_1\left(x,y,z\right)\right)\sqrt{\theta_x^2+\theta_y^2+1}$$

$$\frac{\partial f_q}{\partial x}=0,\ \frac{\partial f_q}{\partial y}=0,\ \frac{\partial f_q}{\partial \theta_x}=0,\ \frac{\partial f_q}{\partial \theta_y}=0,\ \frac{\partial f_q}{\partial q}=0\,.$$

It's worth noting that $\frac{\partial \mathbf{B}}{\partial x}$, $\frac{\partial \mathbf{B}}{\partial y}$ and $\frac{\partial \mathbf{B}}{\partial z}$ are in practice very small. Due to this, Equations (5), (6), (7) and (8) are redefined as:

$$\frac{\partial f_{\theta_x}}{\partial x} = 0,\ \frac{\partial f_{\theta_x}}{\partial y} = 0,\ \frac{\partial f_{\theta_y}}{\partial x} = 0,\ \frac{\partial f_{\theta_y}}{\partial y} = 0\,.$$

All of these equations of motions are then solved via Runge Kutta 4 as explained in Section 3.7.

After transporting from one measurement plane to the next, EKF's **Update** phase must be applied. The input state vector is again the same $\mathbf{x_{k|k-1}}$, with its related covariance matrix denoted as $P_{k|k-1}$, and the measurement vector $\mathbf{z}_k$, for each measurement plane z, is defined as:

$$\mathbf{z}_k(z) = \begin{pmatrix} x_m \\ u \\ t \\ l \\ w \end{pmatrix},$$

A measurement results from an effective hit to a particular wire where a local tilted coordinate system is defined in which $y = 0$ for convenience. Each of the scalar components denote different attributes related to the measurement itself:

- x_m: **measured** x **at** z. Note that since y is fixed to 0 and the measurement plane z is known, only x needs to be measured.
- u: **uncertainty of the measurement**. Converting the time to distance from the moment the measurement is taken at the wire, a certain statistical uncertainty is related to the measurement, with a minimum close to 300 [ns].
- t: **measurement tilt**. Tilt used to define the tilted coordinate system which can be either 30° or −30°.
- l: **length of the wire from which the measurement is obtained**. As can be seen in Figure 11 from Section 3.3, each superlayer in each sector has an isosceles trapezoid shape when viewed transverse to the beam line at the target location. This means that wires can have different lengths depending on where on the trapezoid they're strung, so this needs to be considered when using the measurements.
- w: **the wire's maximum sag**. This variable denotes the maximum possible sag caused by the catenary the wire naturally follows due to gravity.

then, vector $\mathbf{h}_k(z)$ is computed:

$$\mathbf{h}_k(z) = \begin{pmatrix} 1 \\ -\tan(t) - \frac{4w}{l} \cdot (1 - \frac{2y}{l}) \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{9}$$

which is a vector used to define the transformation necessary to project the state vector into the local coordinate system of the measurement.

A H_k matrix can then be computed from $\mathbf{h}_k(z)$ to update the covariance matrix:

$$H_k = \mathbf{h}_k \mathbf{h}_k^T ,$$

then, to calculate the filtered covariance matrix $P_{k|k}$:

$$P_{k|k} = \left(P_{k|k-1}^{-1} + \frac{H_k}{|u|} \right)^{-1} ,$$

From this new covariance matrix, a vector $\mathbf{k}_k$ can be computed:

$$\mathbf{k}_k = \frac{P_{k|k} \times \mathbf{h}_k}{|u|} ,$$

using $\mathbf{h}_k(z)$ from Equation (9). From this vector $\mathbf{k}_k$, both the state vector $\mathbf{x}_{k|k}$ and the quadratic error χ_k^2 (described in Appendix 8.2) can be obtained:

$$\mathbf{x}_{k|k} = \mathbf{x}_{k|k-1} + (x_m - (x - h_{k(2)})) \cdot \mathbf{k}_k$$
$$\chi_k^2 = \chi_{k-1}^2 + \frac{(x_m - (x - h_{k(2)}))^2}{|u|} ,$$

where $h_{k(2)}$ is the second scalar component of the vector $\mathbf{h}_k(z)$ from Equation (9). The quadratic error obtained can be used to evaluate how well the fit is going and, when the EKF is finished, to measure the quality of the fit.

The pseudocode for this particular implementation of the EKF for CLAS12 is presented:

Algorithm 1: Kalman Filter

$Z \leftarrow$ array of measurement planes
$m \leftarrow$ array of measurements
$\mathbf{s}_0 \leftarrow$ initial state vector
$S_0 \leftarrow$ initial covariance matrix
$I \leftarrow$ total number of iterations
def run_fitter$(Z, m, \mathbf{s}, S, I)$**:**
    $\chi^2_b \leftarrow \infty$
    $K \leftarrow \text{size}(Z) - 1$
    for $i \leftarrow 0$ **to** I **do**
        $\chi^2_c \leftarrow 0$
        if $i > 0$ **then**
            $\mathbf{s}_0, S_0 \leftarrow \text{transport}(K, 0, \mathbf{s}_K, S_K, \chi^2_c, Z)$
        for $k \leftarrow 0$ **to** K **do**
            $\mathbf{s}_{k+1}, S_{k+1} \leftarrow \text{transport}(k, k+1, \mathbf{s}_k, S_k, Z)$
            $\mathbf{s}_{k+1}, S_{k+1}, \chi^2_c \leftarrow \text{filter}(\mathbf{s}_{k+1}, S_{k+1}, m_{k+1}, \chi^2_c)$
        end
        if $|\mathbf{s}_b.Q - \mathbf{s}_K.Q| < 5 \cdot 10^{-4}$ and $|\mathbf{s}_b.x - \mathbf{s}_K.x| < 10^{-4}$ and $|\mathbf{s}_b.y - \mathbf{s}_K.x| < 10^{-4}$ and $|\mathbf{s}_b.\theta_x - \mathbf{s}_K.\theta_x| < 10^{-6}$ and $|\mathbf{s}_b.\theta_y - \mathbf{s}_K.\theta_y| < 10^{-6}$ **then**
            $i = I$
        if $\chi^2_c < \chi^2_b$ **then**
            $\chi^2_b \leftarrow \chi^2_c$
            $\mathbf{s}_b \leftarrow \mathbf{s}_K$
    end
    return $\mathbf{s}_K$, S_K, χ^2_b

Algorithm 2: Transport

$i \leftarrow$ initial state $(k-1|k-1)$
$f \leftarrow$ final state $(k|k-1)$
$\mathbf{s}_i \leftarrow$ state vector at i
$S_i \leftarrow$ covariance matrix at i
$\chi^2 \leftarrow \chi^2$ error
$Z \leftarrow$ array of measurement planes
def transport$(i, f, \mathbf{s}_i, S_i, \chi^2, Z)$**:**
  $\mathbf{s}_f \leftarrow \mathbf{s}_i$
  $S_f \leftarrow S_i$
  $h \leftarrow 1$ `// step size`
  $z \leftarrow Z_i$
  while $\text{sign}(Z_f - Z_i) \cdot \mathbf{s}_f.z < \text{sign}(Z_f - Z_i) \cdot Z_f$ **do**
    $h_{\text{signed}} \leftarrow \text{sign}(Z_f - Z_i) \cdot h$
    if $\text{sign}(Z_f - Z_i) \cdot (\mathbf{s}_f.z + h_{\text{signed}}) > \text{sign}(Z_f - Z_i) \cdot Z_f$ **then**
      $h_{\text{signed}} \leftarrow \text{sign}(Z_f - Z_i) \cdot |Z_f - \mathbf{s}_f.z|$
    $z, \mathbf{s}_f, S_f, \chi^2 \leftarrow \text{rk4\_transport}(z, h_{\text{signed}}, \mathbf{s}_f, S_f, \chi^2)$
  end
  return $\mathbf{s}_f$, S_f

Algorithm 3: Filter

$\mathbf{s}_i \leftarrow$ state vector at i $(k|k-1)$
$S_i \leftarrow$ covariance matrix at i $(k|k-1)$
$\mathbf{m}_k \leftarrow$ measurement vector at k
$\chi^2 \leftarrow \chi^2$ error before running the filter
def filter$(\mathbf{s}_i, S_i, \mathbf{m}_k, \chi^2)$**:**
  $\mathbf{h} \leftarrow \text{get\_h}(\mathbf{s}_i.y, \mathbf{m}_k.t, \mathbf{m}_k.w, \mathbf{m}_k.l)$
  $S_f \leftarrow S_i - \frac{S_i \mathbf{h}\mathbf{h}^T S_i}{1+\mathbf{h}^T S_i \mathbf{h}}$
  $\mathbf{k} \leftarrow < 0, 0, 0, 0, 0 >$
  for $j \leftarrow 1$ **to** 5 **do**
    $k_j \leftarrow \frac{(h_0 \cdot S_{f(j,0)} + h_1 \cdot S_{f(j,1)})}{|\mathbf{m}_k.u|}$
  end
  $c \leftarrow \text{get\_correction}(\mathbf{s}_i.x, \mathbf{s}_i.y, \mathbf{m}_k.t, \mathbf{m}_k.w, \mathbf{m}_k.l)$
  $\chi^2 \leftarrow \chi^2 + \frac{(\mathbf{m}_k.x - c)^2}{|\mathbf{m}_k.u|}$
  $\mathbf{s}_f \leftarrow \mathbf{s}_i + \mathbf{k} \cdot (\mathbf{m}_k.x - c)$
  return $\mathbf{s}_f$, S_f, χ^2
def get_h(y, t, w, l)**:**
  return $< 1, -\tan(6t) - w \cdot \frac{4}{l} \cdot \left(1 - \frac{2y}{l}\right), 0, 0, 0 >$
def get_correction(x, y, t, w, l)**:**
  return $x - \tan(6t) \cdot y - w \cdot \left(1 - \frac{2y}{l}\right)^2$

3.7 Fourth Order Runge Kutta

The Fourth Order Runge Kutta method or simply Runge Kutta 4 (RK4) is an iterative method used to solve initial value problems, originally proposed by Carl Runge and Wilhelm Kutta [Runge, 1895]. In CLAS12, the method is used to estimate a state vector $\mathbf{x}(z+h)$ between two measurement planes z_{k-1} and z_k, denoted as $z_{k-1}+h$, where h is the step size used, given that $z_k > z_{k-1}+h$, and thus $\mathbf{x}(z+h)$ is between $\mathbf{x}_{k-1|k-1}$ and $\mathbf{x}_{k|k-1}$.

Let an initial value problem for estimating $\mathbf{x}(z+h)$ be:

$$\dot{\mathbf{x}} = \mathbf{f}(z, \mathbf{x}), \mathbf{x}(z_0) = \mathbf{x}_0\,,$$

where $\mathbf{x}$ is the unknown vector function of the measurement plane z to be approximated, $\dot{\mathbf{x}}$ is its rate of change, defined as a function of z and $\mathbf{x}$ itself. $\mathbf{f}$ denotes the equations of motion from Section 3.6 and the initial z_0 and $\mathbf{x}_0$ are given. Using the step size $h > 0$ defined before:

$$\mathbf{x}(z+h) = \mathbf{x}(z) + \frac{1}{6}(\mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3 + \mathbf{r}_4)\,, \tag{10}$$

with $\mathbf{r}_1$, $\mathbf{r}_2$, $\mathbf{r}_3$ and $\mathbf{r}_4$ defined as:

$$\begin{aligned}
\mathbf{r}_1 &= h \cdot \mathbf{f}(z, \mathbf{x}(z))\,,\\
\mathbf{r}_2 &= h \cdot \mathbf{f}\left(z + \frac{h}{2}, \mathbf{x}(z) + \frac{\mathbf{r}_1}{2}\right),\\
\mathbf{r}_3 &= h \cdot \mathbf{f}\left(z + \frac{h}{2}, \mathbf{x}(z) + \frac{\mathbf{r}_2}{2}\right),\\
\mathbf{r}_4 &= h \cdot \mathbf{f}(z+h, \mathbf{x}(z) + \mathbf{r}_3)\,.
\end{aligned}$$

Finally, the $\mathbf{x}(z+h)$ presented in Equation (10) is named the RK4 approximation of $\mathbf{x}(z+h)$ [Sauer, 2012].

The pseudocode for the implementation of RK4 at the DC software is presented:

Algorithm 4: Runge Kutta 4

$z \leftarrow$ initial measurement location
$h \leftarrow$ step size
$\mathbf{s}_k \leftarrow$ state vector at k
$S_k \leftarrow$ covariance matrix at k
$\chi^2 \leftarrow \chi^2$ error
def rk4_transport$(z, h, \mathbf{s}_k, S_k, \chi^2)$**:**
  // r_1
  $\mathbf{B} \leftarrow$ bfield at location
  $\mathbf{k}_{\mathbf{x}(1)} \leftarrow \mathbf{f}_{\mathbf{x}}(z, \mathbf{x}, \mathbf{B})$
  $K_{J(1)} \leftarrow F_J(z, \mathbf{x}, J, \mathbf{B})$
  // r_2
  $\mathbf{B} \leftarrow$ bfield at new location
  $\mathbf{k}_{\mathbf{x}(2)} \leftarrow \mathbf{f}_{\mathbf{x}}(z + \frac{h}{2}, \mathbf{x} + \frac{h}{2}\mathbf{k}_{\mathbf{x}(1)}, \mathbf{B})$
  $K_{J(2)} \leftarrow F_J(z + \frac{h}{2}, \mathbf{x} + \frac{h}{2}\mathbf{k}_{\mathbf{x}(1)}, J + \frac{h}{2}K_{J(1)}, \mathbf{B})$
  // r_3
  $\mathbf{B} \leftarrow$ bfield at new location
  $\mathbf{k}_{\mathbf{x}(3)} \leftarrow \mathbf{f}_{\mathbf{x}}(z + \frac{h}{2}, \mathbf{x} + \frac{h}{2}\mathbf{k}_{\mathbf{x}(2)}, \mathbf{B})$
  $K_{J(3)} \leftarrow F_J(z + \frac{h}{2}, \mathbf{x} + \frac{h}{2}\mathbf{k}_{\mathbf{x}(2)}, J + \frac{h}{2}K_{J(2)}, \mathbf{B})$
  // r_4
  $\mathbf{B} \leftarrow$ bfield at new location
  $\mathbf{k}_{\mathbf{x}(4)} \leftarrow \mathbf{f}_{\mathbf{x}}(z + h, \mathbf{x} + h\mathbf{k}_{\mathbf{x}(3)}, \mathbf{B})$
  $K_{J(4)} \leftarrow F_J(z + h, \mathbf{x} + h\mathbf{k}_{\mathbf{x}(3)}, J + hK_{J(3)}, \mathbf{B})$
  // RK4
  $z \leftarrow z + h$
  $\mathbf{x} \leftarrow \mathbf{x} + \frac{h}{6}(\mathbf{k}_{\mathbf{x}(1)} + 2\mathbf{k}_{\mathbf{x}(2)} + 2\mathbf{k}_{\mathbf{x}(3)} + \mathbf{k}_{\mathbf{x}(4)})$
  $J \leftarrow J + \frac{h}{6}(K_{J(1)} + 2K_{J(2)} + 2K_{J(3)} + K_{J(4)})$
  $S_k \leftarrow$ update_cov_mat(S_k, J)
  $S_k \leftarrow$ q_process_estimate$(S_k, z, h, \mathbf{x}, \text{mass})$
  return $z, \mathbf{x}, S_k, \chi^2$

where $\mathbf{f}_{\mathbf{x}}(z, \mathbf{x}, \mathbf{B})$ and $F_J(z, \mathbf{x}, J, \mathbf{B})$ are the equations of motion described in Section 3.6, `update_cov_mat`(S_k, J) applies the equations of motion of the Jacobian matrix to the covariance matrix and `q_process_estimate`$(S_k, z, h, \mathbf{x}, \text{mass})$ updates it from the estimated noise in q. More detail on this can be seen in the original source code from the repository linked in Appendix 8.14.

CHAPTER 4
CODE PROFILING

Since the project consists on the heavy optimization of the DC code, a logical starting point is a detailed profiling of the compiled program in order to figure out where the performance bottlenecks are found so as to focus work on them. Different profiling programs or **profilers** were evaluated for this purpose, but in the end the default java profiler, or **jvisualvm**, was used. A small analysis into the reasoning behind this decision is shown in Section 4.1 and then the profiling results and a short analysis of them is given in Section 4.2.

Profiling or "software profiling" is a dynamic program analysis method that measures all factors to analyze the performance of an application. These factors were separated into four categories: CPU/GPU usage, memory usage, IO and network operations and database queries. These categories are used because they encompass all the processes that can be commonly found in an application.

A **profiler** is a tool that analyzes either the source code or the running program to report its performance after or while it is running, usually offering information like the program's total memory use, the number of times a certain method was called or the largest bottlenecks in the programs execution [Ball and Larus, 1997]. There are three ways to obtain profiling information: **code instrumentation**, **statistical sampling** and **application monitoring**.

Code instrumentation is the act of placing new lines of code in a program and counts how many times they were ran, so as to give the exact number while the program is running or after it is completed [Ball and Larus, 1994]. Instrumentation can give a very detailed view into exactly how the program is working, but due to the fact that new lines of code are added to the program, the execution time can drastically increase while profiling is taking place. An important terminological consideration is that code instrumentation is usually simply referred to as profiling, which can lead to confusion when studying the area.

In stark contrast, **statistical sampling** is a less disruptive method that work by stopping the program in regular intervals during its execution, taking a so-called "snapshot" and then allowing the application to continue, sacrificing accuracy in favor of speed [Wenisch et al., 2006]. Summarized, sampling may reveal the relative percentage of time spent in frequently-called methods, but cannot provide the exact number of times each method is invoked, unlike instrumentation.

Application monitoring provides a so-called "eagle's point of view" onto a running application, focusing on the high-level picture of it. Monitoring provides very basic information about an application but is the least disruptive of all the methods mentioned,

and thus is mostly used for web services that require reports onto how they're running while also minimizing the disruption this analysis has on the running program so as to not impact its performance [Hunt and John, 2011].

As mentioned before, all of the effort done in this work is based on multiple instances of performance analysis done via the **jvisualvm** tool. Also, all the results given in Section 4.2 and chapter 6 are obtained by measuring the total execution time of the program with the tool in a personal computer, an Intel(R) Core(TM) i5-6600K CPU with 4 cores running at 3.50GHz with 8GB of DDR4 2133MHz RAM.

4.1 Tool Comparison

Due to the large significance of the profiling process and its effects on the final results of any analysis, a detailed analysis of the many tools available to profile applications is performed. They are compared based on four criteria:

- **CPU Profiling**: How accurate and fast are the sampling and profiling processes available in the application, and if the tool can provide an immediate view into the **hotspots** in the application. A hotspot is the common term used to describe a specific method or code segment that takes up a large portion of the total computation time [Hunt and John, 2011].
- **Memory Analysis**: Since the main focus of the profiling process for this project is the analysis of the CPU time used, the only memory analysis systems required for that purpose are the detection of memory leaks and the count of class instances generated since both of these can impact the computation time. A memory leak occurs when object references that are no longer needed are unnecessarily maintained, which is a precise problem related to the java garbage collector [Xu and Rountev, 2008].
- **Remote Profiling**: Considering that the application will run at the computer facilities at JLab, some of the performance analysis should be done in this environment to assert that the results obtained in a personal computer are correlated to the ones found in it. For this, it is crucial that the selected tool allows for remote profiling, so that it is possible to perform analysis locally and remotely.
- **License**: Finally, the cost of the profiling application is evaluated to see if the additional features outweigh this price.

Using each criterion, the tools analyzed are the default java profiler, **jvisualvm**, a set of paid tools, **JProfiler**, **YourKit**, **XRebel** and **JProbe**, and a small set of free applications, including **Callgrind**, **Honest Profiler** and **JIProf**. While this set is far from comprehensive, the tools analyzed are the ones mainly used in the market as reported in [Maple, 2015a] and [Maple, 2015b], along with various scattered sources.

As mentioned before, it was decided to use **jvisualvm** due to price restrictions and the fact that it is a robust tool despite being free, contrasted with the fact that the other applications proposed cost at least $499 USD at the time of publication. Java VisualVM or jvisualvm for short is a graphical user interface that provides tools to profile, monitor and troubleshoot Java applications while they are running on the Java Virtual Machine (JVM) [Oracle, 2019]. The tool was developed by Oracle, the company behind Java, and is provided to programmers for free.

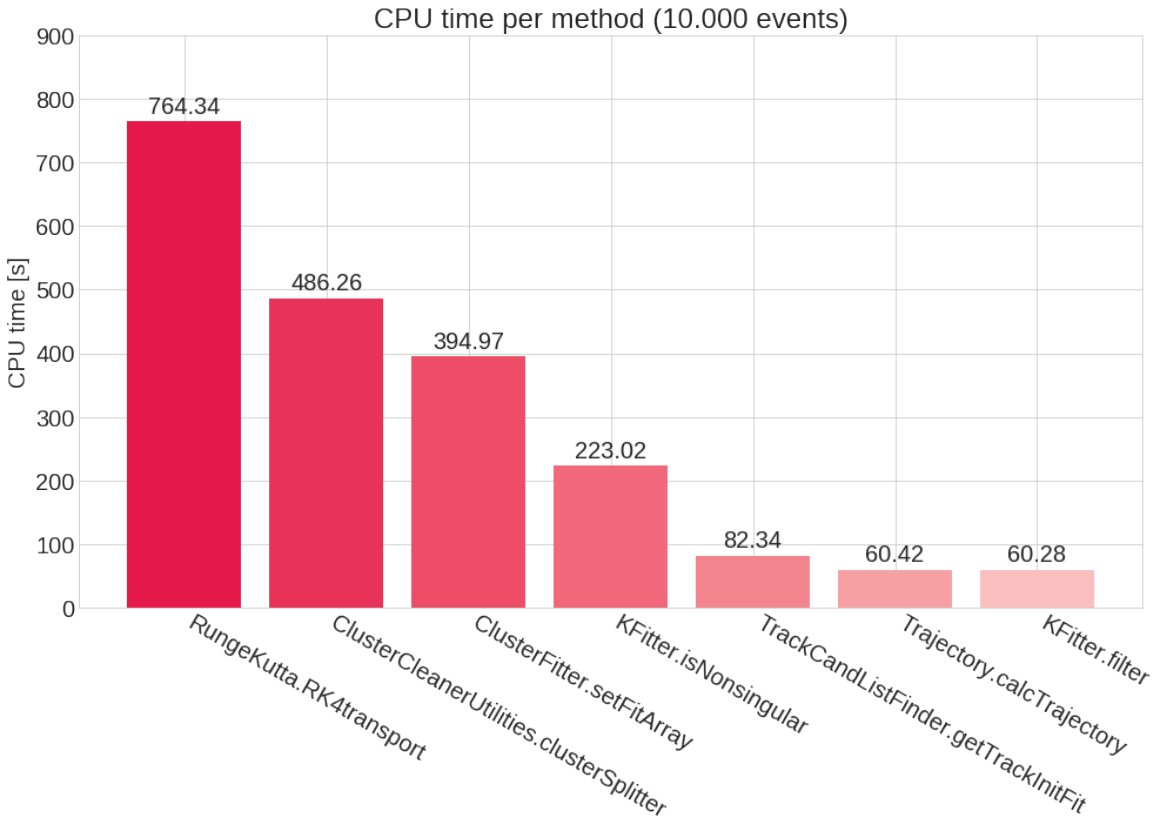

Figure 15: Methods' CPU time on the original source code (version 1.0).

4.2 Results

Using the **jvisualvm** profiler, the time taken by each component of the CLAS12 offline software can be measured, thus finding the bottlenecks in the execution in order to prioritize work on them. It's worth noting that all the profiled data presented in this publication is obtained via **jvisualvm** unless stated otherwise, running the CLAS12 software on 10.000 events with real experimental data taken from the test file `clas_004013.0.hipo`.

As a note before continuing, all figures that relate to profiling information will be always associated to a "version". This is different from the versioning system utilized in the CLAS12 offline software, and is instead one used only in this work to quickly refer to different states of the software after each improvement, and thus, "version 1.0" refers to the initial state of the code, before any changes are made by the author.

The methods taking up more than 2% of the total CPU time before any change is made on the source code is shown in Figure 15. A brief description of the methods in the figure are:

- **Runge Kutta 4**: named `RK4transport` from the `RungeKutta` (`RK4` in Figure 15) class, the method is a direct implementation of the algorithm described in Section 3.7, taking up a total of 35.1% (764.34 seconds) of the total computation time. As is explained in detail in Section 5.3, most of the time used in this method is taken up by a method to obtain the magnetic field at specific points.
- **Cluster Splitter**: named `clusterSplitter` from the `CCU` class, which is short for `ClusterCleanerUtilities`. As the name suggests, it refers to the process of splitting the clusters described in Section 3.4 done via the Hough transform, which in simple terms consists of finding line shaped sub-clusters in the split cluster [Hough, 1962] and is described in detail in Appendix 8.5.

  The process is useful because it splits malformed clusters with poor performance to ones that are better fit, but, as can be seen in Figure 15, takes 22.4% (486.26 seconds) of the total CPU time. This is attributed to the fact that it transforms every hit in the original cluster to a so-called "Hough Space" to find the split clusters, which is a slow process due to the raw number of hits in each cluster and the number of clusters.
- **Set Fit Array**: named `setFitArray` from the `ClusterFitter` (`CF` in the Figure) class, the method receives clusters and creates arrays from the hits in them which are later fit into a line by the `fitCluster` method from the same class. While the algorithm itself involves only sorting the cluster and then adding them to arrays, the method takes so much time (18.2% or 394.97 seconds of the total CPU time) because it's called several times by many methods, especially the

`findHitBasedClusters` method in the `ClusterFinder` class, and it sorts the cluster for each of these calls.

- **Is Nonsingular**: named `isNonsingular` from the class `KFitter` (`KF` in Figure 15), it's a method that returns `true` if a matrix is nonsingular and `false` otherwise, taking up a total of 10.2% (223.02 seconds) of the total computation time. The method takes up this much time because it is called twice every time the `filter` method is called to check if the covariance matrix is singular, and it calculates the determinant of the matrix to do this. It's acceleration is discussed in detail in Section 5.2.

- **Get Track Initial Fit**: named `getTrackInitFit` from the `TCLF` class, short for `TrackCandListFinder`. The method simply provides an initial fit to the Kalman Filter from which to start working, but it takes a total of 3.8% (82.34 seconds) mainly due to lack of good programming practices. As can be seen later in Section 6.1, the method's time was lowered by a large factor simply by refactoring and applying simple optimizations to the code.

- **Calculate Trajectory**: named `calcTrajectory` from the `Trajectory` class (`Traj` in the Figure), this method receives a set of parameters from a track's state vector and, as the name suggests, it calculates and returns an estimated trajectory that the particle described by the vector could take. The method's time of 2.8% (60.42 seconds) is attributed to the fact that it's ran many times in the code.

- **Filter**: represented by the `filter` method in the `KFitter` class, it's an implementation of the filtering component of the Kalman filter, described in detail in Section 3.5. It takes in total of 2.8% (60.28 seconds) of the CLAS12 CPU time and its acceleration is described along with the one for `isNonsingular` in Section 5.2.

It is worth noting that to obtain the measurements given in this section, the CLAS12 code was profiled in **jvisualvm** via statistical sampling with a sampling period of 100 [ms] and by profiling only the packages related to the Drift Chambers (DC): `org.jlab.service.dc` and `org.jlab.rec.dc`. This was done to shorten the profiling sessions time by focusing only on the DC code.

CHAPTER 5
SOLUTION PROPOSAL

The approach used to improve on the software's computing time followed a logical order. First, the algorithm is exhaustively analyzed and profiled to find the bottlenecks in the execution, so as to be able to focus the work on them. Then, the cause for each significant bottleneck is identified and investigated, so as to propose solutions using the state of the art in the field, and after a good understanding of each problem is reached, they are addressed. Finally, thorough validation sessions are attempted so as to be sure that the changes made don't affect negative the software's results and to secure that the computing time is effectively reduced.

5.1 Refactoring and Optimizations

Before any large change or algorithm implementation is considered on the software, it was decided to do a so-called "code-cleanup". This was considered necessary mainly because of two reasons:

First, the current implementation had a low readability, presenting various issues like uneven indentation style, non-compliance with the Java code conventions [Sun Microsystems, 1997], inconsistent and arbitrary variable and method names, among many others. This can be seen in the link to the original repository provided at Section 8.14.

The second reason is the large presence of many unintentional, unmanaged architectural technical debt (technical debts and their classification are explained briefly in Appendix 8.6). Many instances of these debts were found in the DC code, in the form of repeated portions of code, a general lack of error checking and reporting and a general lack of comments in difficult-to-understand sections of code, among others.

Due to both these reasons and simply to improve the author's familiarity with the code, the previously mentioned clean-up was done. The measures implemented in this process are the following:

- **Addition of comments**: A comprehensive list of the segments of code with a non-intuitive purpose was written. A small research was then performed for each point in the list, looking into the math, physics or general algorithms from which it could be derived. When this was found, a comment was written either providing a small explanation or referring to the name of the formula or algorithm used. When this research was unfruitful, the original author of the code was contacted

so that she could provide enough information about the code segment so as to write this explanatory comment (Contact information is given in Appendix 8.14).

Apart from the general comments, a complete JavaDoc documentation was added to the most important classes and methods in the DC code, with sparse documentation added also to hard-to-understand classes or methods in less critical classes.

- **Indentation Consistency**: Considering that the indentation style at the DC software was very inconsistent, this was fixed to improve readability. Now, the "ideal length" of indentations and the use of tab characters or space characters for it has been a long-running debate between programmers, so instead of finding the "best option" for it, the current standard used in most of the CLAS12 codebase was implemented, which simply is four space characters per indent.

- **Line Length Fixes**: While the code doesn't follow a rigorous standard, a line length of 100 characters can be seen somewhat consistently thorough the CLAS12 codebase, sticking to the java conventions [Sun Microsystems, 1997]. This line length is not strictly applied to the DC code, but instead some extremely long ($>$ 300 characters) segments of code are separated into various lines to aid readability.

- **Class, Method and Variables Names**: The standard Java conventions stipulate a consistent camel case naming standard, and while most of the code follows this style, the DC code tends to stray away from it in many ways. These include variables in lower case with or without underscores, variables and methods starting with an underscore, methods starting with uppercase, etc. The standard naming convention is applied to all of these instances.

5.1.1 Refactoring

By definition, refactoring is the art of safely improving the design of existing code without changing its behaviour. Changes done to the code through refactoring do not add new functionalities or represent design improvements, but only work to eliminate or reduce **code *smells***. Code *smells* are warning signs about potential problems in the code, meaning points that are worthy of a revision [Wake, 2004].

A small list of some the code *smells* found in the DC code together with examples and the treatment done follows. It is worth noting that this list is far from comprehensive, and just lists the *smells* relevant to the discussion:

- **Long Methods**: Many methods with a very large amount of lines can be found thorough the code. In a normal situation this wouldn't be considered to be a big issue, but in the case of DC, many of these large methods contain repeated code

or very similar segments, which in itself is considered a problem since it breeds inconsistencies and longer implementation times to any change done to the code.

An example of this is the method `RK4transport` from the class `RungeKutta`, from the package `org.jlab.rec.dc.track.fit`, which is 269 lines long. It's worth mentioning that the algorithm implemented in this method is **Runge Kutta 4**, and is described in Section 3.7.

The treatment for this problem is easy to implement, and involves simply splitting the method. What is done to do this is to extract the repeating segments into their own private methods, allowing then the original method to call these new methods. The benefit of this change is the elimination of duplicated code and a large performance improvement due to the elimination of unnecessary loops, which is explained further in Section 5.1.2.

In the mentioned example, the method can be split into various methods: First, the computation of k_1, k_2, k_3 and k_4 can all be done in the same method, reducing 172 lines of code into 4 method calls, with the new method being only 45 lines long. After this, the final $RK4$ computation can be left as is, and some extra steps done to compute the covariance matrix can all be split into their own methods, easing the future debugging of the method and allowing each computation to be changed or optimized separately. The final `RK4transport` method is left with a total of 33 lines of code with 7 method calls. It's worth noting that this number can be further reduced to 20 lines by packing intermediary variables into arrays, but to avoid the danger of reducing code readability it was considered a good idea to avoid this option.

- **Long Parameter Lists**: Similar to the last *smell*, long parameter lists can be found thorough the DC code. In most cases specific to the software in study, these lists are expressed in the form of a long list of inputs for methods. An example for this is the `getTrackInitFit` method in the class `TrackCandListFinder` contained in the `org.jlab.rec.dc.track` package, where 20 parameters are given to the method. These parameters are the integer `sector`, a long list of doubles: `x1`, `y1`, `z1`, `x2`, `y2`, `z2`, `x3`, `y3`, `z3`, `ux`, `uy`, `uz`, `thX`, `thY`, `theta1`, `theta3`, `iBdl`, `TORSCALE`, and an instance of the `Swim` class.

  Fixing such issues is usually easy, just requiring some degree of packaging for the parameters. For the example given, the previosly mentioned method is called by `GetTrackCands`, another method from the same class. In this method, three **cross** objects are unpacked to obtain the first 14 doubles from the mentioned list, with two more doubles coming from clusters contained in the crosses and finally `iBdl` coming from the trajectory formed by the crosses. These **cross** objects are sets of two clusters from adjacent superlayers, while the `iBdl` variables denotes the integral of the magnetic field. The `sector` integer mentioned can also be obtained from the first cross. As can be seen, the 20 input parameters given to the method can be replaced with three `cross` objects, the `TORSCALE` value and

the instance of the `Swim` class previously mentioned, reducing the parameter list to four objects and only one parameter.

- **Commented and Dead Code**: This *smell* consists in the presence of commented code, and is usually caused by either old code that the programmer decided to leave commented instead of deleting in case that it becomes useful in the future. Dead code is the phenomena that occurs when new methods are written and old and unused methods are never deleted. Both commented and dead code segments are present in large quantities in the DC software, and after a meeting between the author and the programmer it was decided to simply remove all this code since analyzing which segments would be useful or not would probably be too large a task for the minimal benefit provided by doing it.

- **Duplicated Code**: This *smell* is characterized by two or more code fragments that look almost or completely identical. Duplication usually occurs when multiple programmers are working on different parts of the same program at the same time, but in the case of the DC software this can be attributed to accidents. The solution to this problem is fairly simple: extract the duplicated code, create a new method to run this code and have both instances call this new method.

  In the case of the DC software, there is one instance where code duplication is very easily spotted; both the `RoadFinder` class from the `org.jlab.rec.dc.trajectory` package and the `CrossListFinder.TrajectoryParametriz` class from the `org.jlab.rec.dc.cross` package contain the exact same method, even using the same name, `evaluate`, and the code in this method does the same: fit a set of points into a line and return the parameters of the fit along with the χ^2 probability and the number of degrees of freedom.

- **Data Class**: The final *smell* that will be discussed in this work is the so-called "data class" *smell*, which refers to a class that contains only fields and setters and getters to access them. In the case of the DC codebase, an example of this would be both the `StateVec` and the `CovMat` classes from the `org.jlab.rec.dc.track.fit` package. Both classes just define a state vector (described in Section 3.6) and a covariance matrix (described in Appendix 8.3), without providing any methods apart from the basic definition of the concepts.

  The treatment for this problem is through the encapsulation of the data class inside another class, ideally one that uses it. In the example, both these classes were encapsulated in another class named `StateVecs` from the same package.

5.1.2 Optimizations

While cleaning up and refactoring the code, many optimizations were applied to the critical areas described in Section 4.2 so as to accelerate the bottlenecks and speed up

the whole CLAS12 software. These optimizations involve the addition of variables to precompute results instead of calculating them several times, the reduction of complexity orders by removing unnecessary loops, and the improvements brought by simply restructuring the code as described in the last Section.

5.2 Matrices Handling Changes

Based on the profiling analysis shown in Section 6.1, This section focuses on reducing the CPU time of both the `isNonSingular` and the `filter` methods, which are explained in the **Update** segment of Section 3.5. For the calculation of the computational cost of mathematical operations, it is assumed that all "basic" operations: summation, subtraction, multiplication and division have a cost of $O(1)$. The reasoning behind this decision is described in Appendix 8.1.

It's worth noting that a very usual approach for accelerating matrix operations at the time of writing is leveraging the computation to GPUs, and it has been commonly used in particular to accelerate the KF itself [Huang et al., 2011, Blattner and Yang, 2011]. This approach is not considered useful for CLAS12 because the matrices that are being handled are very small (the covariance matrix is only 5×5) and it is easy to realize that the time used to leverage the matrices to a GPU would likely make the software slower than the current implementation due to the time consumed in the memory transfer overhead [Moravánszky, 2003].

In the same spirit, neither the Strassen [Strassen, 1969] or the Coppersmith-Winograd [Coppersmith and Winograd, 1990] algorithms are considered to replace matrix multiplications or inversions since they're only useful for large matrices [Ballard et al., 2012]. As an example, the actual amount of operations required for computing the Strassen algorithm is $n^2 + n^{2.807}$, which for the case of a 5×5 matrix is ~ 117 contrasted with the 125 required by the naïve implementation and adding additional numerical instability [Ballard et al., 2012]. This reduction is small when compared with the specific optimizations that are described below, which eliminate matrix multiplication altogether for the methods brought into attention.

5.2.1 Matrix Inverse

To reduce the time spent by the update phase or filter method, the effort was mainly focused in the reduction of the time it takes to calculate the covariance matrix *a posteriori* as seen in Equation (3). The specific implementation of the equation presented in that section for the DC software is the following:

First, the determinant of the covariance matrix $P_{k|k-1}$ is computed and compared with 0 to drop the track if the matrix is singular or near-singular. Then, using the measurement data described in Section 3.6, the vector $\mathbf{h}_k$ is defined as:

$$\mathbf{h}_k = \begin{pmatrix} 1 \\ -\tan(6t) - \frac{4w}{l}(1 - \frac{2y}{l}) \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{11}$$

which is simply defined to project the state vector and covariance matrix to the tilted coordinate system from which the measurement is obtained. From this, the matrix $P_{k|k}$ is computed as:

$$P_{k|k} = (P_{k|k-1}^{-1} + H_k)^{-1}\,, \tag{12}$$

where $H_k = \mathbf{h}_k^T \mathbf{h}_k$. Finally, the $P_{k|k}$'s determinant is compared with 0, again to drop the track if the new covariance matrix is singular.

To reduce the computation time of this calculation, the **Sherman-Morrison formula** (described in Appendix 8.7) can be applied as long as $|P_{k|k-1}| \neq 0$ [Sherman and Morrison, 1950]. Applying the formula to Equation (12):

$$P_{k|k} = P_{k|k-1} - \frac{P_{k|k-1}\mathbf{h}_k\mathbf{h}_k^T P_{k|k-1}}{1 + \mathbf{h}_k^T P_{k|k-1}\mathbf{h}_k}\,. \tag{13}$$

This helps to heavily reduce the total computation time since, as can be seen in the equation, no matrix inversions are required, which are very heavy operations [Fatahalian et al., 2004]. It's worth noting that for this approach to be effective, the order in which the matrix-matrix, matrix-vector and vector-matrix multiplications in Equation (13) are applied is of utmost importance, so that no matrix-matrix multiplications are required. This is done by first multiplying $P_{k|k-1} \cdot \mathbf{h}_k$ and $\mathbf{h}_k^T \cdot P_{k|k-1}$ in the fraction's numerator.

5.2.2 Determinant Calculation

To reduce the time spent on checking if the covariance matrices $P_{k|k-1}$ and $P_{k|k}$ are singular, various paths were taken. First, two alternative ways for computing the matrix's determinant were proposed: The **Su-Chang formula** [Su and Chang, 1996] and a method based on the Cholesky decomposition [van de Geijn, 2011] of the matrix:

- **Su-Chang Formula**: The Su-Chang formula is a recursive algorithm that can be used to compute the determinant of a square matrix. It is explained in detail in Appendix 8.8. As is described by Equation (19) in the appendix, the number of operations required is $\frac{1}{3}(2n^3 - 3n^2 + 7n - 18)$, in contrast with the usual n^3 operations needed to directly compute the determinant using a LU decomposition [Banachiewicz, 1937]. In the particular case of the KF from CLAS12, considering the covariance matrix's size is of size 5×5, the total number of computations required by the Su-Chang formula is 64, while the direct computation of a determinant using the LU decomposition is 125.
- **Cholesky Decomposition**: Over the real number field, the Cholesky decomposition or factorization is the decomposition of a positive-definite matrix into the product of a lower triangular matrix with its transpose, described in detail in Appendix 8.9. A naïve algorithm implementing the Cholesky decomposition

requires $\frac{1}{6}n(2n^2+3n+1)$ operations, with $n-1$ additional operations to multiply the matrix's diagonal and 1 to compute its square, which leaves the number of operations required at $\frac{1}{6}n(2n^2+3n+7)$. For the studied case with $n=5$, the total number of computations required to obtain the matrix's determinant is 60.

Besides the computation of these three methods separately, another option was evaluated: Hardcoding each alternative. Considering that the methods only work with a fixed size input matrix (5×5), it is possible to embed the direct calculation, Su-Chang formula, and Cholesky decomposition methods into the code and optimize the operations required as much as possible. This optimization was done via reading the expression in algebraic form and counting all the multiplications between 2 or more variables, returning a list with each one of these combinations along with how many times they were repeated, thus allowing the programmer to add the precomputation of these operations to the code.

It's worth noting that other efforts in this same field exist and are well-known [Pelegri-Llopart and Graham, 1988, Aho et al., 1989] along with their implementations [Fraser et al., 1991]. This is because the problem is an instance of the **C-Reachability problem**, an extension of the **Reachability** problem often used in compiler construction theory, defined as:

Given a computational (potentially infinite state) system with a set of allowed rules or transformations with a given associated cost, find the set of rules with the smallest associated cost that reaches a certain state of the system from a given initial state [Pelegri-Llopart and Graham, 1988].

Despite the fact that designing an implementation of a solution to this problem might help to further optimize the determinant calculation, this was ultimately considered excessive considering the scope of the problem and in view of the results presented at the end of this section, where the over-optimization starts suffering from setbacks related to the **Java HotSpot Performance Engine** [Meloan, 1999].

Then, the fact that computing the determinants of both $P_{k|k-1}$ and $P_{k|k}$ is needed is disputed. There are four possible cases to be analyzed:

(I) $|P_{k|k}| = 0 \big| |P_{k|k-1}| = 0,$

(II) $|P_{k|k}| = 0 \big| |P_{k|k-1}| \neq 0,$

(III) $|P_{k|k}| \neq 0 \big| |P_{k|k-1}| = 0,$

(IV) $|P_{k|k}| \neq 0 \big| |P_{k|k-1}| \neq 0.$

If it's possible to prove that cases (II) and (III) are invalid, then it's clear that only one of the two determinants needs to be computed to assert that both matrices are non-singular and thus effectively half the computation time required to run these checks.

Based on Equation (12), the determinant of the covariance matrix $P_{k|k}$ can be defined as:

$$\begin{aligned}|P_{k|k}| &= \left|P_{k|k-1}^{-1} + \frac{H_k}{u}\right|^{-1} \\ &= \left|P_{k|k-1}^{-1} \cdot (I + P_{k|k-1} \cdot \frac{H_k}{u})\right|^{-1} \\ &= |P_{k|k-1}| \cdot \left|I + P_{k|k-1} \cdot \frac{H_k}{u}\right|^{-1},\end{aligned}$$

Then, considering that matrix I is non-singular by definition and that $P_{k|k-1} \cdot \frac{H_k}{u}$ can be expressed as the product of two vectors $\mathbf{v}^T\mathbf{u}$, the matrix determinant lemma is used (described in Appendix 8.10):

$$\begin{aligned}&= |P_{k|k-1}| \cdot ((1 + \mathbf{v}^T \cdot I^{-1} \cdot \mathbf{u})|I|)^{-1} \\ &= |P_{k|k-1}| \cdot ((1 + \mathbf{v}^T \cdot I \cdot \mathbf{u}))^{-1} \\ &= |P_{k|k-1}| \cdot (1 + \mathbf{v}^T \cdot \mathbf{u})^{-1}, \end{aligned} \tag{14}$$

where, using p_{ij} to denote the j's column of the i's row of matrix $P_{k|k-1}$ and $h_{k(2)}$ denotes $-\tan(6t) - \frac{4w}{l}(1 - \frac{2y}{l})$ (from Equation (11)), $\mathbf{u}$ and $\mathbf{v}$ can be defined from:

$$\begin{aligned}P_{k|k-1} \cdot \frac{H}{u} &= \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} \\ p_{12} & p_{22} & p_{23} & p_{24} & p_{25} \\ p_{13} & p_{23} & p_{33} & p_{34} & p_{35} \\ p_{14} & p_{24} & p_{34} & p_{44} & p_{45} \\ p_{15} & p_{25} & p_{35} & p_{45} & p_{55} \end{bmatrix} \times \begin{bmatrix} \frac{1}{u} & \frac{h_{k(2)}}{u} & 0 & 0 & 0 \\ \frac{h_{k(2)}}{u} & \frac{h_{k(2)}^2}{u} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\ &= \frac{\begin{bmatrix} p_{11} + p_{12}h_{k(2)} & p_{11}h_{k(2)} + p_{12}h_{k(2)}^2 & 0 & 0 & 0 \\ p_{12} + p_{22}h_{k(2)} & p_{12}h_{k(2)} + p_{22}h_{k(2)}^2 & 0 & 0 & 0 \\ p_{13} + p_{23}h_{k(2)} & p_{13}h_{k(2)} + p_{23}h_{k(2)}^2 & 0 & 0 & 0 \\ p_{14} + p_{24}h_{k(2)} & p_{14}h_{k(2)} + p_{24}h_{k(2)}^2 & 0 & 0 & 0 \\ p_{15} + p_{25}h_{k(2)} & p_{15}h_{k(2)} + p_{25}h_{k(2)}^2 & 0 & 0 & 0 \end{bmatrix}}{u} \\ &= \mathbf{u} \cdot \mathbf{v}^T,\end{aligned}$$

where:

$$\begin{aligned}\mathbf{u}^T &= \left\langle p_{11} + p_{12}\, h_{k(2)}, p_{12} + p_{22}\, h_{k(2)}, p_{13} + p_{23}\, h_{k(2)}, p_{14} + p_{24}\, h_{k(2)}, p_{15} + p_{25}\, h_{k(2)} \right\rangle \\ \mathbf{v}^T &= \left\langle \frac{1}{u}, \frac{h_{k(2)}}{u}, 0, 0, 0 \right\rangle,\end{aligned}$$

then, continuing from Equation (14):

$$\begin{aligned}|P_{k|k}| &= |P_{k|k-1}| \cdot (1 + \mathbf{v}^T \cdot \mathbf{u})^{-1} \\ &= |P_{k|k-1}| \cdot \left(\frac{p_{22}h_{k(2)}^2 + 2p_{12}h_{k(2)} + p_{11}}{u} + 1\right)^{-1},\end{aligned}$$

and finally:

$$|P_{k|k}| = \frac{|P_{k|k-1}| \cdot u}{p_{22}h_{k(2)}^2 + 2p_{12}h_{k(2)} + p_{11} + u}\,. \tag{15}$$

From Equation (15), considering that u is the uncertainty of the measurement and cannot be lesser than ~ 300 [ms] as is explained in Section 3.6, conditions (II) and (III) will be impossible if:

$$p_{22}h_{k(2)}^2 + 2p_{12}h_{k(2)} + p_{11} \geq 0\,. \tag{16}$$

To prove this, the fact that square real numbers are always positive is used:

$$\begin{aligned} 0 &\leq \left(\sqrt{p_{22}}h_{k(2)} + \sqrt{p_{11}}\right)^2 \\ &= p_{22}\,h_{k(2)}^2 + 2h\sqrt{p_{11}\,p_{22}}h_{k(2)} + p_{11}\,, \end{aligned}$$

then, using the Cauchy-Schwarz inequality (explained in Appendix 8.11):

$$\leq p_{22}\,h_{k(2)}^2 + 2h_{k(2)}\,|p_{12}| + p_{11}\,. \tag{17}$$

On a similar vein, it can be proven that:

$$\begin{aligned} 0 &\leq \left(\sqrt{p_{22}}h_{k(2)} - \sqrt{p_{11}}\right)^2 \\ &= p_{22}\,h_{k(2)}^2 - 2h\sqrt{p_{11}\,p_{22}}h_{k(2)} + p_{11} \\ &\leq p_{22}\,h_{k(2)}^2 - 2h_{k(2)}\,|p_{12}| + p_{11}\,. \end{aligned} \tag{18}$$

Taking Equations (17) and (18) into account, it can be seen that:

$$0 \leq p_{22}\,h_{k(2)}^2 + 2h_{k(2)}\,p_{12} + p_{11}\,,$$

thus proving Equation (16).

Finally, it is noted that Equation (15) can be used to compute the second determinant while running the **Update** part of the KF in a manner much faster than any of the other methods mentioned in this section, running only 9 operations given that $|P_{k|k-1}|$ is already known (5 multiplications, 3 additions and 1 division).

Table 1 presents the total CPU time used up by each method for computing the determinant presented in this section. In the first row the CPU time taken by computing both determinants is shown, as was done in the original formulation of the software, while the second row reflects the computation of only the first determinant, and the third the use of Equation (15) to compute the second determinant. The columns in the table are separated into two categories, the naïve and the hardcoded implementation of the algorithms. Inside both categories the direct computation, the Su-Chang formula and the Cholesky decomposition-based methods are displayed for comparison.

		Both determinants	First determinant	Special Method
Naïve	Direct	121.68*	111.17	117.27
	Su-Chang	151.59	129.40	144.07
	Cholesky	117.21	102.73	108.84
Hardcoded	Direct	104.86	109.02	106.19
	Su-Chang	103.45	106.55	104.34
	Cholesky	108.10	104.94	**102.09**

Source: own elaboration.

Table 1: CPU time [s] comparison between different determinant calculation methods.

The time the determinant calculation takes in the original software before any changes are applied is marked with an asterisk, and the best time found is marked in bold. The numbers presented are in seconds [s] and reflect the total time used by each method in the computation of the first $5,000$ events of the test file `clas_004013.0.hipo` measured with the method and in the hardware described in Section 3.7. Various conclusions can be drawn from the table:

First, the total time taken by various methods seem to be very inconsistent with the expected improvements. This is attributed to the fact that Java optimizes dynamically using the **Java HotSpot Performance Engine**, which by design optimizes the parts of the code that are running slowly in run-time. This comes with the counter-intuitive disadvantage that optimized code can actually end up running slower due to it being less of a priority to the engine, to which is added that the engine is better at accelerating simple code rather than code that is already hardcoded and optimized [Meloan, 1999]. This performance is contrasted with the results of the hardcoded determinant calculation algorithms in C, where times of 118.98, 52.34 and 39.56 nanoseconds per 5×5 matrix are reported for the direct, Su-Chang and Cholesky implementations.

Then, attention is drawn to the fact that the worst times are associated to the naïve Su-Chang method, but the hardcoded version presents much better results. This is attributed to the fact that the method requires the initialization of various Matrix objects (3 per step in the determinant calculation), operations that weren't considered expensive in the calculation of the method's operation count but that seem to be in the matrix package utilized in CLAS12, **JAMA** [Hicklin et al., 2000].

Finally, the method selected is the one where the first determinant is computed via the hardcoded Cholesky decomposition-based determinant calculation, while the second is computed from Equation (15).

5.3 Magnetic Field Interpolation

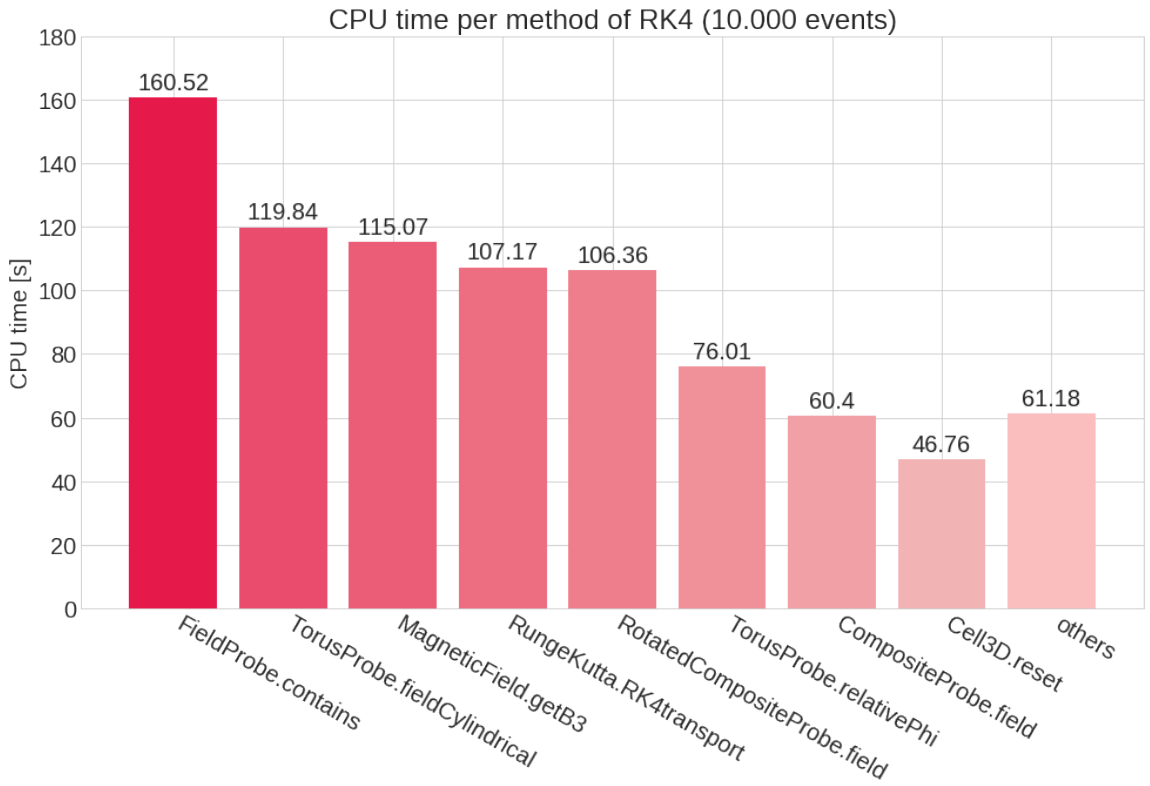

Figure 16: CPU times consumed by each Runge Kutta 4 component.

As seen in Figure 16, most of the Runge Kutta 4 algorithm's runtime is actually used by methods retrieving the magnetic field from the `swimmer` package. These methods' purposes is to compute the magnetic field $\mathbf{B}(x, y, z) = \langle \mathrm{B}_1(x, y, z), \mathrm{B}_2(x, y, z), \mathrm{B}_3(x, y, z) \rangle$ at a specific (x, y, z) point inside the CLAS12 detector, which is done by placing virtual "probes" inside it.

To accelerate this computation, the possibility of using trilinear interpolation [Bourke, 1999] for $\mathbf{B}$ was analyzed. The option was considered favorable since the imprecision brought by interpolating the value instead of directly computing it could be diminished by the **update** part of the KF and by only using the interpolated magnetic field in a fixed number of iterations of the KF before using the real magnetic field for the remaining ones. Appendix 8.12 describes how trilinear interpolation works.

The algorithm used is a direct transcription of the definition of trilinear interpolation, and is defined in two steps. First, obtain the regular grid and the measurements at each border, which is done by selecting a grid size $([\min_x, \max_x], [\min_y, \max_y], [\min_z, \max_z])$ from which the magnetic field measurements are taken with step sizes defined as $\{\mathrm{ss}_x, \mathrm{ss}_y, \mathrm{ss}_z\}$. These parameters define a regular tridimensional grid and a measurement for the magnetic field for each point in this grid is obtained.

For the second step, the specific measurement in the location desired is obtained by evaluating the formula defined in 8.12. In the case that the location is outside the range of the regular grid, the real magnetic field is computed for that point.

After the interpolation algorithm was implemented, the borders and the step size of the grid were left as parameters to be tuned. First, $[\min_z, \max_z]$ were set to $[229\text{cm}, 569\text{cm}]$ since these are the limits of the z variable in the detector.

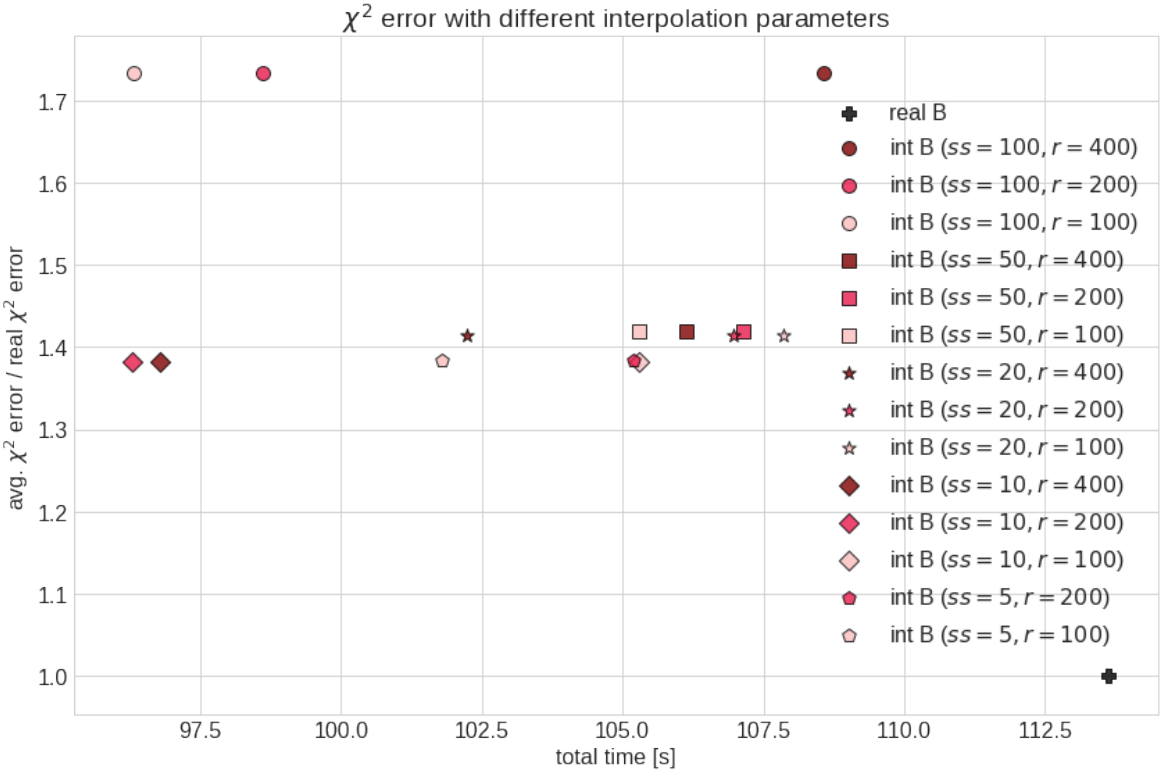

Figure 17: χ^2 error when doing the track fitting with the interpolated magnetic field divided by the obtained when using the real magnetic field vs required time. "real B" is the real magnetic field while "int B" is the interpolated magnetic field with its parameters in parentheses.

To set the step size for each variable, ss is defined such that $\text{ss} = \text{ss}_x = \text{ss}_y = \text{ss}_z$ for simplicity. Then, the range $[\min, \max]$ and the variable r are defined such that $\min = \min_x = \min_y$ and $\max = \max_x = \max_y$ and $r = \max = \text{-min}$, to take advantage of the symmetries of the detector, which can be seen in Figures 4 and 11 from Section 3.3.

Tests are ran by running the software using the interpolated magnetic field with different combinations of parameters, comparing them using the division of the χ^2 error obtained by the one found when running the real field and measuring it against the time it takes to run the program. The results are averaged over a set of 2.500 estimated tracks and the results can be seen in Figure 17. It is worth noting that some outliers are removed

from the figure to aid with visualization. The tested values for each were the following:

$$\text{ss} \in \{5\text{cm}, 10\text{cm}, 20\text{cm}, 50\text{cm}, 100\text{cm}, 200\text{cm}\}$$
$$r \in \{100\text{cm}, 200\text{cm}, 400\text{cm}\}$$

As seen in the figure, the interpolation parameters that provide the best results are $\text{ss} = 10\text{cm}$ and $r = 200\text{cm}$.

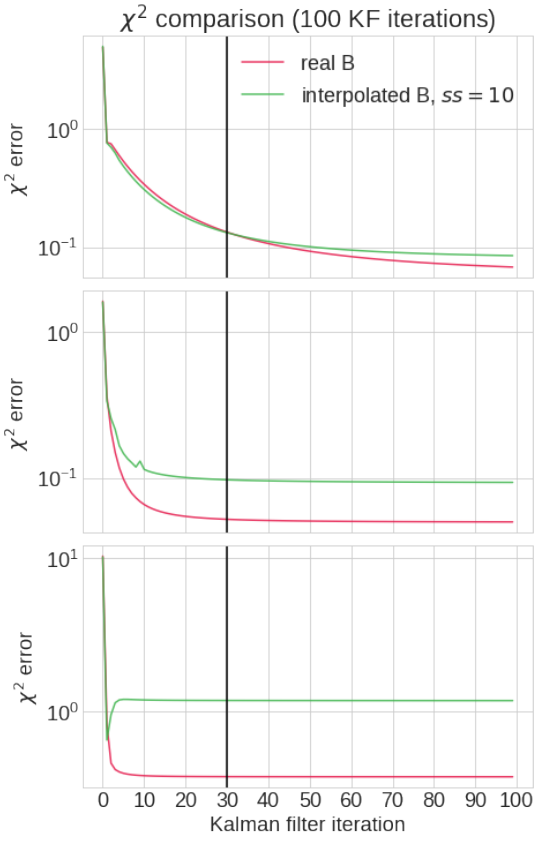

Figure 18: χ^2 error comparison between real magnetic field and interpolated one with a step size of 10 running the KF with 100 iterations. A vertical black line denotes the 30 usual KF iterations.

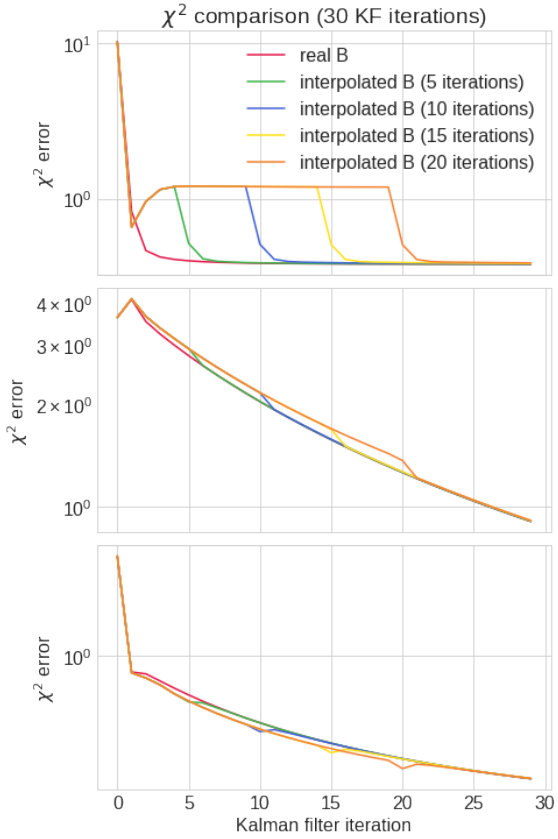

Figure 19: χ^2 error comparison between the real magnetic field and the interpolated one for different numbers of iterations running the KF with 30 iterations.

Figure 18 then shows the χ^2 error of three distinct tracks running 100 KF iterations, contrasting the error of the real magnetic field to that of the selected interpolated version. As can be seen the figure, in most cases both version converge similarly but the one using the real magnetic field provides better results, albeit generally taking up more time.

Finally, a compromise that still provides good results is reached by running only the first KF iterations using the interpolated magnetic field and the last ones using the real

one. The number of iterations using the interpolated field tested are 0, 5, 10, 15, 20, 25 and 30, but at the end all but the last one end up with the same final χ^2 error. Due to this, the number of interpolated magnetic field steps is decided based on the total DCHB and DCTB times when using each version, as presented in Table 2.

It is worth noting that the total times per event listed here are slightly different than the ones presented in the validations of the method, and this is because these tests were ran on 3.000 events instead of the usual 10.000. This decision is taken for the sake of expediency and because it shouldn't affect the results in a significant manner.

Name	DCHB time [s]	DCTB time [s]
Real magnetic field	130.42	48.78
5 interpolated steps	133.42	52.05
10 interpolated steps	110.94	42.24
15 interpolated steps	107.87	40.76
20 interpolated steps	103.83	38.46
25 interpolated steps	109.52	40.14
30 interpolated steps	96.63	35.14

Table 2: DCHB and DCTB times [s] using different interpolated steps.

As can be seen in Table 2, the best times were obtained by running 30 interpolated steps, but the final number of interpolated steps set in the code are 20, the second best. This decision is taken due to the fact that, as can be seen in Figure 19, if no steps are taken with the real magnetic field, the final χ^2 error is much higher than the original one. Figure 19 shows the χ^2 error obtained by using this method.

It's worth noting that the interpolation is only implemented in the Hit-Based tracking part of the KF since the Time-Based part requires as much precision as possible.

5.4 Multithreaded Cluster and Track Finding Algorithms

A multithreaded algorithm is an algorithm that takes advantage of the fact that most modern CPUs have more than one core, thus allowing them to concurrently run various execution threads on these multiple cores in order to take less time processing the same data. This is especially useful in the context that the CLAS12 software is ran at JLab, where many data events are computed concurrently, in different machines. To understand why a multithreaded approach is useful, a bit of insight is needed on how the work in the CLAS12 execution hardware is distributed:

First, a number of threads per engine T is defined in the CLARA configuration file, allowing for T instances of the same engine to process different data events in the different cores at the same time. Then, if the number of engines running in the production

chain is E, at most a number of $T \cdot E$ threads will be running at the same time in the HPC cluster. Dubbing the number of cores at the cluster C, a well-tuned configuration file is set so that $T \cdot E \approx C$ so as to utilize the cluster's capacity as best as possible.

Ideally, this setup would work in such a way that all the C cores remain constantly running, but a problem arises due to the fact that the CLAS12 engines require to run in a sequential manner and, as seen in Figure 5 from Section 2.2, each engine takes a vastly different amount of time to run. Because of this, the cores running the faster engines will be left waiting for the slower ones that come previous to them to finish their execution.

Based on this reasoning, a valid approach for accelerating the software is to distribute the work done in the slowest engines, DCHB and DCTB, onto more threads so that the computing time is improved and a better use of the available hardware is ensured. Two methods for taking advantage of this idea are proposed: A multithreaded algorithm to use in cluster finding and one to use in track finding.

Multithreaded Cluster Finding Algorithm: Analyzing the cluster finding algorithm described in Section 3.4, it can be seen that there is no reason to do the **Clump Finding** step before the **Hit Prunning** one, considering that the second iterates through the hits on a layer-by-layer basis instead of based on the clumps found by the first.

Then, the **Out-of-Timers Removal**, the **Cluster Fitting** and the **Cluster Splitting** steps can be ran in parallel by assigning a thread to each clump found. It is in theory possible to further parallelize the algorithm by assigning individual threads to each cluster found in the Cluster Splitting step and running the fitting done to them separately, but the estimated gain in computing time is very small due to the fact that only one fit is done on the split clusters, so the option was discarded.

Multithreaded Track Finding Algorithm: The track finding done in the DC code described in Section 3.6 requires a set of measurements to run, which are taken from the hits in six different clusters. Due to the geometry of the DC detector (which can be seen in Figure 4 from Section 2.1), finding pairs of clusters (dubbed crosses) in the two superlayers that conform a sector is trivial.

Then, all the combinations of three crosses that could potentially make sense as being formed by a particle's trajectory are considered different, independent tracks. Via the Kalman filter, a fit is attempted for each of these tracks to find the ones that are most likely to be formed by an actual particle passing through the detector. Considering that each track is fit separately and that they are independent from each other, a multithreaded algorithm is implemented where all the tracks in a data event are fit concurrently in different threads.

5.5 Multithreaded Kalman Filter Algorithm Proof of Concept

A multithreaded Extended Kalman Filter (EKF) variant is proposed to accelerate the computation of the EKF in the code and to potentially obtain better solutions. To understand why this option is deemed useful, it has to be noted that the EKF implementation described in Section 3.6 contains a hardcoded stopping point for the EKF after 30 iterations, with no dynamic analysis onto how good the solution found up to that point is or if the methods used have already converged. Changing this might open the possibility of a multithreaded algorithm.

Due to this, the first efforts were placed on finding this dynamic stopping point for the algorithm. Naturally, this criteria must be related to the χ^2 error, which is the variable that is being minimized by the method. As mentioned before, a deifnition of the χ^2 error is given in Appendix 8.2.

The problem with this approach is that for every iteration of the EKF only the so-called "local" χ^2 error is computed, while the "global" one is only computed once the EKF has finished.

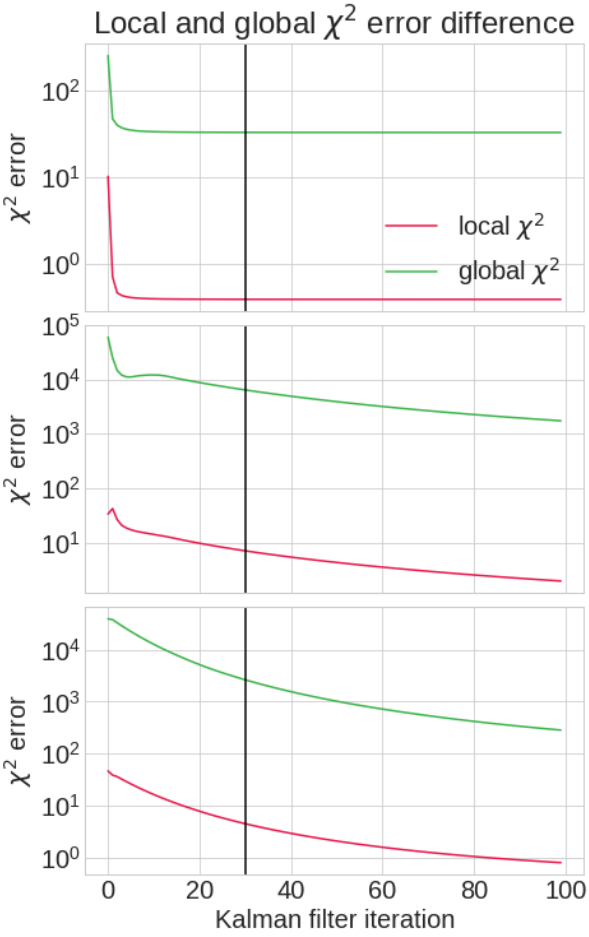

Figure 20: Local and global χ^2 errors for a track in events #5, #67 and #103 for 100 KF iterations. The 30 KF iterations commonly computed are marked by a black vertical line.

The local χ^2 error is computed while the EKF is running by projecting the state vector at a measurement site into the measurement itself and computing the squared difference divided by the uncertainty of the measurement. The problem with this version of the error is that as the EKF runs, each state vector changes, and thus the error computed is different from the real one.

The global χ^2 error is this "real" one, where each state vector near a measurement site is projected into it and the error is calculated. The reason that the global error is only computed after the EKF has ran is that the process of updating the state vector across

the particle trajectory is computationally-intensive, and thus doing it for each iteration of the EKF would be too expensive for the total computing time.

The problem that arises from this concept is that there are tracks where the convergence of the local χ^2 error is not representative of the converge of the global one. As can be seen in the first EKF iterations in the track from event #67 in figure 20, the local and the global χ^2 errors are not always correlated.

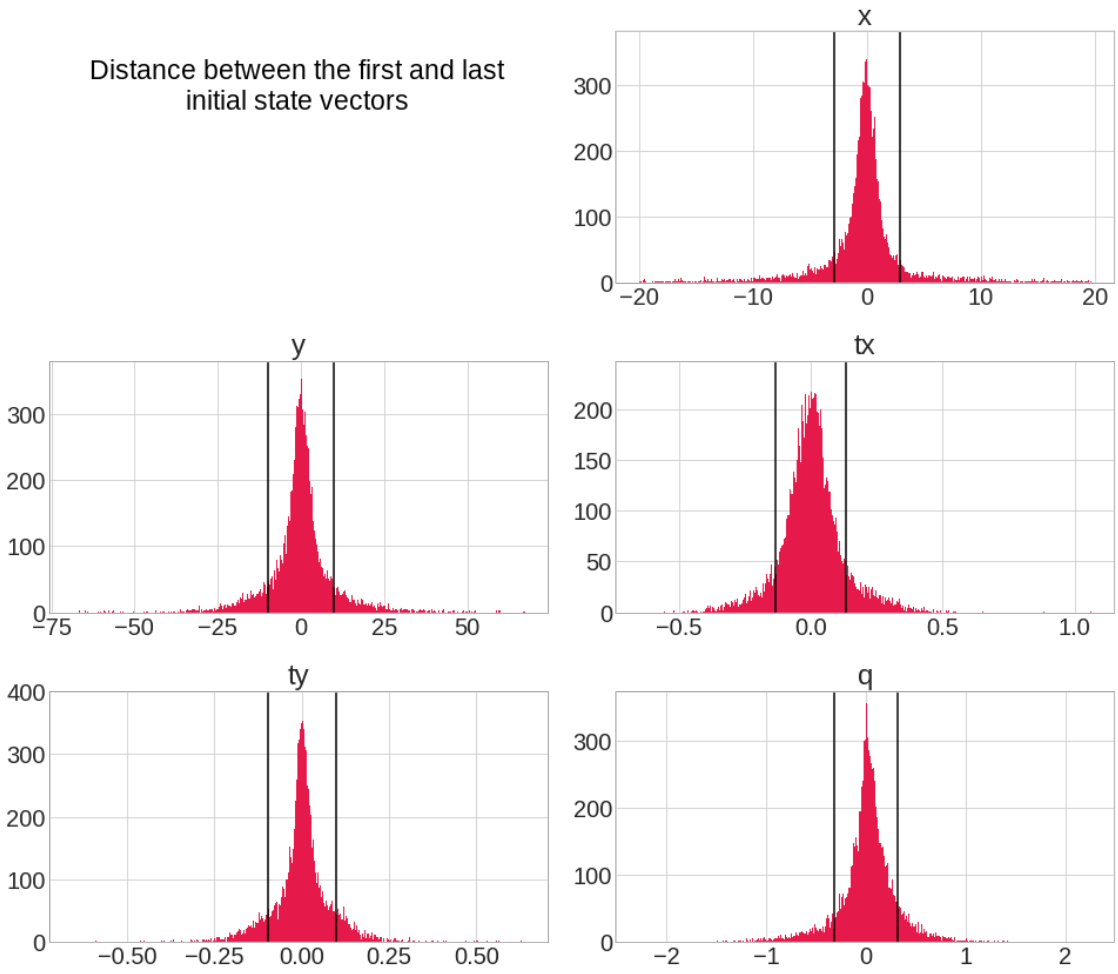

Figure 21: Distance between the first and the last initial vectors for the tracks found in 10.000 events. The black lines mark a region where 80% of the distances are contained.

Then, the idea of computing different initial state vectors $\mathbf{x}(z) = \langle x, y, \theta_x, \theta_y, q \rangle$ in parallel for each track was considered. These different initial state vectors should be consistent, and this is pursued by comparing the different variables from the first and the last state vectors used by the Kalman filter. This difference is recorded for all the tracks found in 10.000 events and its distribution is evaluated. A plot of this distribution for each variable in the state vector can be seen in Figure 21.

To generate the initial state vector for the KF to run, a random perturbation inside a range that captures 80% of the distance values (denoted as black vertical lines in Figure 21) is added to the initial state vector and multiple instances of the KF are ran with these different initial state vectors. The resulting χ^2 error obtained by running the KF in this manner for 6 random initial state vectors can be seen in Figure 22.

As can be seen in the figure, for the tracks in events 67 and 103 the final χ^2 obtained from a random initial state vector was better than the one obtained for the original. Special attention should be given to the ones obtained by the random state vectors 1 and 4 from event 67 which are close to a fifth of the original one found.

It's worth noting that if a dynamic stopping point is implemented, that is to say, a point in which the KF is stopped before reaching the 30 iterations, the multithreaded algorithm may provide better results in the same amount of time as the original one.

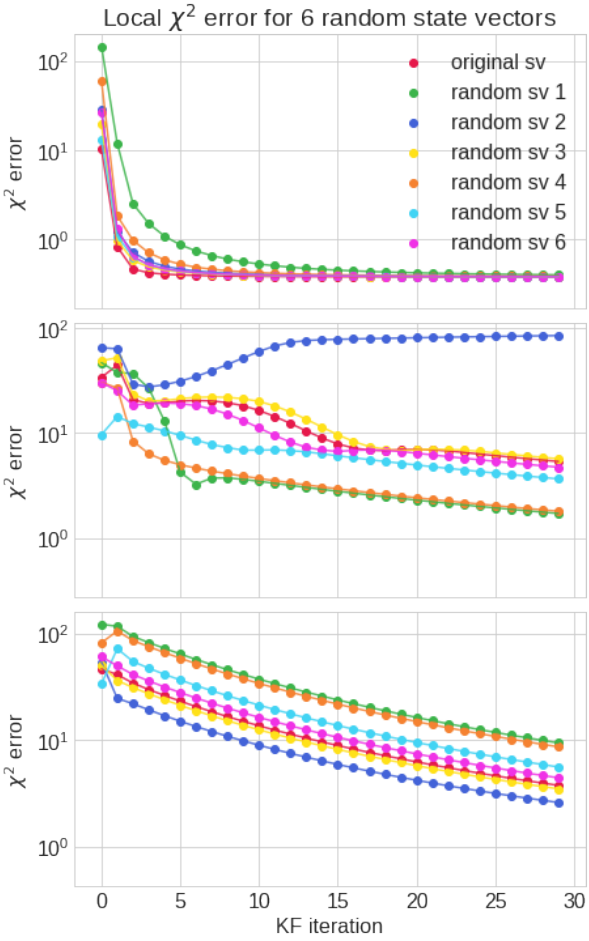

Figure 22: Local χ^2 errors starting from 6 random initial state vectors compared with the original in events #5, #67 and #103 for 30 KF iterations.

It's worth noting that while this algorithm may be able to provide better results in the same amount of time, the approach actually increases the total CPU-time and power consumption required to run the software. This is due to the fact that multiple instances of the KF run in parallel. In the case of the CLAS12 software as is ran in JLab, this additional power consumption meant that the algorithm was programmed but not implemented in the code due to power constraints.

CHAPTER 6
SOLUTION VALIDATION

6.1 Refactoring and Optimizations

The changes made described in Section 5.1 brought a CPU time reduction much larger than anticipated, especially for the DCHB engine. Contrasted to Figure 5, where the time DCHB and DCTB took per event was 169.68 [ms] and 47.76 [ms] on average, the new times for each is 118.14 [ms] and 44.34 [ms] respectively, meaning an improvement of 30.1% for DCHB and, less impressively, of 7.2% for DCTB.

As shown in Figure 23, there are some differences in the percent of the total CPU time distributed on each engine, with a heavy reduction apparent in the `clusterSplitter` and `setFitArray` methods. Using the versioning method mentioned in Section 4.2, this version is named 1.1.

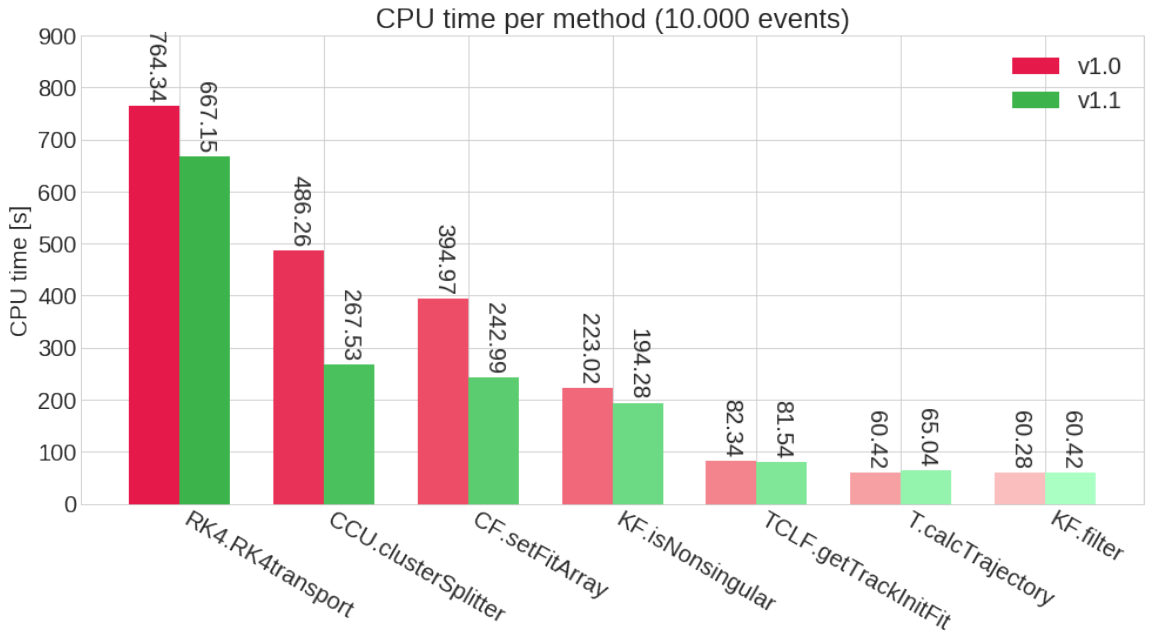

Figure 23: CPU time per method, version 1.1 contrasted with 1.0.

A comprehensive list of the improvements is presented:

- **Runge Kutta 4**: The total reduction in the time spent in the `RK4transport` method was of a 12.7% (97.19 seconds) for the 10.000 events evaluated. This is attributed to the precomputation of values achieved via the refactoring of the code that was done after comparing the original code with the real Runge Kutta 4 algorithm. Since the original implementation makes very little use of the modularity

inherent to the algorithm, the refactoring highlighted that the computations were done repeatedly since the new code is much shorter and easier to comprehend.

- **Cluster Splitter** and **Set Fit Array**: Both these methods were accelerated in a very similar manner to the one last described, reducing the total CPU time by 45.0% (from 486.26 to 267.53 [s]) and by 38.5% (from 394.97 to 242.99 [s]) respectively, thus making these two methods less of a priority when compared to RK4.
- **Is Nonsingular**: For this method a very small reduction in CPU time is perceived, just of 12.9% (from 223.02 to 194.28 [s]) in total. No change was made to this method, so the reduction is associated to the addition of better exception handling in previous steps so that iterations with "useless" data are stopped before reaching this point in the computation.
- **Get Track's Initial Fit**, **Calculate Trajectory** and **Filter**: These methods' CPU time were preserved between versions, seeing only very minor change. This is attributed to the profiler's measurement error.

6.2 Matrices Handling Changes

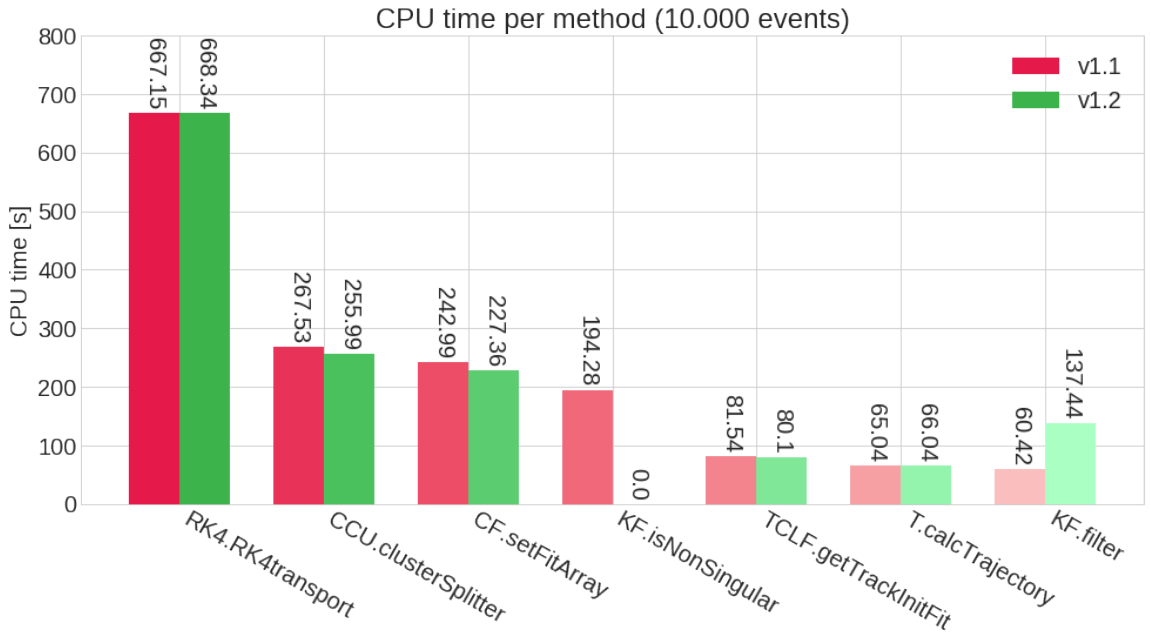

Figure 24: Method CPU times, version 1.2 compared with 1.1.

The changes described in Section 5.2 improve the total CPU time per event of DCHB and DCTB by 4.2% (from 118.14 to 113.19 [ms]) and by 3.2% (from 44.34 to 42.93 [ms]) respectively. As shown in Figure 24, the `isNonSingular` method sees a reduction

down to 0 [s] while the `filter` method sees a significant increase of 127.5% (77.04 [s]) and the rest of the methods see very little change, which is only attributed to error between measurements.

The increased time in the `filter` method is attributed to the fact that the time that was taken by the `isNonSingular` method is transferred to it due to the changes made to the code. The net gain in time is of 46.0% (from 254.70 to 137.44 [s] in total), which isn't great when considering that the total reduction seen in DCHB and DCTB's times is only of 6.36 [ms] per event.

The contrast between all the work done and the unremarkable change is attributed to the fact that this work is focused on finely optimizing the code and finding the best methods available to solve the problems, which goes against the principles behind the Java HotSpot compiler as it favors simple-to-understand methods and optimizes them while the application is running, neglecting already optimized code [Meloan, 1999]. This is opposite to the normal behaviour of more standard compilers like gcc, which in turn favors well-optimized code.

6.3 Magnetic Field Interpolation

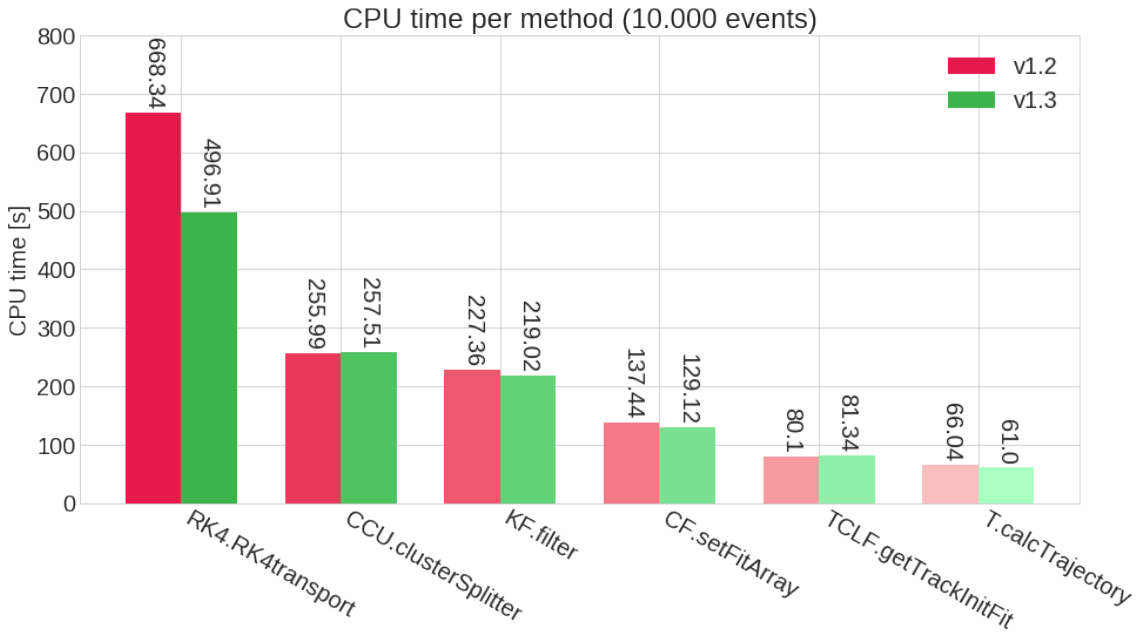

Figure 25: Method CPU times, version 1.3 compared with 1.2.

The time reduction brought by interpolating the magnetic field described in Section 5.3 was of 9.2% (from 113.19 to 102.79 [ms]) per event for the DCHB engine, while no improvement was seen for DCTB because, as described in the subsection, it isn't considered affordable to lose precision in Time-Based tracking.

Figure 25 clearly shows that the largest bulk of improvement was in the `RK4transport` method as should be expected, since this method calls the magnetic field methods that were slowing down the computation. All the other changes in the different methods are attributed to random differences between the profiling sessions, which are to be expected considering that the profiling was done via statistical sampling, as described in Section 3.7.

6.4 Multithreaded Cluster and Track Finding Algorithms

Considering that the nature of the changes done to the code in this version relate purely to using additional threads to compute results and do not reduce the CPU time, but only the throughput. It's worth noting that the style of figures that's been used up until now is no longer appropriate, so another is used.

The total time for the DCHB and DCTB engines are compared by running only the multithreaded cluster finding algorithm (version 1.4.a), only the multithreaded track finding (version 1.4.b), and using both algorithms together (version 1.4) to get the final total engine time for all the work in this document. These improvements can be seen in Figure 26.

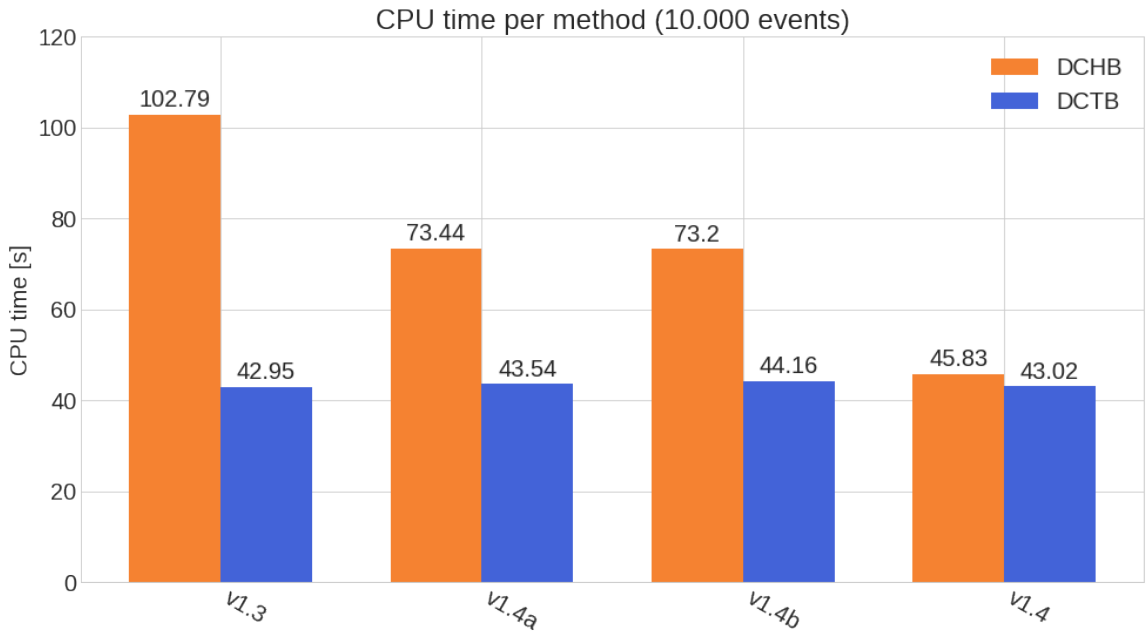

Figure 26: DCHB and DCTB final CPU times, version 1.4 compared with 1.3. Versions 1.4.a and 1.4.b represent only the multithreaded cluster finding and track finding algorithms respectively, while version 1.4 denotes both together.

It's very important to keep in mind that, unlike the other figures, the engine times improvement seen in this figure are not necessarily representative of the actual improvement that will be seen at JLab if and when the changes are implemented. This is because, as mentioned previously, the hardware used to run the performance tests was a personal computer, and not the cluster where the final implementation will be used. In its real context, the effectiveness of this version will depend in part on the number of free cores available while running the program, and thus the results obtained will fluctuate depending on this variable.

CHAPTER 7
CONCLUDING REMARKS

The problem faced in this work is multifaceted: it requires to understand the physics behind how particle accelerators and detectors work, then analyze the algorithms to implement these physics in a computer, and finally to code these algorithms in the implementation's environment. These different facets can also be seen in the contributions of this work. The team of physicists at JLab see the benefit of a quicker software which allows to analyze the data produced by the detector faster, while the general community benefits from ideas related to particle accelerators, Kalman filters and state estimators in general.

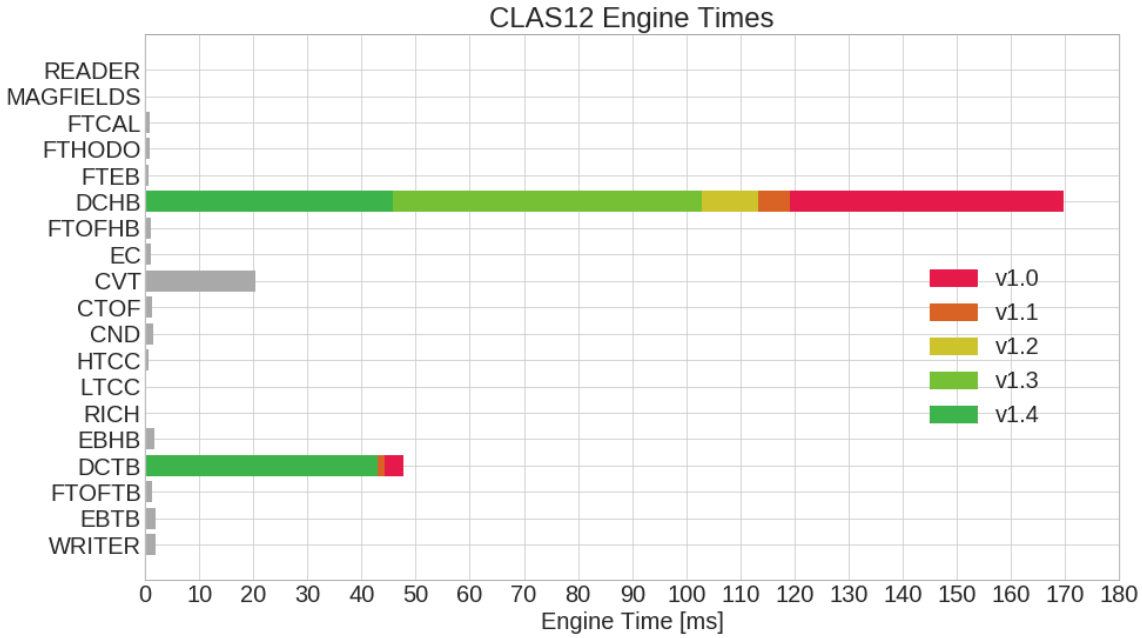

Figure 27: Engine CPU times.

Regarding the work done for JLab, the improvements on the DCHB and DCTB engines are considerable, as can be seen in Figure 27. A comprehensive list of each version, it's associated changes to the code, its effect in the total computation time and how useful it is for the community in general follows:

- **v1.1**: This version consists of the refactoring and optimizations explained in Section 5.1. It's surprising to see that this version brings the largest improvement in computing time, considering that the changes done only apply to the quality of the coding itself and were not changes to the algorithms themselves.

  This odd behaviour is explained by two main reasons: First, the changes reduce a large amount of unnecessary computations. Second and more impor-

tantly, the fact that the code was simplified allowed the Java HotSpot Performance Engine to apply heavier dynamic optimizations in runtime to the program [Hunt and John, 2011].

While this version provides a large improvement for the CLAS12 software, its implementation is mostly only useful for JLab since it is a direct quality improvement of their code and doesn't mean any change for the community in general.

- **v1.2**: This version consists on the implementation of different algorithms to improve the way matrices are handled by the Kalman Filter. This change doesn't provide a very significative improvement to the total computing time of CLAS12, which is attributed mostly to the matrix library used in the software, JAMA. While the changes brought heavily reduce the complexity of the operations completed in the software as is evidenced in Section 5.2, they also require a higher amount of matrices to be initialized, which isn't too efficient in the library [Hicklin et al., 2000].

  However, the changes applied could produce a bigger improvement for other particle detectors' software or for other instances of the EKF, considering that the need for matrix inversions and multiplications is completely eliminated in the Update phase of the filter. A potential future work that is proposed to reap the benefit of the work done in this version would be to switch to another matrix library or implement a new one that takes better advantage of these changes.

- **v1.3**: This version relates to the implementation of a magnetic field interpolation algorithm as is described in Section 5.3. Again, this change doesn't provide a very big difference on the final computing time of DCHB, but it has the advantage of producing such improvements with minimal effect on the precision of the results. As for the benefit that this work could provide for the community in general, its potential is limited by its use; while other publications provide interpolation methods with much higher accuracy [Mackay et al., 2006], this one provides very quickly computed results which can be accurate enough for certain applications, as can be seen in Figure 28.

- **v1.4**: This version simply shows an exploitable capability of particle detectors, which is that many hit clustering and particle tracking algorithms that are currently employed for particle detectors work with independent tracks, and thus are easily parallelizable [Blum et al., 2008].

- **Additional developments**: The work presented in subsection 5.5, while not implemented in CLAS12 due to specific constraints, provides useful insight into they improvement they could bring to other particle detectors.

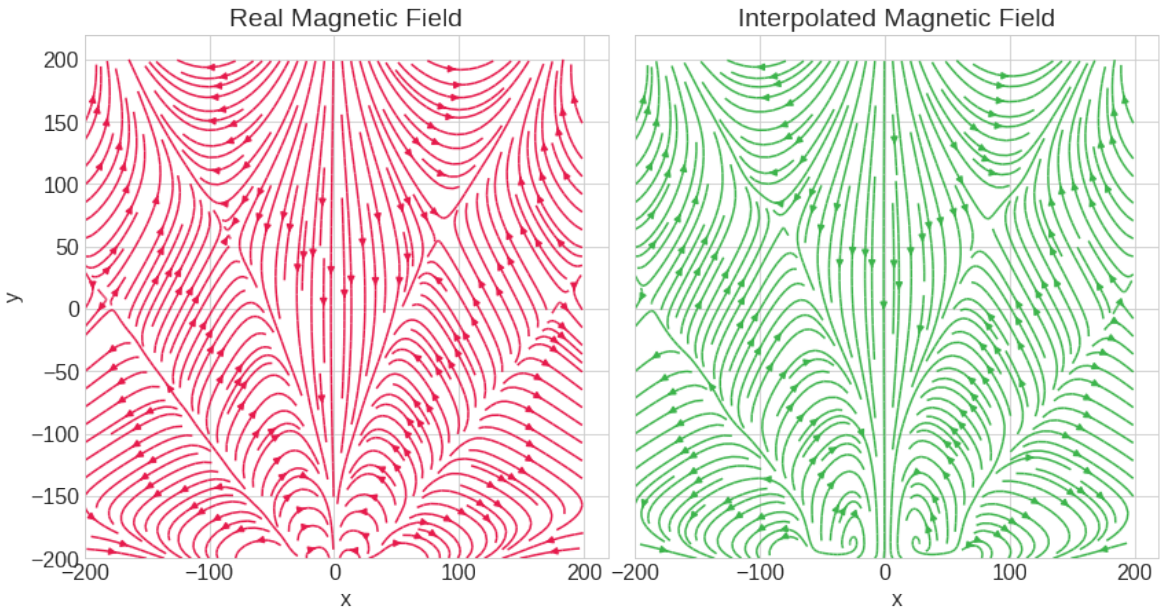

Figure 28: Vertical cut through the magnetic field transverse to the beam line at $z = 500$ [cm].

The alternative of utilizing the multithreaded KF algorithm proposed can potentially find better final tracks found if the additional computing time can be spared. Also, if more time is given to fine-tune this algorithm, it has the potential of improving computing time by analyzing different tracks with less KF iterations, thus diversifying the space used for initial state vectors.

During the development of this project, some ideas were not evaluated or implemented because they fell out of the scope of this work or because they were not useful in the specific case of CLAS12, but may be useful for other particle detectors that employ Drift Chambers. A list of these is offered with the hope that they will be investigated in future works:

- **Fourth Order Adams-Bashforth-Moulton method**: As can be seen in Figure 4 from Section 3.3, the measurements can be divided in three regions due to the geometry of the detector. While most of the distances between measurements are usually at most 4 or 5 [cm], the distance between regions can be more than a meter. Also, due to the way that the Kalman Filter works, after transporting the state vector from the first measurement site to the last, this vector needs to be transported back to run the KF again, which requires it to be moved for around 3.5 meters, or the entire length from sector 3 to sector 1.

Currently, this movement of the state vector is done through Runge Kutta 4, as is described in Section 3.3 with a step size $h = 1$ so that the z variable is updated by one centimeter in each RK4's iteration, occasionally updating it by 2 [cm] when the magnetic field at that point is small enough. Considering that for most particles the majority of the movement from regions 1 to 2, 2 to 3 and from 3 back to 1 the step size is kept at 1 [cm], the option of switching RK4 for the Fourth order Adams-Bashforth-Moulton (ABM4) method for these intervals is evaluated.

ABM4 is a multi-step method used to solve initial value problems described in detail in Appendix 8.13. The method can solve the same kind of problems as RK4, but it has the advantage that it only requires to evaluate the $\mathbf{f}(z, \mathbf{x})$ function twice per iteration, whereas RK4 does this four times.

The change does not come without disadvantage however, since it requires the use of four previous steps to compute the next one as can be seen in equations (20) and (21) from the appendix. Due to this the step size must be kept constant through each step, and RK4 needs to be ran for three steps after the initial state vector. It's worth noting that due to the particular conditions of the problem, z_{k+1} does not necessarily fulfill the condition that $z_{k+1} = z_k + nh$ where n is an arbitrary natural number, so it's usually impossible to reach the next measurement site from the last one using ABM4, but this can be fixed easily by using RK4 between the last step reached by ABM4 and the measurement plane z_{k+1}.

Due to specific difficulties and time constraints this option was programmed but not implemented, but it's considered a valuable option for future work.

- **Split cluster filtering**: While working with the software, it was observed that some split clusters separated using the cluster splitting algorithm described in Section 3.4 are actually produced by what looks like one or a few particles and a large amount of noise. The split clusters then go through the track finding algorithm described in Section 3.6 normally and a large portion of time is spent fitting each separately to then be scrapped as noise.

  This time could potentially be reduced by running a special track finding algorithm that analyzes them with more scrutiny before running the rest of the algorithm. Another option would be trying to fine tune the parameters in the cluster splitting algorithm so that, ideally, these noise clusters would be removed before it is attempted to find tracks from them.

- **Magnetic field interpolation parameter tuning**: Regarding the magnetic field interpolation proposed in Section 5.3, the parameter tuning could use much more work which was not considered due to the large amount of time the tests would take. The option of fine tuning each parameter separately opens the possibility of a more precise or faster implementation by offering a smaller step size in some dimensions or different ranges for the x and y variables.

- **Alternate magnetic field interpolation methods**: Attention is also brought to the fact that only trilinear interpolation is used. Triquadratic or tricubic interpolation might produce more accurate results, but they were not attempted due to the fact that they could potentially be far slower than the original implementation, which would work against the original objective of interpolating the field in the first place.

- **Kalman Filter Fine Tuning**: The EKF implemented in CLAS12 works with various parameters, like the number of iterations, the Runge-Kutta 4 original step size and its criteria for increasing this step size, the convergence criteria for stopping the filter before reaching the maximum number of iterations, and the newly introduced number of iterations using the interpolated magnetic field.

  All of these parameters were set somewhat haphazardly, by trial and error. The option of running another Kalman Filter that trains the original one by fine-tuning these parameters is proposed, with the alternative of using methods from machine learning, which has already been done in the academy [Abbeel et al., 2005].

- **Hough-Transform-based cluster finding**: An option that was considered but quickly discarded due to time constraints is that of using the Hough Transform (described in Appendix 8.5) over the set of all the hits in one event instead of only to split clusters.

  While this option would probably be slower in CPU, it offers the possibility of leveraging this computation to a GPU, which has already been proved possible [Van den Braak et al., 2011], even for the specific case of particle accelerators, such as the LHC [Halyo et al., 2013].

- **FPGA-or-ASIC-based cluster finding**: Another option that was considered but not tried is that of migrating the cluster finding described in Section 3.4 to hardware via Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). The proposal comes from the fact that the Clump Finding and Hit Pruning parts of the algorithm are very basic and can be done without any floating-point computation, task that famously can be hard for FPGAs [Fagin and Renard, 1994].

  It might even be possible to run the whole cluster finding algorithm in FPGAs by discovering a good implementation of the **Cluster Fitting** and **Cluster Splitting** parts of the algorithm that don't heavily rely on floating-point operations.

Special attention is brought again to the fact that the biggest improvement in the code is seen in version 1.1, the "code clean-up". The refactoring done was not completely thorough, and perhaps even larger changes can be obtained by reworking the DC software by rewriting most classes from the algorithm up.

A final reflection on this same subject relates to the language choice for the CLAS12 software, Java. In the current programming environment, where processing units' focus is shifting from complex and fast general purpose CPUs to lower-power, multi-core processor units, the software should be adapting alongside the hardware and a shift to languages with small memory footprint and well-defined states should be preferred. This change of focus can be seen, for example, in the software development for the LHC, where clear guidelines have been stated to switch software to C++ [Foundation et al., 2017] and to use simpler date structures, drifting away from the objects common to object-oriented programming [Cerati et al., 2016].

While originally the choice of using Java might have made sense, in the current context of high performance computing this option is disputed. It has been repeatedly proven that C/C++ provides at least double the computing speed of Java and at most a quarter of the memory usage [Prechelt, 2000], with even larger differences found in other studies [Prechelt et al., 1999]. Changing the codebase to C/C++ can even be aided by the computing framework, CLARA, due to the fact that it supports services in both Java and C++ [Gyurjyan et al., 2013], which means that the change could be applied gradually; thus allowing for incremental tests to be applied while this change is taking place.

APPENDICES

8.1 On the Flops of Different Operations

Floating point operations per second or flops is a measure for computer performance usually considered more accurate than the instructions per second. It is commonly stated that some operations require more flops than others, so different operations are considered separately when analyzing the computational complexity. For the study done in this document, different operations are counted together for simplicity and due to the fact that, as can be seen in Table 3, their computation time is not too different in modern architectures, or at least in a personal computer's CPU.

operation	time [ns]
sum	1.7005
mul	1.4478
div	1.8587
sqrt	1.7818

Table 3: Operation times in [ns].

Due to a very high variance in the operation time during different iterations of the same code in Java and the language's lack of capacity to do high-precision tests, the tests to obtain these values were performed in C.

The tests were ran by measuring the CPU time in the machine before and after each operation was done on pseudo-random double-precision floating point values using the `clock()` function from the `time.h` library. Each operation was performed 10.000.000 times and the average computation time was used to minimize statistical error on an Intel(R) Core(TM) i5-6600K CPU with 4 cores running at 3.50GHz with 8GB of DDR4 2133MHz RAM. The source code for this test can be seen in `https://gist.github.com/bleaktwig/211dab3a5f95b4e857f1f001d7b13654`.

8.2 Error Measurements

The quadratic error, mean squared error (MSE) or χ^2 error of a procedure estimating a random variable measures the average of the squares of the errors between each measured variable and their estimated value. With $\hat{y} = f(x)$ being a function of measured values y and n being the number of samples taken such that $y_1, y_2, ..., y_n$ are known and $\hat{y}_1, \hat{y}_2, ..., \hat{y}_n$ can be obtained from $\hat{y}_i = f(x_i)$, the quadratic error of the

function can be defined as [Taylor, 1997]:

$$\chi^2 = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \hat{y}_i)^2}{\hat{y}_i}\,.$$

In the case of the DC software, the quadratic error is used to describe how good a track fit is when compared to the measured data.

8.3 Covariance Matrix

Given a random vector $\mathbf{x}$ of size n defined as $\mathbf{x} = \{x_1, x_2, ..., x_n\}$, the covariance matrix Σ of such vector is usually defined as:

$$\Sigma = \begin{bmatrix} \mathrm{var}(x_1) & \mathrm{cov}(x_1, x_2) & \dots & \mathrm{cov}(x_1, x_n) \\ \mathrm{cov}(x_2, x_1) & \mathrm{var}(x_2) & \dots & \mathrm{cov}(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{cov}(x_n, x_1) & \mathrm{cov}(x_n, x_2) & \dots & \mathrm{var}(x_n) \end{bmatrix},$$

with $\mathrm{var}(x_i)$ signifying the variance of the random variable x_i and $\mathrm{cov}(x_i, x_j)$ the covariance of variables x_i and x_j. Variances and covariances are defined as:

$$\mathrm{var}(x) = \frac{1}{m}\sum_{i=1}^{m}(x_i - \bar{x})^2$$

$$\mathrm{cov}(x, y) = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})\,,$$

where $\bar{x}$ denotes the mean or average of x, and is defined as:

$$\bar{x} = \frac{1}{m}\sum_{i=1}^{m} x_i\,,$$

where m denotes the number of measurements.

Due to the fact that by definition neither variances or covariances can be negative and that $\mathrm{cov}(x_i, x_j) = \mathrm{cov}(x_j, x_i)$, the covariance matrix Σ of a vector is always symmetric and positive-definite.

8.4 Jacobian Matrix

For an arbitrary vectorial function $\mathbf{f}(\mathbf{x})$, the jacobian matrix is a matrix that lists all its first-order partial derivatives. Assuming that the function $\mathbf{f}(\mathbf{x})$ can receive a generic input vector $\mathbf{x} = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^n$ and produces an output $\mathbf{f}(\mathbf{x}) = \{f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_m(\mathbf{x})\} \in \mathbb{R}^m$, then the Jacobian matrix J_f of $\mathbf{f}(\mathbf{x})$ is the $m \times n$ matrix defined in the following manner:

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

8.5 Hough Transform

The Hough Transform is a method usually utilized to extract features in image analysis, originally proposed by Paul Hough [Hough, 1962]. The method is used to detect simple shapes such as straight lines or circles in any set of points with given coordinates (usually from images) by finding groupings in these sets via a voting procedure over a set of parameterized properties. An example of the detection of straight lines can be seen in Figure 29. In the left image the points are marked in white, while in the right image the lines found are marked in red.

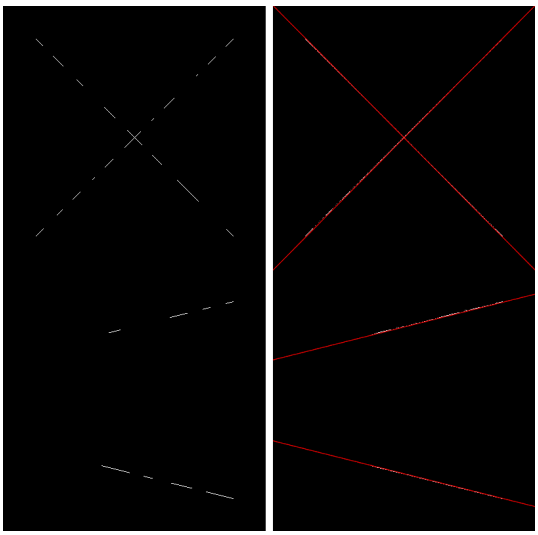

Figure 29: Hough transform example. The red lines on the right image denote the lines found by the method.

For the case of detecting straight lines in two dimensions, the line is defined as $y = mx + b$ with its representation in its **Hessian normal form** [Gellert et al., 2012] is:

$$r = x\cos\theta + y\sin\theta\,,$$

where r is the distance from the origin to the closest point in the line and θ is the angle between the x axis and the line. Given a point in the plane, the set of all lines going through that point corresponds to a sinusoidal curve in the (r, θ) plane. A set of more than one point that form a straight line will produce multiple sinusoids crossing at the (r, θ) plane for that line, and thus, the original problem of detecting collinear points can be expressed as the problem of finding concurrent curves.

8.6 Architectural Technical Debt

The term "Technical Debt" is attributed to a concept described as a debt that arises when "shipping first time code" [Cunningham, 1992]. This is a debt in the sense that the code written works correctly but will eventually require the programmer to use time to re-write it with the added knowledge of how it will interact with the rest of the code. The term becomes significant when the programmer doesn't spend the time required to fix the code and continues working on the code base, causing the debt to accumulate "interest", so that when the issue is finally addressed it is much harder to do so due to all the code that works interconnected to this "dirty" code.

Technical debt can be intentional or unintentional, and managed or unmanaged. An intentional debt refers to one incurred by a programmer who is aware of the future time it will take to fix the code, while an unintentional one to one incurred by accident due to either lack of knowledge about the "engineering best practices", not applying the correct abstractions, writing a simple but slow algorithm, among others. A managed debt refers to one where the programmer is aware of the debt and plans to fix it before creating new components dependant on the code where the debt is owed, while an unmanaged debt refers to the contrary [Allman, 2012].

It is evident that an unintentional, unmanaged debt causes long term instability issues with a program, while an intentional and managed debt can be used as a tool by the programmer to provide quick fixes or releases, but taking the time later to repay the debt. The two other possible cases are more rare; an intentional, unmanaged debt either means that the programmer is in a hurry or he/she simply doesn't care. An unintentional managed debt is even more unlikely [Allman, 2012] and may be simply described as a happy accident.

8.7 Sherman-Morrison Formula

Originally proposed by Jack Sherman and Winifred J. Morrison [Sherman and Morrison, 1950], the Sherman-Morrison formula says that if $A \in \mathbb{R}^{n\times n}$ is a nonsingular matrix and $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are vectors:

$$(A + \mathbf{u}\mathbf{v}^T) = A^{-1} - \frac{A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}}{1 + \mathbf{v}^T A^{-1}\mathbf{u}}\,,$$

given that $1 + \mathbf{v}^T A^{-1}\mathbf{u} \neq 0$.

This formula is especially useful when A^{-1} is unknown but A is known, or in special cases such as the one seen in Equation (13), where:

$$\begin{aligned} P_{k|k} &= (P_{k|k-1}^{-1} + \mathbf{h}\mathbf{h}^T)^{-1} \\ &= P_{k|k-1} - \frac{P_{k|k-1}\mathbf{h}\mathbf{h}^T P_{k|k-1}}{1 + \mathbf{h}^T P_{k|k-1}\mathbf{h}}, \end{aligned}$$

since it eliminates the need to compute the inverse of $P_{k|k-1}$ and, if the operations are resolved in a particular order, it even eliminates the need for multiplying matrices, which is an expensive operation by itself.

8.8 Su-Chang Formula

Originally proposed by Ching Tzong-Su and Feng Cheng Chang [Su and Chang, 1996], the Su-Chang formula proposed a method to compute the determinant of a square matrix A by splitting it into a matrix $W \in \mathbb{R}^{(n-1)\times(n-1)}$, two column vectors $\mathbf{u} \in \mathbb{R}^{n-1}$ and $\mathbf{v} \in \mathbb{R}^{n-1}$ and a scalar $r \in \mathbb{R}$:

$$\begin{aligned} |A| &= \begin{vmatrix} W & \mathbf{v} \\ \mathbf{u}^T & r \end{vmatrix} \\ &= r \cdot \left| W - \frac{\mathbf{v}\mathbf{u}^T}{r} \right|, \end{aligned}$$

given that $r \neq 0$.

A recursive algorithm can then be defined as:

Algorithm 5: Su-Chang algorithm

def suchang_det(A: matrix, n: int)**:**
  if $n == 1$ **then**
    return a_{00}
  $W = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,(n-1)} \\ a_{1,0} & a_{1,1} & \dots & a_{1,(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1),0} & a_{(n-1),1} & \dots & a_{(n-1),(n-1)} \end{bmatrix}$
  $\mathbf{v} = \langle a_{0,n}, a_{1,n}, \dots, a_{(n-1),n} \rangle$
  $\mathbf{u} = \langle a_{n,0}, a_{n,1}, \dots, a_{n,(n-1)} \rangle$
  $r = a_{n,n}$
  return $r \cdot$ suchang_det$\left(W - \frac{\mathbf{v}\mathbf{u}^T}{r}\right)$

where n is the square matrix A's row and column dimension.

If each step of the algorithm is denoted by i, it's easy to see that each step requires $(n-i)^2+1$ multiplications, $(n-i)^2$ subtractions and 1 division. From this, the mathematical complexity can be calculated:

$$\begin{aligned}
\mathrm{C}(\text{suchang_det}(A)) &= \sum_{i=1}^{n-2}((n-i)^2+1+(n-i)^2+1) \\
&= 2\sum_{i=1}^{n-2}((n-i)^2+1) \\
&= 2\sum_{i=2}^{n-1}(i^2+1) \\
&= \frac{1}{3}(2n^3-3n^2+7n-18)\,, \qquad (19)
\end{aligned}$$

which is much less than the $n^3+\mathcal{O}(n^2)$ usually required to compute a matrix's determinant through LU decomposition [Banachiewicz, 1937].

It's worth noting that other publications extend this algorithm, but only address the problem that arises when a component of the A matrix's diagonal is 0, and don't improve further its efficiency [Chang and Su, 1998, Chang, 2014].

8.9 Cholesky Matrix Decomposition

If $A \in \mathbb{R}^{n\times n}$ is a symmetric positive-definite matrix, there exists a lower diagonal matrix $L \in \mathbb{R}^{n\times n}$ such that:

$$A = L\,L^T\,.$$

Algorithmically, a naïve implementation for each component of L is shown in algorithm 6.

From the algorithm, it can be seen that the decomposition requires the calculation of n square roots, $\frac{1}{2}n(n-1)$ divisions, $\frac{1}{6}n(n^2-1)$ multiplications, $\frac{1}{6}n(n^2-3n-4)$ additions and $\frac{1}{2}n(n+1)$ subtractions. Summed together, these add up to a total of $\frac{1}{6}n(2n^2+3n+1)$ operations.

Algorithm 6: Cholesky Decomposition

def cholesky_decomposition(A: matrix, n: int)**:**

$l_{1,1} = \sqrt{a_{1,1}}$

for $j \leftarrow 2$ **to** n **do**

$l_{j,1} = \frac{a_{j,1}}{l_{1,1}}$

end

for $i \leftarrow 2$ **to** n **do**

$l_{i,i} = \sqrt{a_{i,i} - \sum_{p=1}^{i-1} l_{i,p}^2}$

if $i \neq n$ **then**

for $j \leftarrow i+1$ **to** n **do**

$l_{j,i} = \frac{1}{l_{i,i}} \left(a_{j,i} - \sum_{p=1}^{i-1} l_{i,p} \cdot l_{j,p} \right)$

end

end

8.10 Matrix Determinant Lemma

The matrix determinant lemma can be defined as follows [Brookes, 2019]:

Lemma 1 *If $A \in \mathbb{R}^{n \times n}$ is an invertible square matrix and $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are column vectors, it can be proven that:*

$$|A + \mathbf{u}\mathbf{v}^T| = (1 + \mathbf{v}^T A^{-1} \mathbf{u}) \cdot |A| \,.$$

The lemma is especially useful in cases where $|A|$ and A^{-1} are already known, since it allows for the calculation of the determinant of A updated by adding the matrix $\mathbf{u}\mathbf{v}^T$ to it without computing the new determinant from scratch.

8.11 Cauchy-Schwarz Inequality

The Cauchy-Schwarz inequality or Cauchy-Bunyakovsky-Schwarz inequality is a very useful inequality which states that, if $\mathbf{u}$ and $\mathbf{v}$ are any vectors inside an inner product space, it can be proven that:

$$|\langle \mathbf{u}, \mathbf{v} \rangle| \leq \|\mathbf{u}\| \|\mathbf{v}\| \,,$$

where $\langle \cdot, \cdot \rangle$ is the inner product [Strang, 2005].

Considering that the expectation of a product of two random variables is an inner product, this leads to the covariance inequality:

$$|\mathrm{cov}(x, y)|^2 \leq \mathrm{var}(x)\mathrm{var}(y) \,,$$

where var$(\cdot)$ denotes the variance of a variable and cov$(\cdot,\cdot)$ the covariance between two, and are defined in Appendix 8.3.

8.12 Trilinear Interpolation

Trilinear Interpolation [Bourke, 1999] is a method that can be used to compute the value in a tridimensional space given that a regular grid of (x, y, z) points and the value of the function to be interpolated f is known for this set of points.

Given a point (x, y, z) from which the value $f(x, y, z)$ is to be estimated, it is easy to see that the value lies inside a cube in the given regular grid with borders (x_0, x_1), (y_0, y_1), (z_0, z_1), so that the values c_{000}, c_{001}, c_{010}, c_{011}, c_{100}, c_{101}, c_{110}, c_{111}, defined as $c_{000} = f(x_0, y_0, z_0)$, $c_{001} = f(x_0, y_0, z_1)$ and so on are already known, as seen in Figure 30.

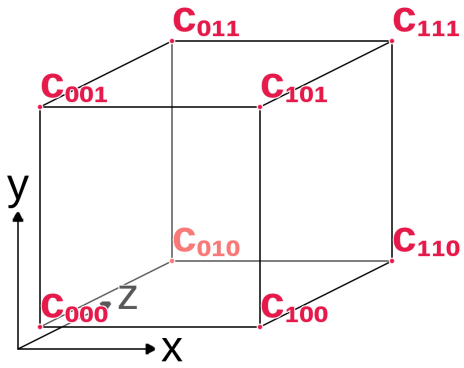

Figure 30: Trilinear interpolation visual interpretation.

From this data, the values x_d, y_d and z_d can be obtained as:

$$x_d = \frac{x - x_0}{x_1 - x_0},\ y_d = \frac{y - y_0}{y_1 - y_0},\ z_d = \frac{z - z_0}{z_1 - z_0},$$

from which linear interpolation can be done "in steps", first for x:

$$\begin{aligned} c_{00} &= c_{000}(1 - x_d) + c_{100}x_d \\ c_{01} &= c_{001}(1 - x_d) + c_{101}x_d \\ c_{00} &= c_{010}(1 - x_d) + c_{110}x_d \\ c_{00} &= c_{011}(1 - x_d) + c_{111}x_d\,, \end{aligned}$$

then for y:

$$\begin{aligned} c_0 &= c_{00}(1 - y_d) + c_{10}y_d \\ c_1 &= c_{01}(1 - y_d) + c_{11}y_d \end{aligned}$$

,

and finally for z:

$$c = c_0(1 - z_d) + c_1 z_d\,,$$

with c denoting the interpolated $\tilde{f}(x, y, z)$ value.

8.13 Fourth Order Adams-Bashforth-Moulton Method

The Fourth Order Adams-Bashforth-Moulton method or simply Adams-Bashforth-Moulton 4 (ABM4) is a multi-step method used to solve initial value problems similar to RK4 in use. Solving the same equation as RK4, let an initial value problem for estimating $\mathbf{x}(z+h)$ be:

$$\dot{\mathbf{x}} = \mathbf{f}(z, \mathbf{x}), \mathbf{x}(z_0) = \mathbf{x}_0\,,$$

where the definitions are equivalent to the ones given in Section 3.7. To solve the problem using a step size h, first the fourth order Adams-Bashforth (AB4) method is proposed:

$$\mathbf{x}^*_{n+1} = \mathbf{x}_n + \frac{h}{24}(55\cdot\mathbf{f}(z,\mathbf{x}_n) - 59\cdot\mathbf{f}(z-h,\mathbf{x}_{n-1}) + 37\cdot\mathbf{f}(z-2h,\mathbf{x}_{n-2}) - 9\cdot\mathbf{f}(z-3h,\mathbf{x}_{n-3}))\,, \tag{20}$$

which is commonly denoted as the "explicit" AB4 method. Then, from this result it's possible to compute a more accurate result for $\mathbf{x}_{n+1}$ using the fourth order Adams-Moulton (AM4) method, or the "implicit" AM4 method:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{24}(9\cdot\mathbf{f}(z+h,\mathbf{x}^*_{n+1}) + 19\mathbf{f}(z,\mathbf{x}_n) - 5\mathbf{f}(z-h,\mathbf{x}_{n-1}) + \mathbf{f}(z-2h,\mathbf{x}_{n-2})\,. \tag{21}$$

While the results of this method are not necessarily more accurate than the ones obtained by RK4 since both methods have the same convergence order, the method has the advantage that it only requires to compute $\mathbf{f}(\cdot,\cdot)$ twice for every time-step (assuming that the previous results are stored), whereas RK4 requires this function to be computed four times per time-step. This can result in a serious improvement if the time it takes to compute $\mathbf{f}(\cdot,\cdot)$ is large.

One problem that arises from the implementation though is that at the beginning of the computation only $\mathbf{x}_0$ is known, and four steps are needed to start. To solve this, it's not uncommon to use RK4 to compute $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ and then start using ABM4.

8.14 Code and Reproducibility

The original CLAS12-offline-software repository can be found in:

`https://github.com/JeffersonLab/clas12-offline-software`

while the commit it was in as the project started is `fb6d5b2` (link last accessed as of September 27 2019).

The repository's fork where the author's work can be found in is:

`https://github.com/bleaktwig/clas12-offline-software`

and the commit by the end of this project's development is `12924ec` (link last accessed as of September 27 2019).

The contact information for the people behind the software, CLAS12 and Hall B is as follows:

- V. Ziegler, CLAS12 Software Project Coordinator (`ziegler@jlab.org`) 757-269-6003
- L. Elouadrhiri, Control Account Manager (`latifa@jlab.org`) 757-269-7303
- G. Young, Associate Project Manager for Physics (`young@jlab.org`) 757-269-6904
- V. D. Burkert, Hall B Group Leader (`burkert@jlab.org`) 757-269-7540

REFERENCES

[Aad et al., 2012] Aad, G., Abajyan, T., Abbott, B., Abdallah, J., Khalek, S. A., Abdelalim, A. A., Abdinov, O., Aben, R., Abi, B., Abolins, M., et al. (2012). Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc. *Physics Letters B*, 716(1):1–29.

[Abbeel et al., 2005] Abbeel, P., Coates, A., Montemerlo, M., Ng, A. Y., and Thrun, S. (2005). Discriminative training of kalman filters. In *Robotics: Science and systems*, volume 2, page 1.

[Aho et al., 1989] Aho, A. V., Ganapathi, M., and Tjiang, S. W. (1989). Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516.

[Allman, 2012] Allman, E. (2012). Managing technical debt. *Commun. ACM*, 55(5):50–55.

[Ball and Larus, 1997] Ball, T. and Larus, J. (1997). Efficient path profiling. In *MICRO*, pages 46–57.

[Ball and Larus, 1994] Ball, T. and Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360.

[Ballard et al., 2012] Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., and Schwartz, O. (2012). Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 193–204. ACM.

[Banachiewicz, 1937] Banachiewicz, T. (1937). Zur berechnung der determinanten, wie auch der inversen und zur darauf basierten auflosung der systeme linearer gleichungen. *Acta Astronom. Ser. C*, 3:41–67.

[Blattner and Yang, 2011] Blattner, T. and Yang, S. (2011). Performance study on cuda gpus for parallelizing the local ensemble transformed kalman filter algorithm. *Concurrency and Computation: Practice and Experience*, 24(2):167–177.

[Blum et al., 2008] Blum, W., Riegler, W., and Rolandi, L. (2008). *Particle detection with drift chambers*. Springer Science & Business Media.

[Bourke, 1999] Bourke, P. (1999). Interpolation methods. Online; accessed 23-September-2019.
`http://paulbourke.net/miscellaneous/interpolation/`.

[Brookes, 2019] Brookes, M. (2019). The matrix reference manual. Online; accessed 28-August-2019
`http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/identity.html`.

[Cerati et al., 2016] Cerati, G., Tadel, M., Würthwein, F., Yagil, A., Lantz, S., McDermott, K., Riley, D., Wittich, P., and Elmer, P. (2016). Kalman-filter-based particle tracking on parallel architectures at hadron colliders. In *2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, pages 1–4. IEEE.

[Chang, 2014] Chang, F. C. (2014). Determinant of matrix by order condensation. *British Journal of Mathematics & Computer Science*, 4:1843–1848.

[Chang and Su, 1998] Chang, F. C. and Su, C.-T. (1998). More on quick evaluation of determinants. *Applied Mathematics and Computation*, 93(1):97–99.

[Collaboration et al., 2008] Collaboration, C. et al. (2008). The cms experiment at the cern lhc.

[Coppersmith and Winograd, 1990] Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280.

[Cunningham, 1992] Cunningham, W. (1992). The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30.

[Demchenko et al., 2013] Demchenko, Y., Grosso, P., De Laat, C., and Membrey, P. (2013). Addressing big data issues in scientific data infrastructure. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 48–55. IEEE.

[Duda and Hart, 1971] Duda, R. O. and Hart, P. E. (1971). Use of the hough transformation to detect lines and curves in pictures. Technical report, Sri International Menlo Park Ca Artificial Intelligence Center.

[Einicke, 2006] Einicke, G. A. (2006). Optimal and robust noncausal filter formulations. *IEEE Transactions on Signal Processing*, 54(3):1069–1077.

[Fagin and Renard, 1994] Fagin, B. and Renard, C. (1994). Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(3):365–367.

[Fatahalian et al., 2004] Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH EUROGRAPHICS conference on Graphics hardware*, pages 133–137. ACM.

[Foundation et al., 2017] Foundation, H., Albrecht, J., Alves Jr, A. A., Amadio, G., Andronico, G., Anh-Ky, N., Aphecetche, L., Apostolakis, J., Asai, M., Atzori, L., et al. (2017). A roadmap for hep software and computing r&d for the 2020s. *arXiv preprint arXiv:1712.06982*.

[Fraser et al., 1991] Fraser, C. W., Henry, R. R., and Proebsting, T. A. (1991). Burg–fast optimal instruction selection and tree parsing. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

[Gavalian, 2019] Gavalian, G. (2019). Data processing for clas12. Online; accessed 12-June-2019.
`https://agenda.infn.it/event/18179/contributions/89830/attachments/63379/76297/EIC-Streaming-Camogli-2019.pdf`.

[Gellert et al., 2012] Gellert, W., Hellwich, M., Kästner, H., and Küstner, H. (2012). *The VNR concise encyclopedia of mathematics*. Springer Science & Business Media.

[Gyurjyan et al., 2011] Gyurjyan, V., Abbott, D., Carbonneau, J., Gilfoyle, G., Heddle, D., Heyes, G., Paul, S., Timmer, C., Weygand, D., and Wolin, E. (2011). Clara: A contemporary approach to physics data processing. *Journal of Physics: Conference Series*, 331:032013.

[Gyurjyan et al., 2015] Gyurjyan, V., Bartle, A., Lukashin, C., Mancilla, S., Oyarzún, R., and Vakhnin, A. (2015). Component based dataflow processing framework. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1938–1942. IEEE.

[Gyurjyan et al., 2013] Gyurjyan, V., Mancilla, S., and Oyarzún, R. (2013). Clara: The clas12 reconstruction and analysis framework. *Bulletin of the American Physical Society*, 58.

[Halyo et al., 2013] Halyo, V., Hunt, A., Jindal, P., LeGresley, P., and Lujan, P. (2013). Gpu enhancement of the trigger to extend physics reach at the lhc. *Journal of Instrumentation*, 8(10):P10005.

[Hicklin et al., 2000] Hicklin, J., Moler, C., Webb, P., Boisvert, R. F., Miller, B., Pozo, R., and Remington, K. (2000). Jama: A java matrix package. *URL: http://math.nist.gov/javanumerics/jama*.

[Higgs, 1964] Higgs, P. W. (1964). Broken symmetries and the masses of gauge bosons. *Physical Review Letters*, 13(16):508.

[Hintjens, 2013] Hintjens, P. (2013). *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc.

[Hough, 1962] Hough, P. V. (1962). Method and means for recognizing complex patterns. US Patent 3,069,654.

[Huang et al., 2011] Huang, M.-Y., Wei, S.-C., Huang, B., and Chang, Y.-L. (2011). Accelerating the kalman filter on a gpu. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 1016–1020. IEEE.

[Hunt and John, 2011] Hunt, C. and John, B. (2011). *Java performance*. Prentice Hall Press.

[Julier and Uhlmann, 1997] Julier, S. J. and Uhlmann, J. K. (1997). New extension of the kalman filter to nonlinear systems. In *Signal processing, sensor fusion, and target recognition VI*, volume 3068, pages 182–194. International Society for Optics and Photonics.

[Kalman, 1960] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45.

[Karimipour and Dinavahi, 2015] Karimipour, H. and Dinavahi, V. (2015). Extended kalman filter-based parallel dynamic state estimation. *IEEE transactions on smart grid*, 6(3):1539–1549.

[Le Duff, 2005] Le Duff, J. (2005). Longitudinal beam dynamics in circular accelerators.

[Leemann et al., 2001] Leemann, C. W., Douglas, D. R., and Krafft, G. A. (2001). The continuous electron beam accelerator facility: Cebaf at the jefferson laboratory. *Annual Review of Nuclear and Particle Science*, 51(1):413–450.

[Lukashin et al., 2015] Lukashin, C., Gyurjyan, V., Bartle, A., and Callaway, E. (2015). Earth science and data fusion: Naiads concept and development. Online; accessed 12-June-2019.
`https://esto.nasa.gov/forum/estf2015/presentations/Lukashin_S7P9_ESTF2015.pdf`.

[Mackay et al., 2006] Mackay, F., Marchand, R., and Kabin, K. (2006). Divergence-free magnetic field interpolation and charged particle trajectory integration. *Journal of Geophysical Research: Space Physics*, 111(A6).

[Maple, 2015a] Maple, S. (2015a). Developer productivity report 2015: Java performance survey results. Online; accessed 22-July-2019.
`https://jrebel.com/rebellabs/developer-productivity-report-2015-java-performance-survey-results/2/`.

[Maple, 2015b] Maple, S. (2015b). Top 5 java profilers revealed: Real world data with visualvm, jprofiler, java mission control, yourkit and custom tooling. Online; accessed 22-July-2019.
`https://jrebel.com/rebellabs/top-5-java-profilers-revealed-real-world-data-with-visualvm-jprofiler-java-mission-control-yourkit-and-custom-tooling/`.

[Mecking et al., 2003] Mecking, B. A., Adams, G., Ahmad, S., Anciant, E., Anghinolfi, M., Asavapibhop, B., Asryan, G., Audit, G., Auger, T., Avakian, H., et al. (2003). The cebaf large acceptance spectrometer (clas). *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 503(3):513–553.

[Meloan, 1999] Meloan, S. (1999). The java hotspot performance engine: An in-depth look. *Technical Whitepaper*. Online; accessed 23-August-2019. `https://www.oracle.com/technetwork/java/whitepaper-135217.html`.

[Mestayer et al., 2000] Mestayer, M., Carman, D., Asavapibhop, B., Barbosa, F., Bonneau, P., Christo, S., Dodge, G., Dooling, T., Duncan, W., Dytman, S., et al. (2000). The clas drift chamber system. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 449(1-2):81–111.

[Mestayer et al., 2017] Mestayer, M., Elouadrhiri, L., Young, G., and Burkert, V. D. (2017). Clas12 - drift chambers (dc). Technical report, Jefferson Laboratories.

[Moravánszky, 2003] Moravánszky, A. (2003). Dense matrix algebra on the gpu. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:352–380.

[Oracle, 2019] Oracle (2019). Java visualvm. Online; accessed 25-June-2019 `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jvisualvm.html`.

[Pelegri-Llopart and Graham, 1988] Pelegri-Llopart, E. and Graham, S. L. (1988). Optimal code generation for expression trees: an application burs theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308. ACM.

[Perkins and Perkins, 2000] Perkins, D. H. and Perkins, D. H. (2000). *Introduction to high energy physics*. Cambridge University Press.

[Pharr and Fernando, 2005] Pharr, M. and Fernando, R. (2005). *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional.

[Pinchoff, 2005] Pinchoff, N. (2005). Introduction to rf linear accelerators.

[Prechelt, 2000] Prechelt, L. (2000). An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29.

[Prechelt et al., 1999] Prechelt, L. et al. (1999). Comparing java vs. c/c++ efficiency differences to interpersonal differences. *Commun. ACM*, 42(10):109–112.

[Rauch et al., 1965] Rauch, H., Tung, F., and Striebel, C. (1965). Maximum likelihood estimates of linear dynamic systems. *American Institute of Aeronautics and Astronautics (AIAA) Journal*, 3 i8:1445–1450.

[Runge, 1895] Runge, C. (1895). Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178.

[Sauer, 2012] Sauer, T. (2012). *Numerical Analysis*. Pearson.

[Sauli, 1977] Sauli, F. (1977). Principles of operation of multiwire proportional and drift chambers. Technical report, Cern.

[Sherman and Morrison, 1950] Sherman, J. and Morrison, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127.

[Simon, 2006] Simon, D. (2006). *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons.

[Sirunyan et al., 2017] Sirunyan, A. M., collaboration, C., et al. (2017). Particle-flow reconstruction and global event description with the cms detector. *JINST*, 12(10):P10003.

[Strang, 2005] Strang, G. (2005). *Linear Algebra and its Applications (4th ed.)*. Cengage Learning.

[Strassen, 1969] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356.

[Su and Chang, 1996] Su, C.-T. and Chang, F. C. (1996). Quick evaluation of determinants. *Applied mathematics and computation*, 75(2-3):117–118.

[Sun Microsystems, 1997] Sun Microsystems (1997). Java code conventions. Technical report, Oracle. Online; accessed 13-June-2019
`https://www.oracle.com/technetwork/java/codeconventions-150003.pdf`.

[Taylor, 1997] Taylor, J. R. (1997). *An introduction to error analysis*. University Science Books.

[van de Geijn, 2011] van de Geijn, R. A. (2011). Notes on cholesky factorization.

[Van den Braak et al., 2011] Van den Braak, G.-J., Nugteren, C., Mesman, B., and Corporaal, H. (2011). Fast hough transform on gpus: Exploration of algorithm trade-offs. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 611–622. Springer.

[Wake, 2004] Wake, W. C. (2004). *Refactoring workbook*. Addison-Wesley Professional.

[Wenisch et al., 2006] Wenisch, T. F., Wunderlich, R. E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. C. (2006). Simflex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31.

[Xu and Rountev, 2008] Xu, G. and Rountev, A. (2008). Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th international conference on Software engineering*, pages 151–160. ACM.

[Yeo et al., 2019] Yeo, B., Lee, M. J., Semertzidis, Y. K., and Kuno, Y. (2019). Gpgpu tracking in the comet phase-i cylindrical drift chamber. *arXiv preprint arXiv:1908.01949*.

[Zarchan and Musoff, 2013] Zarchan, P. and Musoff, H. (2013). *Fundamentals of Kalman filtering: a practical approach*. American Institute of Aeronautics and Astronautics, Inc.

[Ziegler, 2013] Ziegler, V. (2013). Clas12 software document - hall b 12 gev upgrade. Technical report, Jefferson Lab. Software Draft.