UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

LAUREA TRIENNALE IN FISICA

GENERAL PURPOSE TOOLS FOR LONGITUDINAL BEAM DYNAMICS STUDIES

CERN-THESIS-2023-035
//2023

Supervisor:
Prof. Davide Galli

Student:
Fabiana Lauro
917451

Co-supervisor:
Dr. Lucio Rossi

External supervisor:
Dr. Simon Albright

Anno Accademico 2021-2022

Acknowledgements

First, I would like to express my deepest gratitude to Simon, my CERN supervisor, for the way in which he has accompanied and led me through this work, and for his enormous availability and patience: no matter how busy he was, he always found time to dedicate to me, whenever I needed it. I also greatly appreciated his encouragement and his concrete example to be curious, to learn, to experiment and to get involved. During this year I have grown a lot, and I owe it largely to Simon.

I am also very thankful to all the rest of the team who I worked with at CERN, starting from our section leader, Heiko. They have created the perfect work environment, and I really couldn't have wished for anything better.

Finally, and not less importantly, I am immensely grateful to my university supervisors, Prof. Galli and Prof. Rossi, for accepting to supervise this rather "unconventional" thesis, and for always being available and supportive. It is not something that I take for granted, and I appreciated it very much.

Contents

Introduction **4**

1 CERN **5**
1.1 What is CERN . . . 5
1.2 CERN accelerator complex . . . 5
1.3 Accelerators at CERN . . . 7
1.3.1 Linear accelerators . . . 7
1.3.2 Synchrotrons . . . 9
1.3.3 Decelerators . . . 10
1.4 Experiments at CERN . . . 11
1.4.1 LHC experiments . . . 11
1.4.2 SPS experiments . . . 13
1.4.3 PS experiments . . . 14
1.4.4 PSB experiments . . . 14
1.5 Cycles, supercycles, beam types, destinations . . . 15
1.6 CERN's internal structure . . . 17
1.6.1 SY-RF-BR . . . 17

2 Longitudinal beam dynamics in synchrotrons **19**
2.1 Introduction . . . 19
2.2 Motion of the synchronous particle . . . 19
2.2.1 Energy gain . . . 19
2.2.2 Energy ramping and phase of the synchronous particle . . . 20
2.2.3 Synchronicity . . . 21
2.3 Bunches and synchrotron oscillations . . . 22
2.3.1 Dispersion effects and transition energy . . . 22
2.3.2 Synchrotron oscillations . . . 24
2.4 Phase space . . . 25
2.5 Longitudinal equations of motion . . . 26
2.5.1 Discretisation . . . 28
2.6 Hamiltonian of the system . . . 29

3 Signals **31**
3.1 Introduction . . . 31

3.2 PyJapc . . . 31
3.3 NXCALS . . . 33
3.4 Example of some parameters for longitudinal analysis . . . 36
3.4.1 Voltage signals . . . 36
3.4.2 BCT . . . 36
3.4.3 Tomoscope . . . 38

4 PyDAq **41**
4.1 Introduction . . . 41
4.1.1 Challenges . . . 41
4.2 Functionality . . . 42
4.2.1 Live acquisitions from a single accelerator . . . 42
4.2.2 Historical data extraction from a single accelerator . . . 44
4.2.3 Multiple machines acquisitions . . . 45
4.2.4 Parametric scans . . . 46
4.2.5 Acquisitions from input file . . . 49
4.3 Saving structure . . . 49
4.3.1 Logging messages . . . 50
4.4 Code architecture for live acquisitions . . . 50
4.4.1 Parameter and device dataclasses . . . 51
4.4.2 Acquisition control . . . 51
4.4.3 Single machine logger . . . 51
4.4.4 Multiple machines logger . . . 52
4.4.5 Example of a multiple machine acquisition . . . 52
4.5 Practical uses . . . 53
4.6 Performance and limits . . . 55
4.7 Possible improvements and future work . . . 58

5 PyLAn **59**
5.1 Introduction . . . 59
5.2 Pipeline and Data Manager . . . 60
5.2.1 The Pipeline class . . . 60
5.2.2 The Data Manager class . . . 61
5.3 Pipeline evaluation . . . 62
5.3.1 The `eval_speed` function . . . 62
5.3.2 The `Profiling` class . . . 62
5.4 Future work . . . 63
5.5 Uses . . . 64

Conclusions **66**

Glossary **67**

A Examples of input files for PyDAq acquisitions **70**

Introduction

The European Organization for Nuclear Research (CERN) is an international organisation dedicated to the development of particle physics and nuclear physics. There, research is done mainly by accelerating particles in particle accelerators and making them collide with fixed targets or with some other particles. My thesis work took place at CERN, in the framework of a technical student programme lasting from March 2022 to April 2023, under the supervision of Dr. Simon Albright, on the "Longitudinal beam observation and measurement instrument data analysis routines" project.

At CERN particles are accelerated thanks to electric fields oscillating inside radio-frequency cavities. In order to maintain, and increase, the quality of the beam, it is crucial to study in deep how the beam interacts with the RF cavities. Moreover, in order to deliver to the experiments a beam meeting the requests of the users, some beam manipulations on the longitudinal plane have to be performed, by acting on the electric field oscillating in the cavities.

Data acquisition and analysis are essential tools to study and manipulate the longitudinal motion of the beam circulating in the accelerators, diagnose problems and improve the beam quality. Nevertheless, a unified method to acquire and analyse data has never been implemented, resulting in the existence of many different ways to perform similar tasks, leading to lack of efficiency, reliability and robustness.

The aim of this work is to simplify and standardise the ways of acquiring data from the beam observation tools and analysing them, through the development of two python-based libraries. This is needed to improve the workflow of beam studies and operations, making it less time consuming and easier to obtain, compare and analyse data. The first library which was developed is called PyDAq and is useful for data acquisition, the second one is called PyLAn, which focuses on data analysis.

After some introductory chapters, in which my work will be contextualised and justified, the two libraries that I've developed will be described in detail. In particular, chapter 1 is dedicated to introducing CERN and its activity, in chapter 2 I will focus on longitudinal beam dynamics. In chapter 3, an overview on the signals produced by the accelerators devices, and the ways to acquire them, will be given. Finally, chapters 4 and 5 are dedicated to describing and explaining the libraries that I've developed, respectively PyDAq and PyLAn. After this, some conclusions and considerations will be exposed.

Chapter 1

CERN

1.1 What is CERN

CERN (*Organisation européenne pour la recherche nucléaire*, formerly *Conseil Européen pour la Recherche Nucléaire*) is an international scientific organisation established for the purpose of collaborative research into high-energy particle physics and nuclear physics. At the moment it comprises 23 member states.

The main objective of CERN is to study atomic nuclei and fundamental particles, looking into the standard model, and discover new physics; this is mostly done by accelerating particles to very high energies and make them collide with each other or with stationary targets. A recent and significant discovery is the observation of a particle compatible with the Higgs boson, as predicted by the standard model (Chatrchyan et al., 2012; Aad et al., 2012).

Built on the Franco-Swiss border near Geneva, CERN is one of the biggest scientific laboratories in the world, counting several particle accelerators and two particle decelerators, delivering beam to the LHC (Large Hadron Collider), as well as many other experiments and experimental facilities.

1.2 CERN accelerator complex

CERN accelerator complex is composed of various accelerators, chained one after the other in order to reach very high energies, providing beam to a big variety of experiments. A schematic of the complex as of 2022 is reported in fig. 1.1. Although many experiments work with different types of particles, the LHC injectors complex works mostly with protons and lead ions.

In the following paragraphs, a description of the "journey" that these particles undertake from one machine to another, until collision, is given. All of the accelerators and experiments mentioned will be described more in detail in the next sections of this chapter.

The protons journey starts with a H^- source that injects into Linac4. The ions are then stripped of their electrons and the remaining protons are injected and accelerated in the PSB. From there, they can be sent to the ISOLDE facility, or continue their journey through the PS. In the second

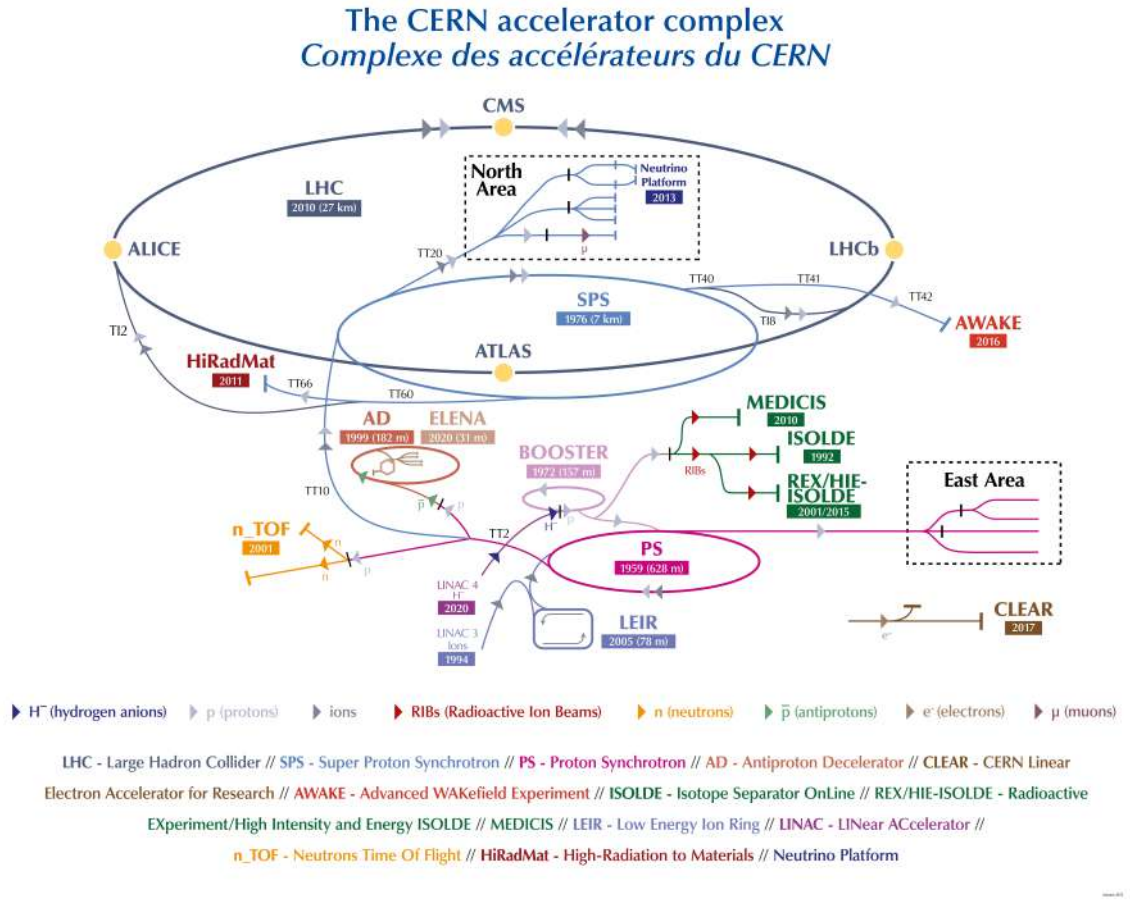


Figure 1.1: CERN accelerator complex in 2022

case, after being accelerated in the PS, the protons can be injected in the SPS, or be sent to the East Area, to the nTOF facility or to the AD. If injected into the SPS, they are further accelerated and then sent to the North Area, to the AWAKE experiment, to HiRadMat, or injected into the LHC, where they are further accelerated and then collide in the collision points.

The lead ions journey, instead, starts from vaporised lead, which gets ionised by an electron current and injected in Linac3, and continues through LEIR, PS, and SPS, ending up in the LHC. The ions are accelerated in every machine, and undergo two electron strips: one in Linac3 (from Pb^{29+} to Pb^{54+}), and another one in the transfer line between PS and SPS, which leaves only bare nuclei (Pb^{82+}). When arriving in the LHC, the nuclei collisions take place in the collision points.

1.3 Accelerators at CERN

CERN's particle accelerators are mainly of two types: linear accelerators (linacs) and synchrotrons. Most of them make up the LHC injectors complex, although some others have different purposes and uses.

1.3.1 Linear accelerators

A linear accelerator (also simply called linac) is a device that accelerates charged particles (electrons, protons, ions) along a straight trajectory. Linacs can be of three different types, depending on how the acceleration is performed: DC linacs, RF linacs and induction linacs. At CERN, RF linacs are used: the particles gain energy thanks to electromagnetic fields confined in resonant cavities fed by sinusoidal time-varying power sources. (Alesini, 2022)

The main advantage of linacs is their capability to produce high-energy, high-intensity particle beams of excellent quality in terms of beam emittance and energy spread. Moreover, in the case of light particle (like electrons or positrons) acceleration, they exhibit low synchrotron radiation losses. On the contrary, the main drawback is the fact that they require a large number of cavities to reach a desired energy, since the beam passes only once in each accelerating structure (Alesini, 2022). CERN accelerator complex includes various linacs, used for different purposes. Two of them are part of the LHC injector complex, both used as injectors into synchrotrons, performing the first step in the particle acceleration process:

- **Linac4** accelerates negative hydrogen ions (H^-) up to 160 MeV, then the ions are stripped of their two electrons, leaving only protons at injection to the Proton Synchrotron Booster (PSB).
- **Linac3** accelerates ions (mostly Pb) up to 2.5 keV/nucleon, and injects them into the Low Energy Ion Ring (LEIR).

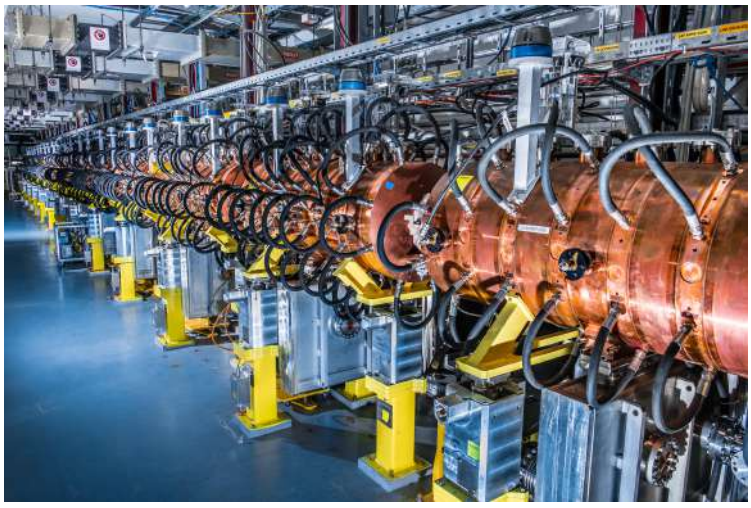


Figure 1.2: A photo of Linac4

In addition to these two, some other linear accelerators are present at CERN: some of them are hosted in experimental facilities (HIE-ISOLDE, AWAKE...), while **CLEAR (CERN Linear Electron Accelerator for Research)** constitutes an independent linear accelerator. It accelerates electrons and hosts several experiments with different purposes, from prototyping and validation of accelerator components to irradiation tests for space and high-energy physics, from studies of high-gradient acceleration methods to dosimetry for medical applications.

The main components of linear accelerators are RF (radio frequency) cavities, metallic chambers able to support an oscillating electromagnetic field, shaped to resonate at specific frequencies. An accelerating cavity is constructed such that the beam can pass through it, inside a vacuum pipe, and that the electric field points in the direction of particle motion (beam axis). If the electric field is well synchronised with the passing particles bunches, these latter are accelerated and gain energy (see chapter 2). The electric field is provided to the cavities by powerful RF amplifiers, via a power coupler. They are driven by the low-level RF system, which provides the necessary small-amplitude signals at the correct frequency and phase. Often, low-level RF systems also include electronic feedback loops, that apply corrections to the signals sent to the cavities, improving the longitudinal beam stability. (Damerau, 2022)

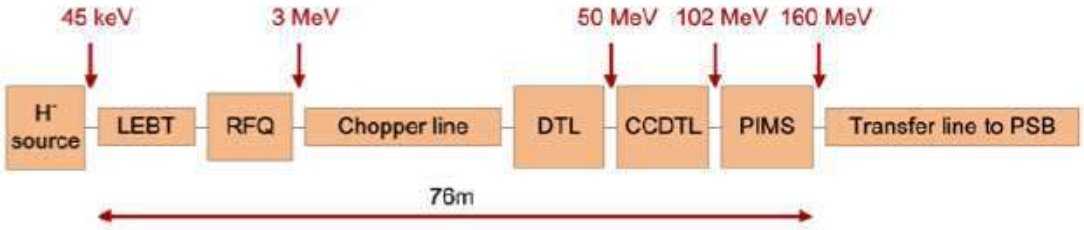


Figure 1.3: A schematic layout of Linac4 (Roncarolo et al., 2010)

RF cavities are mainly used to accelerate the beam and keep it focused in the longitudinal plane, but linacs also employ some special cavities, called RFQ (RF quadrupoles), able to act not only in the longitudinal plane, but also in the transversal one, with the aim of focusing the beam. Their working principle is similar to that of a magnetic quadrupole, with no transverse field in the centre of the structure and a transverse electric field that increases linearly with beam off-axis, with a focusing effect in one plane and a defocusing effect in the other (see fig. 1.5). The succession of many quadrupoles, rotated of 90° with respect to the previous one, results in an overall focusing.

Figure 1.4: A Drift Tube Linac (DTL) RF cavity, employed in Linac4

The reason behind the use of electric, instead of magnetic, fields for focusing is connected to the dependency of the magnetic force on velocity ($\vec{F} = q\vec{v} \times \vec{B}$). RFQ are usually employed for the first stage of acceleration in linacs: they deal with particles still at low energies, having therefore a low v, making electric fields more effective on the beam, compared to magnetic fields. Being able to accelerate the beam and focus it in the longitudinal and transversal plane at the same time, RFQ cavities are

also useful to contrast the beam instabilities and space charge effects, very strong in low energy beams.

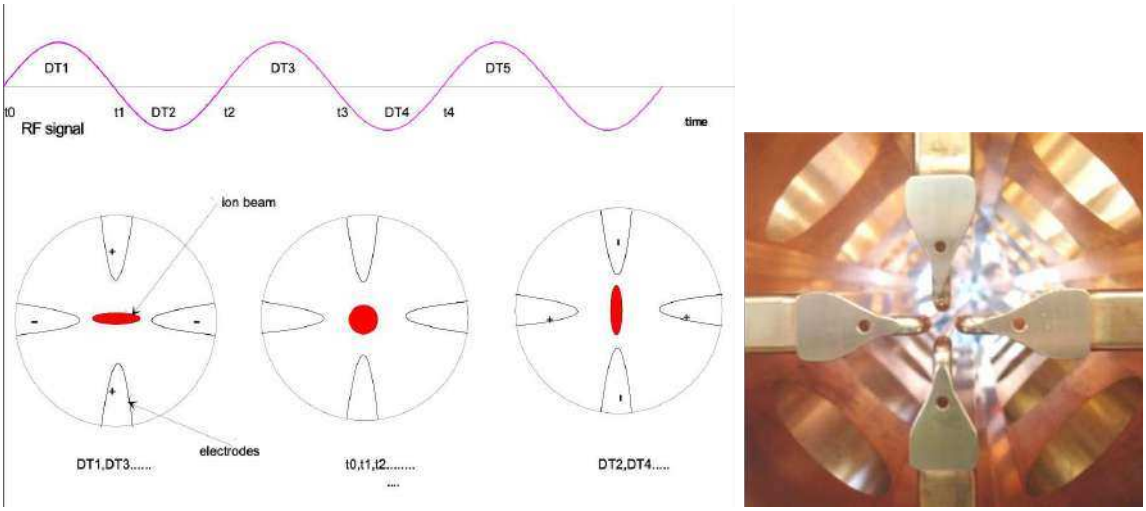


Figure 1.5: On the left is shown a scheme of the RFQ transversal focusing, on the right is shown a photo of an RFQ in Linac4 Alesini (2022)

1.3.2 Synchrotrons

A synchrotron is a circular accelerator, in which the particles travel in a fixed closed-loop trajectory. This means that the particles pass through the accelerating structures multiple times, receiving an energy boost at each turn, allowing higher energies to be reached. The defining characteristic of synchrotrons is that magnetic and electric fields need to be synchronised with the movement of the particles.

In synchrotrons, like in linacs, the acceleration of the particles and the longitudinal focusing of the beam are due to RF cavities (fig. 1.6), while the closed trajectory of the beam is assured by magnetic dipoles, which, by creating a homogeneous magnetic field, bend the particle trajectories according to the Lorentz force. In the LHC, due to the very high energy of the particles, very strong magnetic fields are needed in order to maintain the beam in the nominal trajectory. For this reason, the LHC employs superconducting dipoles, that need to be kept at a temperature of 1.9 K (fig. 1.6). Magnets are also used to focus the beam on the transverse plane: magnetic quadrupoles (whose working principle is the same as the electric quadrupoles, explained above) and higher-order magnets, such as sextupoles or octupoles, keep the beam confined both vertically and horizontally.

The LHC injectors chain counts five synchrotrons:

- **Protron Synchrotron Booster (PSB)**: four superposed synchrotron rings, all with a circumference of 157 m. It receives protons coming from Linac4 and accelerates them from 160 MeV to 2 GeV (for the beam to the PS).

Figure 1.6: On the left is shown an LHC module containing RF cavities, on the right is shown a cryogenic dipole in the LHC

- **Low Energy Ion Ring (LEIR)**: receives lead ions from Linac3 and accelerates them from 4.2 MeV to 72 MeV/u (fig. 1.7).
- **Proton Synchrotron (PS)**: the oldest synchrotron at CERN. It has a circumference of 628 m and accelerates protons from the PSB (up to 26 GeV) or heavy ions from LEIR (up to 5.9 GeV/u).
- **Super Proton Synchrotron (SPS)**: with a circumference of 6.9 km, it takes protons or ions from the PS and accelerates them to provide beams for the LHC or for several experiments. It can reach energies up to 450 GeV for protons and 177 GeV/u for ions.
- **Large Hadron Collider (LHC)**: the world's largest and highest energy particle accelerator (Cern website), it is situated on average 100 m under ground and it has a circumference 26.7 km. It currently circulates two beams in opposite directions with energies of 6.8 TeV per beam. In addition to accelerating the beams, it also makes them collide at four different sites, which correspond to the four large experiments situated around the collider: ALICE, ATLAS, CMS and LHCb. The collisions generate a debris in the form of new particles that fly out in all directions, which are detected and studied by the experiments.

1.3.3 Decelerators

At CERN, antimatter is created, stored and studied thanks to antimatter decelerators.

Antiprotons are created by firing proton beams from the PS into a target. When created, the antiprotons are highly energetic, and they have to be slowed down in order to be stored, manipulated (for example to make anti-hydrogen) and studied.

This is done in CERN's two antimatter decelerators:

- **Antiproton Decelerator (AD)**: A synchrotron whose electric fields decelerate the antiprotons, while a technique known as "cooling" reduces their emittance. Through several cycles of cooling and deceleration, the antiprotons are slowed down to about a tenth of the speed of light.
- **Extra Low ENergy Antiproton (ELENA)**: it is a newer deceleration ring, which slows the

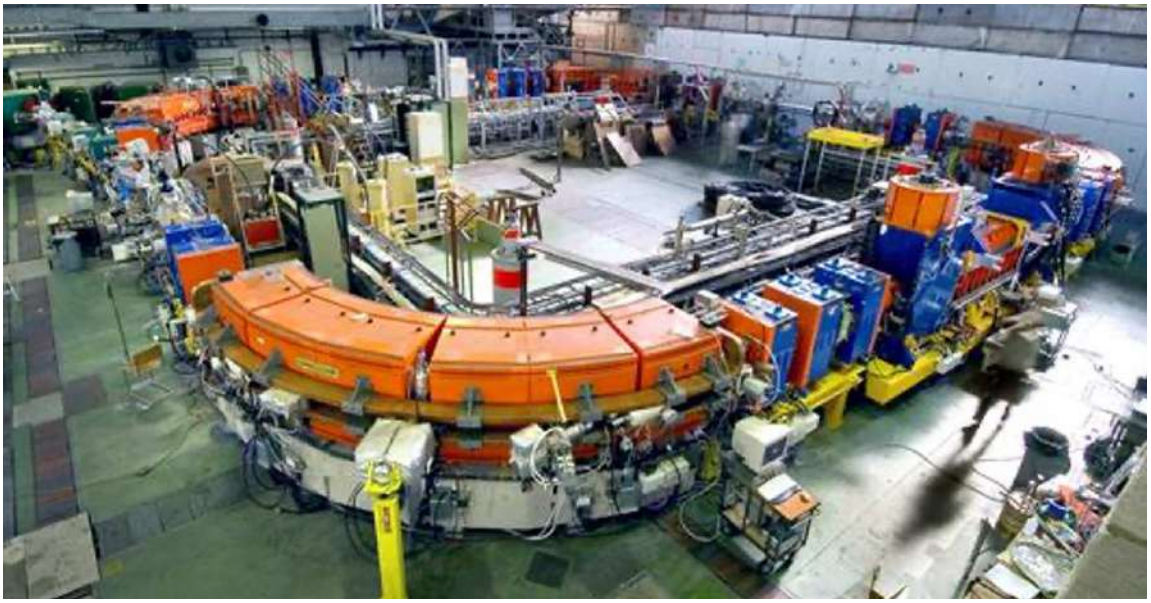


Figure 1.7: A photo of LEIR. The orange structures are the dipoles, while the blue and orange structures are the quadrupoles.

antiprotons even more, reducing their energy by a factor of 50, from 5.3 MeV to just 0.1 MeV, and increasing the beam density thanks to an electron cooling system. With ELENA, the number of antiprotons that can be trapped increases by a factor of 10 to 100 (fig. 1.8).

1.4 Experiments at CERN

At CERN, a very wide range of experiments takes place. The most well known are placed around the LHC and analyse the collisions of the particle beams in the accelerator's collision points. In addition to them, fixed-target experiments, antimatter experiments and experimental facilities make use of the LHC injector chain for their studies.

1.4.1 LHC experiments

The main LHC experiments are four, each one having its own detector in a huge cavern along the LHC ring:

- **ATLAS**: a general-purpose experiment, investigating a wide range of physics, from the Higgs boson to extra dimensions and dark matter. The ATLAS detector is the largest volume detector ever constructed for a particle collider (Atlas website), and it's composed of six different detecting subsystems, arranged in layers around the collision point. It is able to record the trajectory, momentum, and energy of the particles flying out from the collision, allowing them to be individually identified. A huge magnet system bends the paths of charged particles so that their momenta can be measured (fig. 1.9a).

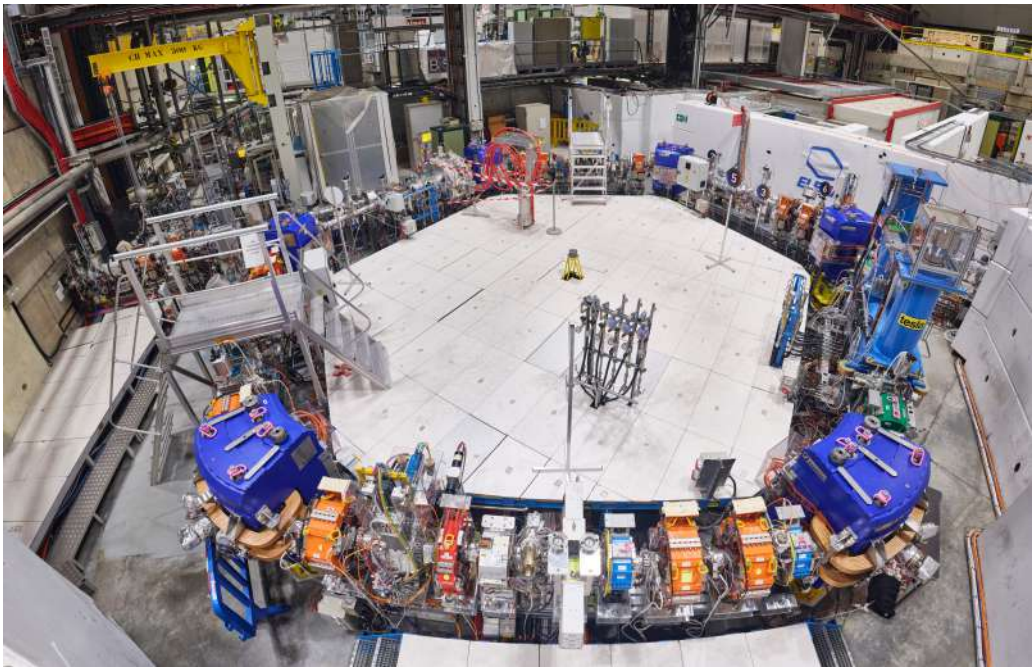
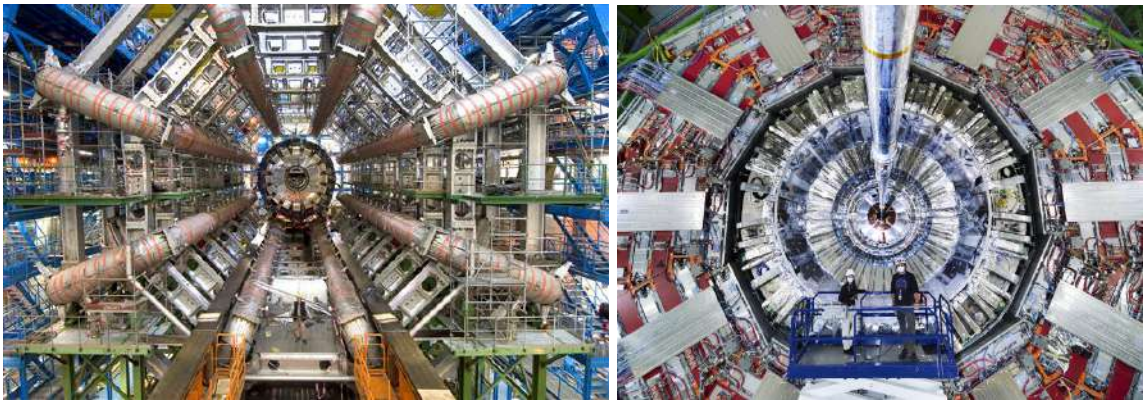


Figure 1.8: A photo of ELENA.

- **CMS (Compact Muon Solenoid)**: also a general-purpose experiment, having similar research aims to ATLAS, but different methods: CMS uses a smaller but heavier detector, built around a huge solenoid magnet, specially designed to detect muons very accurately. Having two collaborations performing the same research with different methods is very important for validation of the discoveries (fig. 1.9b).
- **ALICE (A Large Ion Collider Experiment)**: a detector dedicated to heavy-ion physics. It mainly studies the quark-gluon plasma state, a state of matter in which the quarks overcome their bonds with the gluons. This state can be reached by colliding ion nuclei at extreme energy densities. When the quark-gluon plasma expands and cools, it progressively gives rise to the particles that constitute the matter of our universe today: by studying this process, the ALICE collaboration attempts to find the answer to open questions in the field of quantum chromodynamics (QCD), or regarding the phenomenon of confinement.
- **LHCb (Large Hadron Collider beauty)**: an experiment specialised in studying the decay of particles containing b and anti-b quarks ("B mesons"), with the aim of investigating the differences between matter and antimatter. Differently from the other main LHC experiments, LHCb doesn't employ a big enclosed detector, but it uses a series of subdetectors specialised in detecting the particles thrown forwards by the collision (forward particles). This is due to the fact that the B mesons created in the collisions, and the particles they decay into, tend to stay close to the line of the beam pipe. LHCb's subdetectors cover a length of 20 metres, starting from the collision point, and each one of them is specialised in measuring a different characteristic of the particles produced in the collisions.



(a) The ATLAS detector during LS2. *(b) The CMS detector during LS2.*

Figure 1.9

In addition to these, five other smaller experiments take place along the LHC: **TOTEM**, **LHCf**, **MoEDAL-MAPP**, **FASER** and **SND@LHC**.

1.4.2 SPS experiments

Besides providing beam to the LHC, the SPS serves some other experiments and facilities, including:

- **North Area**: an experimental area situated on one side of the SPS. Consisting in two large surface halls and an underground cavern, it houses a number of different experiments, whose research covers several fields: from phase transition between hadrons and quark-gluon plasma, to rare kaons decays; from radiation processes in strong electromagnetic fields, to interactions between visible matter and dark matter.
- **CERN Neutrino Platform**: a facility situated close to the North Area, where fundamental research in neutrino physics is performed.
- **HiRadMat (High-Radiation to Materials)**: an installation dedicated to the test of material samples and accelerator components under beam impact. It uses high-intensity, short-pulse particle beams, which get delivered to a dedicated target area, where various experiments can be transported.
- **AWAKE (Advanced WAKefield Experiment)**: a proof-of-principle experiment, investigating wakefield plasma acceleration of electrons using a proton bunch as a driver. Plasma wakefield acceleration is an alternative way for accelerating charged particles, making use of accelerating structures called wakefields. These consist in periodic charge density perturbations in a plasma, induced by driver beams that can be either laser pulses or energetic particles (Gschwendtner and Muggli, 2019). The AWAKE experiment is the first one using a proton bunch as a driver. Being able to create a very steep acceleration gradient, wakefield acceleration can overcome some limitations of the conventional microwave radio-frequency acceleration, such as such as the size of the accelerating structures and the maximum achievable

acceleration gradient (Gschwendtner et al., 2022).

1.4.3 PS experiments

Some other facilities take beam from the PS, in particular:

- **East Area**: an experimental area containing four beam lines. It hosts the CLOUD (Cosmics Leaving Outdoor Droplets) experiment, aimed to study the possible link between galactic cosmic rays and cloud formation, the DIRAC (Dimeson Relativistic Atom Complex) experiment, studying the decay of unstable "pionium atoms" in order to gain a deeper understanding of the strong force, and the IRRAD and CHARM facilities, where various irradiations experiments take place.
- **nTOF (Neutron Time Of Flight)**: a pulsed neutron source coupled to a 200-metre flight path. It is designed to study neutron-related processes, and in particular neutron-nucleus interactions. It can provide neutrons with energies ranging from a few meV to several GeV, starting from a pulsed beam of protons coming from the PS, made to collide with a lead target. The neutrons resulting from the collisions are slowed down and then guided through a beam pipe to an experimental area, located at a distance of 185 m from the lead target. A sample is placed in the neutron beam, and by detecting the reaction products and determining the neutron kinetic energy by their time-of-flight, the reaction probability can be reconstructed as a function of the incident neutron energy.
- **Antimatter factory**: the complex composed by the AD and ELENA, which hosts various experiments on antimatter. Three of them (AEgIS, ALPHA and GBAR) trap antihydrogen atoms, with the aim of understanding how antimatter reacts to gravity. Also the ASACUSA experiment deals with antihydrogen, but it manages to create a beam of antiatoms, transporting them to a region where no disturbing fields are present, to perform precision spectroscopy. The BASE experiment compares the magnetic momenta of protons and antiprotons to look for differences between matter and antimatter, and finally the PUMA experiment is designed to carry antiprotons from CERN's Antimatter Factory to the ISOLDE facility.

1.4.4 PSB experiments

The PSB delivers beam to one experimental facility, called **ISOLDE (Isotope mass Separator On-Line DEvice)**, which is dedicated to the study of atomic nuclei. It is a source of low-energy beams of radioactive nuclides, which are created from a proton beam coming from the PSB which, by colliding into some thick targets, creates a large variety of atomic fragments. From these, the radioactive nuclei are ionised, extracted and separated thanks to two mass separators (GPS and HRS), forming a low-energy beam, that is then delivered to various experimental stations, or further accelerated in the HIE-ISOLDE linear accelerator. ISOLDE also drives the **MEDICIS** facility, whose aim is contributing to medical research (help diagnose cancers and other diseases, deliver precise radiation doses to treat diseased cells) by producing novel radioisotopes.

1.5 Cycles, supercycles, beam types, destinations

As previously discussed, the particles can have many different final destinations. Depending on its destination, every beam has precise characteristics, that match the requests and the needs of the users. As an example, a beam destined to the ISOLDE facility has a very high intensity[1], a large emittance[2] and a relatively low energy, while a beam for the LHC requires a lower intensity and a very small emittance.

The injectors' activity is organised in *cycles* and *supercycles*. One cycle corresponds to a complete set of accelerator parameters that have to play through in order to produce a given beam, with the required characteristics. It starts before the beam's injection, and it ends after its extraction. A common way to visualise a cycle is to look at the magnetic field produced in the accelerator (which is proportional to the momentum of the beam), sometimes combining it with the intensity of the circulating beam. One or multiple beams from one machine can be injected in the same cycle of the downstream one For example, in fig. 1.10, multiple batches from the PS are injected in the same LHC cycle, to be then accelerated and collided.

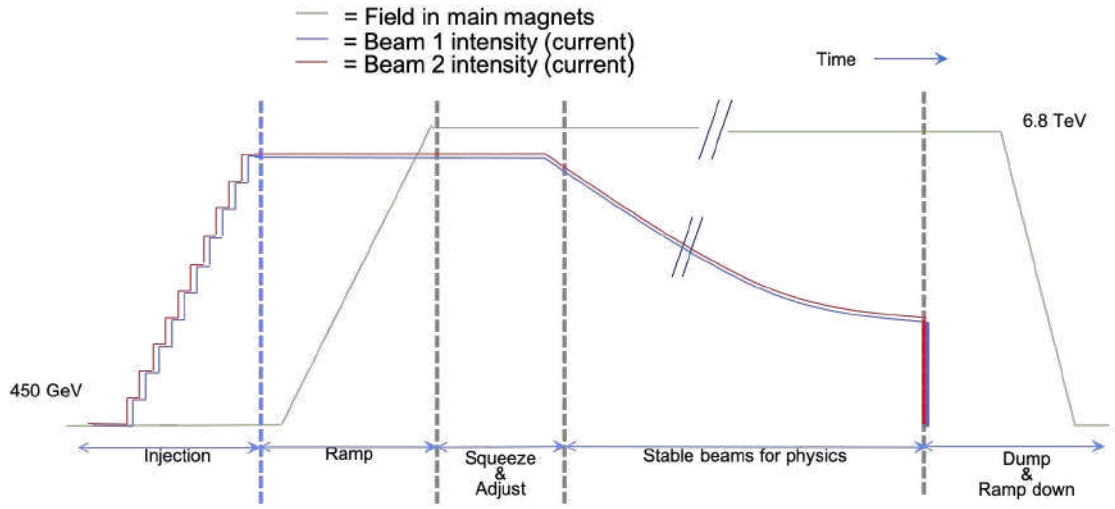


Figure 1.10: A schematisation of an LHC cycle.

In the PS-Complex (Linac4, PSB, PS), as well as in the SPS, AD, ELENA and LEIR, the length of every cycle is a multiple of a so called "basic period" of 1.2 s. The Linac4 and the PSB are always run with a cycle of 1.2 s, whereas the PS can have cycles of different lengths, corresponding to one, two or three basic periods (1.2 s, 2.4 s or even 3.6 s), depending on the specific operation. SPS cycles are generally longer, but of course the SPS can only receive beams from the PS at time instants that correspond to multiples of the PS-complex basic period.

Every cycle is identified by a PLS user, a "container" of the collection of parameters that must be applied in oder to produce a specific beam. These collections of parameters are stored in a database

[1]The beam intensity corresponds to the number of particles composing it.
[2]The beam emittance is linked to the particles spread in space.

called LSA (LHC Software Architecture), under some LSA cycle names. Every PLS user is linked to a specific LSA cycle name, so that every time that a PLS user is played in the machines, the right settings are applied. One or multiple cycles with the same PLS user can be found in one supercycle; in the second case, all the beams with the same PLS user will have the same characteristics (and usually the same destination).

A supercycle is a pre-defined sequence of cycles to be played in a machine, one after the other. Usually the same supercycle is repeated many times, making the cycle sequence periodic. A supercycle usually includes cycles of different beam types, with different destinations, and it determines the schedule of beam injection and extraction An example of an SPS supercycle is shown in fig. 1.11.

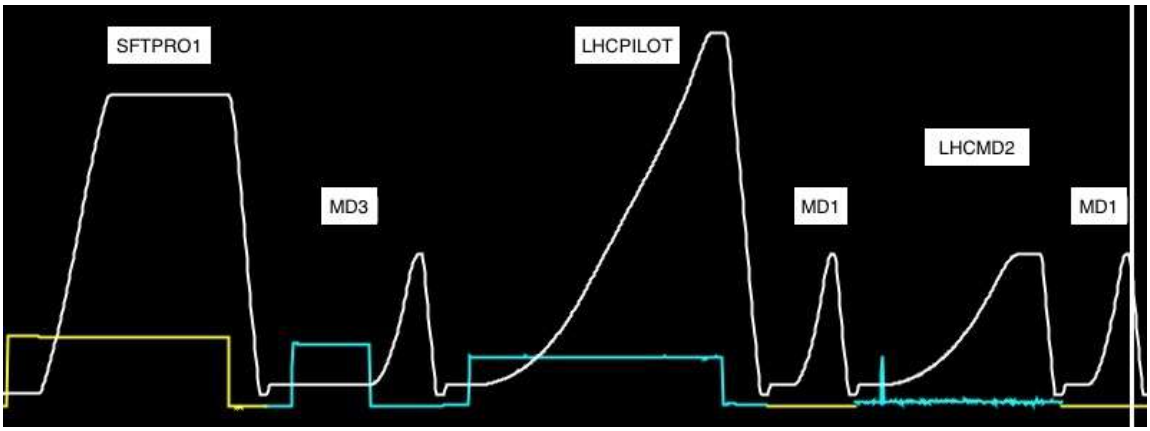


Figure 1.11: A supercycle played in the SPS on 24/03/2023. The white line corresponds to the magnetic field, the blue or yellow line represents the beam intensity. For every cycle, the corresponding PLS user is reported.

Looking at cycles and injections, it becomes clear why a variety of destinations and experiments is needed: in this way, in fact, the accelerators efficiency is maximised. To show this, let's suppose that every injected beam was only used to fill the LHC, and consider the injection scheme depicted in fig. 1.12.

Two consecutive beams in the PSB are accelerated and injected in the PS. But during the energy ramp in the PS, no more beams from the PSB are needed, meaning that the booster will have some empty cycles. Similarly, after the four PS beams are injected in the SPS, during acceleration, no beams from the PS are needed, leading to empty cycles both in the PS and in the booster. The same thing, in even bigger proportions, would happen at the end of the LHC filling: when the collisions take place, for many hours no more beam is needed from the injectors. Having a variety of experiments allows all the cycles to be filled: when the beam is not needed from the downstream accelerator, the beams can be delivered to the other facilities and experimental areas, or some cycles can host MD (Machine Development) beams, beams played not to be provided to physics users, but to perform research activities and machine studies.

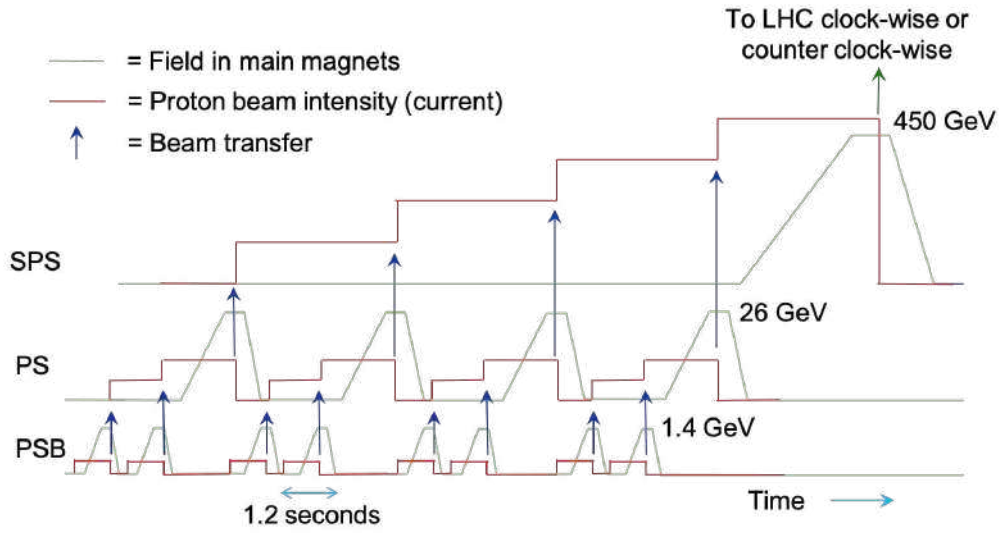


Figure 1.12: An example of injection scheme of an LHC beam.

1.6 CERN's internal structure

CERN is internally organised in a hierarchical structure, with *sections* being the smallest organisational units, composed of people working on similar topics. Sections are structured inside *groups*, and groups are then structured inside *departments*. Finally, departments are grouped inside *sectors*. CERN counts in total 5 sectors: Director-General (DG), Accelerators and Technology (ATS), Finance and Human Resources (FHR), International Relations (IR) and Research and Computing (RCS),

During my stay at CERN, I worked in the ATS sector, and more precisely in the Accelerator Systems (SY) department, Radio Frequency (RF) group, Beam and RF studies (BR) section.

1.6.1 SY-RF-BR

The SY department is responsible for various accelerator beam-related technical systems, including beam instrumentation, beam transfer, electrical power converters, radio frequency, targets, collimators and absorbers. It is involved in the maintenance and operation of the already existing systems in the CERN complex, as well as the development and deployment of new devices (SY website).

Inside the SY department, the RF group is in charge of operations on the accelerators radio-frequency (RF) systems, which include RF cavities, RF power amplifiers, low-level RF (LLRF), and RF controls. In particular, the RF group is responsible for the design, construction or procurement, testing, operation, consolidation and maintenance of the existing RF systems, along with conception and prototyping of RF systems for future projects and studies (SY-RF website).

Finally, inside the RF group, the BR section deals with everything that concerns the interactions of

beams with RF systems and their environment, primarily in the longitudinal plane (BR webpage). It uses a combination of analytical calculations and numerical simulations to obtain their goals, which mainly consist in:

- beam operations: managing and controlling the beams in the accelerators, in order to keep them into specifications
- machine developments: comparing beam measurements with simulations, to improve the understanding of the phenomena and their predictions
- performance improvements (e.g. mitigating longitudinal instabilities) and upgrades of existing accelerators
- impedance studies: measuring and calculating the impedance of various machine elements
- improvements of beam diagnostics in the longitudinal plane
- design and production of new RF systems, for existing and future accelerators
- development of the Beam Longitudinal Dynamics code (BLonD), used in numerous particle simulations, essential to support beam measurements, RF manipulations and to predict beam performance after upgrades

Machine Development

In addition to simulating the longitudinal behaviour of the beam using BLonD code, professionals working in SY-RF-BR often need to work with real beam data. This often involves having to manipulate the beam by changing some machine parameters. In these cases, a dedicated MD beam is prepared. After circulating in the machine of interest, the MD beam is dumped instead of going on through the downstream machine.

The typical workflow during an MD session consists in setting some accelerator parameters and observing the subsequent reaction of the beam, recording acquisitions to be able to analyse them later. Often, the subsequent data analysis combines live data with additional data retrieved from a database. However, not all signals are stored in the database, which makes live data acquisition necessary in most cases.

In any case, a large amount of data has to be collected from the accelerators and analysed: from this the need to have reliable tools that support the process of setting accelerator parameters, saving and analysing data emerges.

Chapter 2

Longitudinal beam dynamics in synchrotrons

In the following paragraphs the basic concepts of longitudinal beam dynamics will be exposed. These concepts constitute the foundation of all the studies conducted in the BR section.

The layout of this chapter follows the description of Longitudinal Beam Dynamics given in (Tecker, 2022)

2.1 Introduction

In synchrotrons, the longitudinal motion of the particles is governed by alternating electric fields, that oscillate inside an RF (radiofrequency) cavity with a certain angular frequency ω_{rf}:

$$\mathcal{E}(t) = \mathcal{E}_0 \sin(\omega_{\mathrm{rf}} t) \tag{2.1}$$

A particle with charge q that enters the cavity at time t will be exposed to an electric field $\mathcal{E}(t)$ and it will consequently experiences a force $F = \frac{\mathrm{d}p}{\mathrm{d}t} = q\mathcal{E}(t)$. Since the electric field changes in time, it is crucial to synchronise its frequency with the motion of the particle.

The alternated electric field is also used to keep the particles inside the synchrotron in well controlled bunches. The following sections will first treat the motion of a single reference particle, and then it will analyse the motion of the whole bunch.

2.2 Motion of the synchronous particle

2.2.1 Energy gain

A charged particle in the synchrotron will gain energy from the electric field.

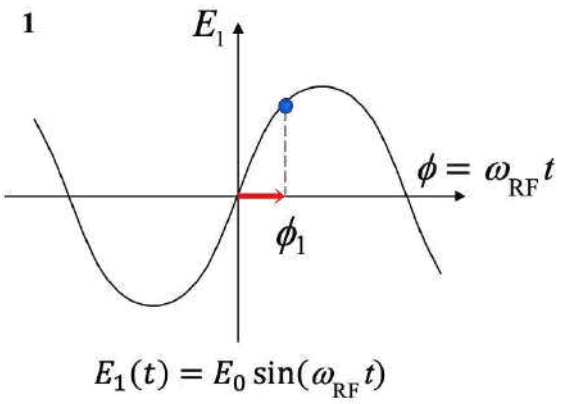


Figure 2.1: A sinusoidal oscillating electric field.
(Tecker, 2022)

In relativistic dynamics, the total energy of a body is:

$$E^2 = E_0^2 + E_{kin}^2 = E_0^2 + p^2c^2 \tag{2.2}$$

with E_0 being the rest mass of the body and c being the speed of light.

By differentiating, we get that:

$$\mathrm{d}E = v\ \mathrm{d}p, \tag{2.3}$$

from which follows:

$$\frac{\mathrm{d}E}{\mathrm{d}z} = v\frac{\mathrm{d}p}{\mathrm{d}z} = \frac{\mathrm{d}p}{\mathrm{d}t} = qE_z. \tag{2.4}$$

Thus, the energy gained from the field in the z direction is:

$$\Delta E = q\int_z E_z\ \mathrm{d}z = q\Phi, \tag{2.5}$$

where Φ represents an electric potential.

2.2.2 Energy ramping and phase of the synchronous particle

During the energy ramping, the magnetic field changes in order to keep the radius of the trajectory constant. The change in the magnetic field is strictly connected to the phase that the particle needs to have, to be able to gain the right amount of energy from the electric field.

The magnetic field required to keep particles with momentum p on a given orbit can be obtained from the *magnetic rigidity* relationship, defined as follows:

$$p = qB\rho, \tag{2.6}$$

where ρ is the bending radius of the particle immersed in the magnetic field B.

Keeping the bending radius constant, and deriving with time, we obtain

$$\dot{p} = q\rho\dot{B} \tag{2.7}$$

For one turn in the machine, this results in

$$(\Delta p)_{\text{turn}} = q\rho\dot{B}T = \frac{2\pi R q\rho\dot{B}}{v}, \tag{2.8}$$

where R is the physical radius of the machine, and T is the revolution period.

From 2.3, we get that $(\Delta E)_{\text{turn}} = v(\Delta p)_{\text{turn}} = 2\pi q\rho R\dot{B}$. Since the energy is provided by the RF system, it is also true that $\Delta E = q\hat{V}\sin\phi_{\text{s}}$.

From this relation we can deduct the stable phase for a particle during acceleration, called *synchronous phase* ϕ_{s}:

$$\sin\phi_{\text{s}} = 2\pi\rho R\frac{\dot{B}}{\hat{V}_{\text{rf}}} \quad \text{or} \quad \phi_{\text{s}} = \arcsin\left(2\pi\rho R\frac{\dot{B}}{\hat{V}_{\text{rf}}}\right). \tag{2.9}$$

2.2.3 Synchronicity

As mentioned before, in a synchrotron the key is to synchronise the particle with the electric field, so that the former will be affected by the right value of the latter. Therefore, the frequency of the RF electric field must be a multiple of the revolution frequency of the particle ω. In terms of angular frequencies,

$$\omega_{\text{rf}} = h\omega, \tag{2.10}$$

where h is an integer called *harmonic number*. h also represents the number of stable synchronous particle locations, equidistantly spaced around the circumference of the accelerator.

During the energy ramping, the velocity of the particle increases, and so does its revolution frequency. The acceleration system needs to remain synchronised with the particle during its acceleration, so the radio frequency has to change consequently (RF frequency sweep):

$$f = \frac{v(t)}{2\pi R} = \frac{1}{2\pi R}\frac{c^2}{E(t)}q\rho B(t) \tag{2.11}$$

Substituting $E^2 = E_0^2 + p^2c^2 = m_0c^2 + (q\rho B(t))^2c^2$, we obtain how the RF must vary in order to follow the variation in the B field:

$$\frac{f_{\text{rf}}}{h} = \frac{c}{2\pi R}\sqrt{\frac{B(t)^2}{\left[m_0c^2/(qc\rho)\right]^2 + B(t)^2}}. \tag{2.12}$$

This effect is stronger at lower energies (i.e. at lower relativistic γ): in a highly relativistic regime, an energy gain doesn't correspond to a significant change in the particle's speed, which also leads to a smaller RF variation. As $v \to c$, the B field term in the equation becomes large compared to the $m_0c^2/(qc\rho)$ term, which leads to $f \to c/(2\pi R)$. This can be noticed by comparing different machines at CERN. For example, in the PSB, for the beams going to the PS, the revolution frequency varies

from 994kHz to 1.81MHz, and it corresponds to an energy variation from 160MeV to 2GeV, while in the LHC the change in the revolution frequency is only $2 \cdot 10^{-6}$, despite an energy variation from 450GeV to 6.8TeV.

2.3 Bunches and synchrotron oscillations

So far, we have studied the motion of only one particle, always having exactly the nominal energy and following precisely the nominal trajectory. A (real or idealised) particle with this characteristics is called *synchronous particle*.

Inside accelerators, though, circulate many particles at once, all having different phases and momenta. These particles are naturally grouped into *bunches*, steady longitudinal distributions of particles around the synchronous particle. The bunches are kept together by *synchrotron oscillations*, oscillations of the particles between extreme values in momentum and phase (or other phase-space coordinates) with respect to the reference point (i.e. the synchronous particle) (Wiedemann, 2015). They work by taking advantage of the sinusoidal shape of the electric field in RF cavities, in conjunction with the particles experiencing different electric fields according to their speed and position within the bunch; this is explained in detail in the coming sections. The idea at the base of this is that particles that arrive in the RF cavity before the synchronous particle see a field that makes them "slow down", and vice versa.

2.3.1 Dispersion effects and transition energy

The key parameter to understand under what conditions a particle can "slow down" is the *slip factor* η, defined as the relative change in revolution period with momentum, given by:

$$\eta = \frac{\mathrm{d}T/T}{\mathrm{d}p/p}. \tag{2.13}$$

For $\eta > 0$, an increase in momentum causes a lower revolution frequency, while in the opposite case ($\eta < 0$) an increase in the momentum corresponds to a higher revolution frequency.

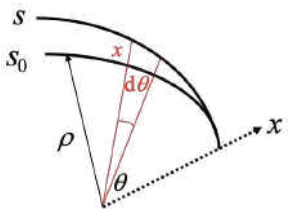


Figure 2.2: Orbit length change for particles with different momenta. (Tecker, 2022)

In order to study η, it is necessary to begin with another parameter, the *momentum compaction factor* α_c, describing the relative change in orbit length C with momentum:

$$\alpha_\mathrm{c} = \frac{\mathrm{d}C/C}{\mathrm{d}p/p}. \tag{2.14}$$

Considering two particles, one with momentum p and another with momentum $p+\mathrm{d}p$, their relative elementary path-length difference is:

$$\frac{\mathrm{d}l}{\mathrm{d}s_0} = \frac{\mathrm{d}s - \mathrm{d}s_0}{\mathrm{d}s_0} = \frac{\rho \mathrm{d}\theta - (\rho + x)\mathrm{d}\theta}{\rho \mathrm{d}\theta} = \frac{x}{\rho}. \tag{2.15}$$

Introducing the *dispersion function* from the transverse beam optics $D_x = x/(\mathrm{d}p/p)$, we get:

$$\frac{\mathrm{d}l}{\mathrm{d}s_0} = \frac{D_x}{\rho}\frac{\mathrm{d}p}{p}, \tag{2.16}$$

which leads to a total change in the trajectory circumference of:

$$\mathrm{d}C = \int_C \mathrm{d}l = \int \frac{x}{\rho}\mathrm{d}s_0 = \int \frac{D_x}{\rho}\frac{\mathrm{d}p}{p}\,\mathrm{d}s_0. \tag{2.17}$$

We can rewrite the momentum compaction factor as:

$$\alpha_\mathrm{c} = \frac{\mathrm{d}C/C}{\mathrm{d}p/p} = \frac{1}{C}\int \frac{D_x}{\rho}\mathrm{d}s_0, \tag{2.18}$$

and since $\rho = \infty$ in the straight sections, we get

$$\alpha_\mathrm{c} = \frac{\langle D_x \rangle_\mathrm{m}}{R}, \tag{2.19}$$

where the average $\langle\cdot\rangle_\mathrm{m}$ is considered over the bending magnets only.

Since the revolution period is $T = C/(\beta c)$, its relative change is:

$$\frac{\mathrm{d}T}{T} = \frac{\mathrm{d}C}{C} - \frac{\mathrm{d}\beta}{\beta} = \alpha_\mathrm{c}\frac{\mathrm{d}p}{p} - \frac{\mathrm{d}\beta}{\beta}, \tag{2.20}$$

and from the relativistic relation $\mathrm{d}p/p = \gamma^2\,\mathrm{d}\beta/\beta$ we obtain:

$$\frac{\mathrm{d}T}{T} = \left(\alpha_\mathrm{c} - \frac{1}{\gamma^2}\right)\frac{\mathrm{d}p}{p}, \tag{2.21}$$

which leads to

$$\eta = \alpha_\mathrm{c} - \frac{1}{\gamma^2}. \tag{2.22}$$

This means that there is one value of γ for which η becomes zero, called *transition energy* γ_t, whose value is given by:

$$\gamma_\mathrm{t} = \frac{1}{\sqrt{\alpha_\mathrm{c}}}. \tag{2.23}$$

The two terms α_c and $1/\gamma^2$ represent two opposing trends of a particle whose momentum is increased: on one side, it tends to increase the circumference of its trajectory, resulting in a higher revolution period, on the other side it tends to increase its velocity, resulting in a lower revolution period.

For $\gamma < \gamma_t$, the increase in the velocity of the particle is the dominating effect, and an increase in the particle's momentum will result in a higher revolution frequency (the particle "speeds up"), while for $\gamma > \gamma_t$, the particle has a velocity close to the speed of light, so its velocity does not change significantly anymore, and the effect of the longer path-length dominates. In this case, an

increase in the particle's momentum will result in a lower revolution frequency (the particle "slows down").

Transition crossing occurs with hadrons and ions, but not with electrons. Owing to the relatively small rest mass of electrons, their relativistic gamma factor is so large that the injection energy is already greater than the transition energy. Hence, tipically the electrons will stay above transition during the whole acceleration cycle.

2.3.2 Synchrotron oscillations

In order to keep the particles bunched, the particles that arrive before the reference particle have to be "slowed down" (i.e. their revolution frequency has to be reduced), and vice versa. This can be done by applying the right electric field, keeping in mind that the effects of energy gain are opposite below and above transition.

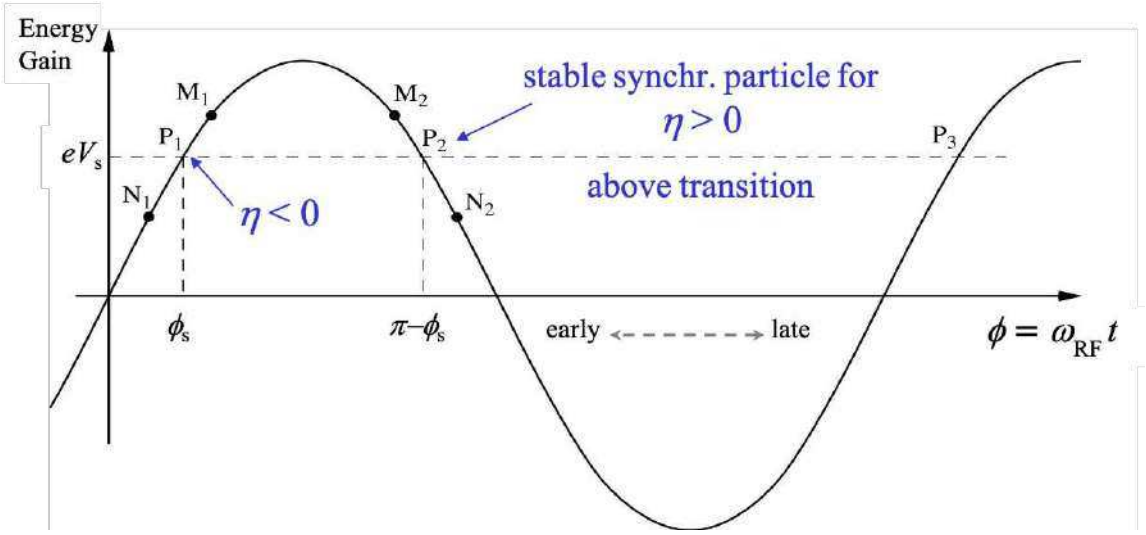


Figure 2.3: Energy gain as a function of particle phase; oscillations are stable around the synchronous-phase particle P1 below transition and around the synchronous-phase particle P2 above transition.
(Tecker, 2022)

In fig. 2.3, the points P_1, P_2, and P_3 represent the fixed points of motion: their energy gain is such that, after one turn, the particle reaches the structure again with the same phase. The particle N_1 arrives earlier compared to P_1, while the particle M_1 arrives later compared to P_1.

Below transition, N_1 gains less energy than the nominal one, and its revolution period will become larger, meaning that at the next turn it will appear closer in time to P_1. On the other hand, particle M_1 will gain more energy and reduce its delay compared to P_1. This makes the point P_1 a stable point for the longitudinal oscillation below transition: particles slightly away from it will experience forces that will reduce their deviation. Point P_3 is a stable point too, while point P_2 produces the opposite effect and therefore it is an unstable point.

After transition, the situation is the opposite: while P_1 and P_3 are unstable points, P_2 becomes

a stable point: a particle that arrives too early (M_2) will gain more energy, its orbit will increase and so will do its revolution time; thus, the particle will arrive later on the next turn, closer to the synchronous phase. Similarly, a particle that arrives late (N_2) will gain less energy and travel a shorter orbit, also moving towards the synchronous phase.

Transition crossing is a very delicate moment: the stable synchronous phase becomes unstable and vice versa, so the RF system needs to make a rapid change of RF phase (a jump of $\pi - 2\phi_s$) when crossing the transition energy. Moreover, at transition, the velocity change and the path-length change with momentum compensate for each other, so the revolution frequency becomes independent of the momentum deviation. In this case, the oscillations stop and the particles that are not at the synchronous phase get the same non-nominal energy gain in each turn, accumulating the energy error. This can lead to a loss of the particles, due to dispersive effects. Therefore, transition has to be passed quickly, in order to minimise the losses.

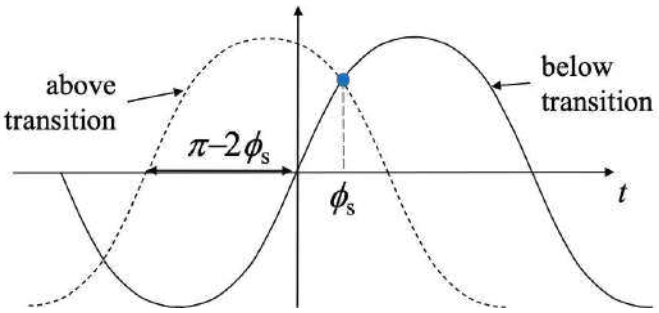


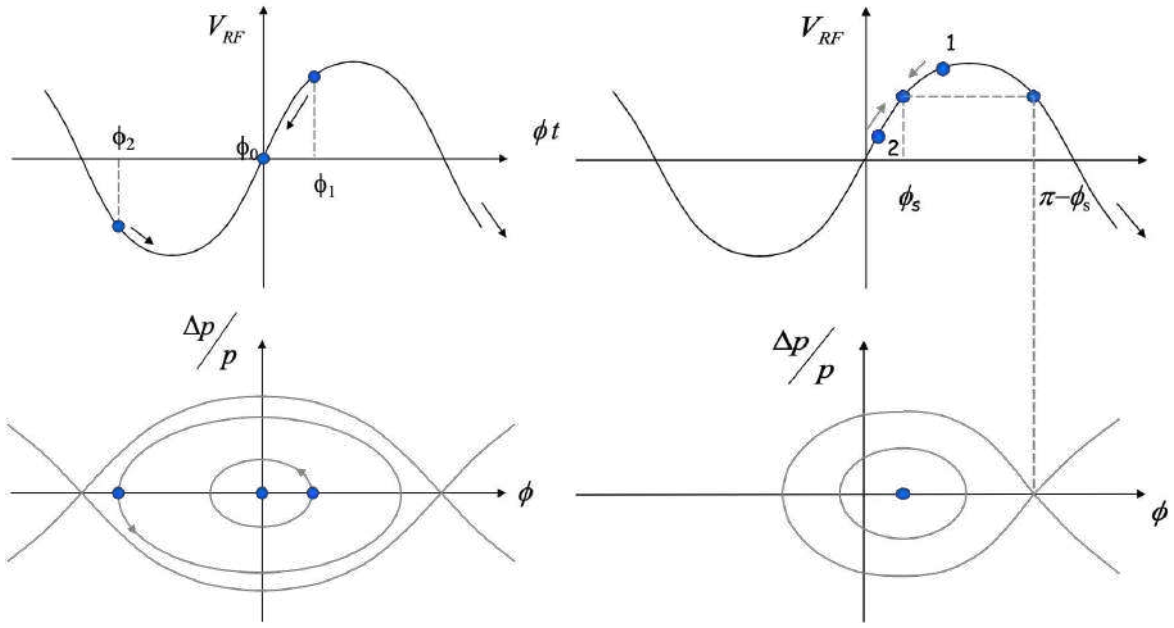
Figure 2.4: At transition crossing, RF phase needs to quickly change by $\pi - 2\phi_s$. (Tecker, 2022)

2.4 Phase space

The particles oscillations can be visualised by plotting the particle distribution in the phase space. A very common choice for the axes is to have the phase ϕ on the x-axis and the energy deviation ΔE from the synchronous particle on the y-axis, but it is not rare to find the time t on the x-axis (making the two variables canonically conjugated), or, on the y-axis, the momentum deviation Δp or the derivative of the phase $\dot{\phi}$.

The trajectory of the synchronous particle in the phase space is a fixed point, while the synchrotron oscillations draw closed lines around the synchronous particle. The unbound motions draw open lines in phase space, and bound and unbound trajectories are separated by a line, called *separatrix*, that sets the border between stable and unstable motion. The area within the separatrix is called *RF bucket* and corresponds to the maximum acceptance in phase space for a bound motion. Particles located inside the RF bucket will tipically follow closed phase-space trajectories. The area in the phase space containing all the particles is called *longitudinal emittance*.

The phase extension of the bucket is maximum for a stationary bucket (i.e. with no acceleration, for $\phi_s = 180°$ or $\phi_s = 0°$). During acceleration, the buckets get smaller, both in length and energy acceptance. In fact, as shown in fig. 2.5, the restoring force goes to zero when ϕ reaches $\pi - \phi_s$, and



(a) Stationary bucket *(b) Acceleration*

Figure 2.5: The phase space plot in relation with the applied voltage, both for a stationary bucket and during energy ramping.

it becomes non-restoring beyond (both below and above transition). Hence the extreme amplitude for a stable motion during acceleration is not π anymore, but it is $\pi - \phi_s$.

2.5 Longitudinal equations of motion

A non-synchronous particle will have:

- energy $E = E_s + \Delta E$
- angular frequency $\omega = \omega_s + \Delta\omega$
- azimuth orbital angle $\theta = \theta_s + \Delta\theta$

The time evolution of the azimuth angle of an arbitrary particle is:

$$\theta(t) = \int_t \omega \, \mathrm{d}t, \tag{2.24}$$

as can be seen in fig. 2.6. Thus, the phase of an arbitrary particle with respect to the RF is, at any time:

$$\phi(t) = -h\Delta\theta = -h \int_t \Delta\omega \, \mathrm{d}t, \tag{2.25}$$

where the minus sign arises from the fact that a particle that is ahead arrives earlier, i.e. at a smaller RF phase.

By differentiating with time and including the phase slip factor $\eta = -p_s/\omega_s\,(\mathrm{d}\omega/\mathrm{d}p)$, we get:

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = -h\Delta\omega = h\eta\omega_s\frac{\Delta p}{p_s} \tag{2.26}$$

And using the differential relationship $\mathrm{d}E/E = \beta^2\mathrm{d}p/p$, we finally obtain the first-order equation for the phase slippage:

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = \frac{h\eta\omega_s^2}{\beta_s^2 E_s}\left(\frac{\Delta E}{\omega_s}\right) \tag{2.27}$$

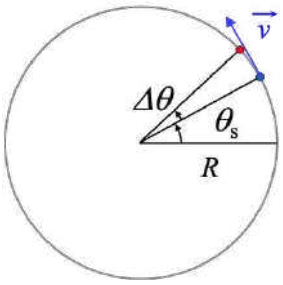


Figure 2.6: Visualisation of the azimuth angle change
(Tecker, 2022)

which can also be rewritten, using the relativistic relation $p = (\beta^2 E)/(\omega R)$, as:

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = \frac{h\eta\omega_s}{p_s R}\left(\frac{\Delta E}{\omega_s}\right). \tag{2.28}$$

The second first-order equation follows from the energy gain of a particle:

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \frac{\omega}{2\pi}q\hat{V}\sin\phi, \tag{2.29}$$

so that the time derivative of the energy difference becomes (in first order approximation):

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\Delta E}{\omega_s}\right) \approx \frac{1}{\omega}\frac{\mathrm{d}E}{\mathrm{d}t} - \frac{1}{\omega_s}\frac{\mathrm{d}E_s}{\mathrm{d}t} = \frac{q\hat{V}}{2\pi}\left(\sin\phi - \sin\phi_\mathrm{s}\right). \tag{2.30}$$

Combining and deriving 2.28 and 2.30, we obtain the second-order, non-linear equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{\beta_s^2 E_s}{h^2\eta\omega_s^2}\frac{\mathrm{d}\phi}{\mathrm{d}t}\right] - \frac{q\hat{V}}{2\pi}\left(\sin\phi - \sin\phi_\mathrm{s}\right) = 0, \tag{2.31}$$

which can be also written as:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{Rp_s}{h\eta\omega_s}\frac{\mathrm{d}\phi}{\mathrm{d}t}\right] - \frac{q\hat{V}}{2\pi}\left(\sin\phi - \sin\phi_\mathrm{s}\right) = 0. \tag{2.32}$$

The parameters within the bracket are, in general, slowly varying in time: if we assume R, p_s, ω_s and η to be constant, we get:

$$\ddot{\phi} + \frac{\Omega_\mathrm{s}^2}{\cos\phi_\mathrm{s}}\left(\sin\phi - \sin\phi_\mathrm{s}\right) = 0, \tag{2.33}$$

where Ω is the synchrotron (angular) frequency, defined as:

$$\Omega_{\mathrm{s}}^2 = \frac{-h\eta\omega_s q\hat{V}\cos\phi_{\mathrm{s}}}{2\pi R p_s} = -\omega_s^2 \frac{h\eta q\hat{V}\cos\phi_{\mathrm{s}}}{2\pi\beta_s^2 E_s} \tag{2.34}$$

Considering small phase deviations from the synchronous particle, we can approximate:

$$\sin\phi - \sin\phi_{\mathrm{s}} = \sin(\phi_{\mathrm{s}} + \Delta\phi) - \sin\phi_{\mathrm{s}} = \sin\phi_s\cos\Delta\phi + \cos\phi_s\sin\Delta\phi - \sin\phi_s \approx \cos\phi_{\mathrm{s}}\Delta\phi, \tag{2.35}$$

which leads to to the harmonic oscillator equation:

$$\ddot{\phi} + \Omega_{\mathrm{s}}^2\Delta\phi = 0, \tag{2.36}$$

Stability is obtained when Ω_{s} is real, which reduces to the condition:

$$\eta\cos\phi_{\mathrm{s}} < 0, \tag{2.37}$$

which tells us that the stable region for the synchronous phase depends on wether the energy is higher or lower than the transition energy, as explained in 2.3.1.

Another important parameter is the *synchrotron tune*, defined as

$$Q_{\mathrm{s}} = \frac{\Omega_{\mathrm{s}}}{\omega_0}, \tag{2.38}$$

which corresponds to the number of synchrotron oscillations per revolution in the machine. Generally $Q_{\mathrm{s}} \ll 1$: to complete one synchrotron oscillation, it typically takes the order of several tens to several hundreds turns.

2.5.1 Discretisation

In Hamiltonian formalism, the RF electric field is considered to be uniformly distributed in an accelerator. In reality, RF cavities are localised in a short section of a synchrotron, so the synchrotron motion is more realistically described by the mapping equations (Lee, 2019):

$$\begin{cases} \Delta E_{n+1} = \Delta E_n + q\hat{V}(\sin\phi - \sin\phi_{\mathrm{s}}) \\ \phi_{n+1} = \phi_n + \dfrac{2\pi h\eta}{\beta^2 E}\Delta E_{n+1} \end{cases} \tag{2.39}$$

At its n^{th} passage through the RF cavity, the particle gains or loses energy, as described by the first equation (*kick* equation).The next RF phase ϕ_{n+1} depends on the new energy owned by the particle, as described by the second equation (*drift* equation). The discrete equations of motion can also be used to simulate numerically the longitudinal motion of a particle in the phase space.

2.6 Hamiltonian of the system

To write the Hamiltonian describing the system, we introduce a new convenient variable $W = \Delta E/\omega_s$. The equations of motion then become:

$$\begin{aligned} \frac{\mathrm{d}\phi}{\mathrm{d}t} &= \frac{h\eta\omega_0}{p_0 R} W, \\ \frac{\mathrm{d}W}{\mathrm{d}t} &= \frac{e\hat{V}}{2\pi} \left(\sin\phi - \sin\phi_\mathrm{s}\right). \end{aligned} \tag{2.40}$$

The two variables ϕ and W are canonical, since it is true that

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = \frac{\partial H}{\partial W}, \quad \frac{\mathrm{d}W}{\mathrm{d}t} = -\frac{\partial H}{\partial \phi} \tag{2.41}$$

with the following Hamiltonian:

$$H(\phi, W, t) = \frac{e\hat{V}}{2\pi} \left[\cos\phi - \cos\phi_\mathrm{s} + (\phi - \phi_\mathrm{s}) \sin\phi_\mathrm{s}\right] + \frac{1}{2} \frac{h\eta\omega_0}{p_0 R} W^2 \tag{2.42}$$

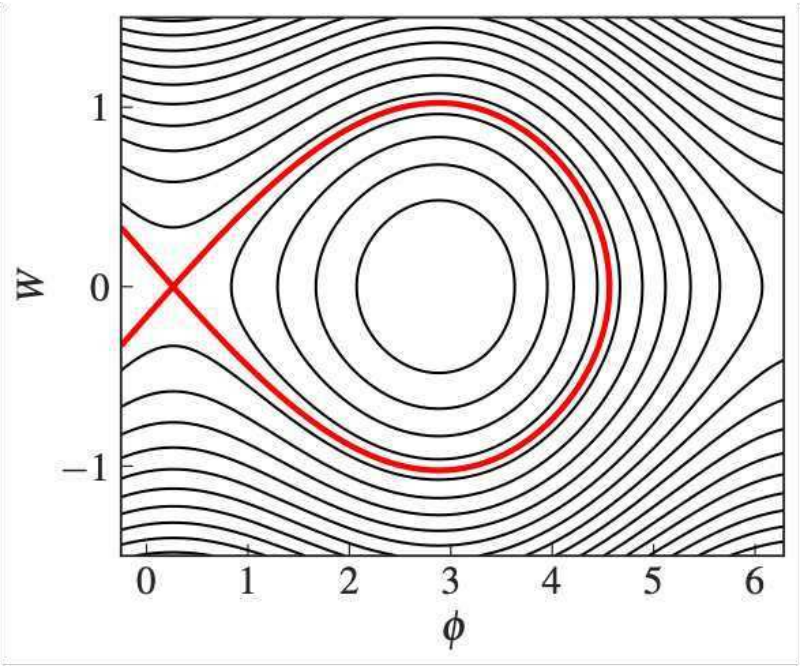
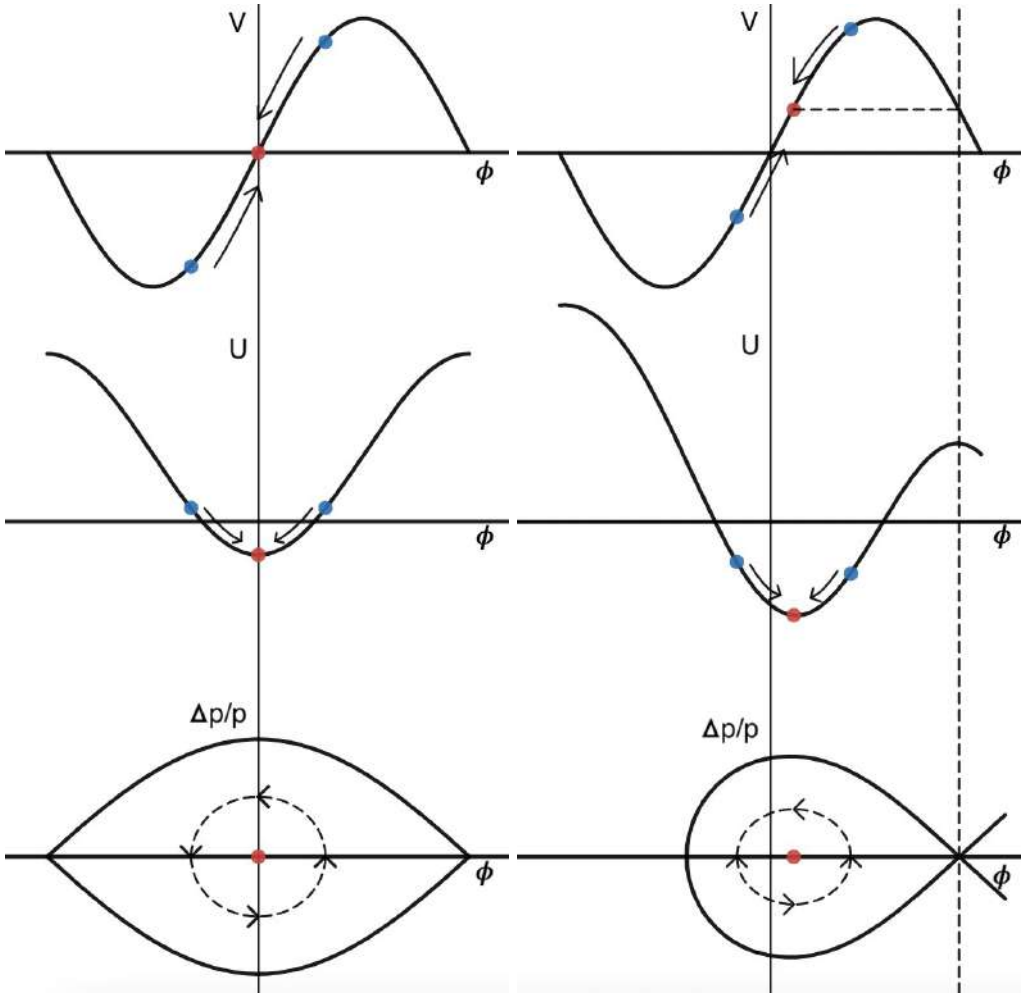


Figure 2.7: Projection of a Hamiltonian function above transition, for a synchronous phase of $\phi_s = 170°$. Equipotential lines are possible phase-space trajectories; the separatrix is shown as a solid red line.

The potential described by the Hamiltonian has the shape of a well: it is symmetric when the bunch is not accelerated, and asymmetric otherwise.



(a) Stationary bucket *(b) Acceleration*

Figure 2.8: The potential well and the phase space plot in relation with the applied voltage, both for a stationary bucket and during energy ramping.

Chapter 3

Signals

3.1 Introduction

Communication with the machines is made possible thanks to a wide range of devices that control the accelerator equipment: they collect information from the beam instruments, set values and states on active elements, monitor the health of sub-systems. A complex Control System allows users to remotely control and monitor these devices (Fortescue-Beck, 2020a).

The devices are directly controlled by front-end computers (FEC), real-time computers that communicate directly with the accelerator equipment. On top of them sits a software layer called FESA (Front-End Software Architecture), a real-time application framework that allows accessing the FEC. Almost all the control of the accelerators is done through FESA.

Every accelerator device is linked to a FESA class, which is internally organised in *properties*, which are then divided in *fields*. Every parameter is identified by a string containing its device, property and field names, with the syntax `DEVICE/Property#field`. The FESA parameters can be of different types, with different characteristics: they can be read-only, read-write or write-only, and their value can change depending on the device's internal behaviour. Some parameters are cycle-dependent (or PPM , standing for pulse-to-pulse modulation), and their value can be different for every cycle, while for some other signals the value is the same for all cycles.

Users can interact live with the devices, by setting parameters or getting data, through a Java API called JAPC (or, through its python wrapper, PyJapc). A significant part of the data published by the devices also gets logged in the NXCALS a database, from where users can fetch historical data from previous cycles.

3.2 PyJapc

PyJapc is a Python interface to JAPC (Java API for Parameter Control), a unified Java API to access different types of devices present in the CERN control system (JAPC user guide, 2023). It mainly consists of a class, called `PyJapc`, that can get or set one or multiple parameters, or subscribe

to specific signals in order to receive continuous updates.

A `PyJapc` instance can be initialised with a timing selector, a string containing the accelerator name and the PLS user to get the data from, hence describing the PLS user for which a PyJapc operation (GET or SET) should act. Examples of timing selectors are: `PSB.USER.NORMGPS` (PSB beam destined to ISOLDE), `CPS.USER.TOF` (PS beam destined to nTOF), or `SPS.USER.SFTPRO1` (SPS beam destined to the North Area).

To request a value from a parameter for a particular selector, two methods of the `PyJapc` class are available: `getParam` returns the latest value published by the device, while `getNextParamValue` waits for the device to publish the next value, and returns that.

```
import pyjapc

pjc = pyjapc.PyJapc("CPS.USER.LHC1") #Create pyjapc object and initialise it to take
                                        data from the LHC1 cycle in the PS


pjc.getParam("PR.BCT.ST/Samples#samples") #Get the latest BCT value

pjc.getNextParamValue("PR.BCT.ST/Samples#samples") #Get the next BCT value
```

Values are returned in python native data types, mainly as simple values like `int`, `float`, `str` or `bool`, as 1D or 2D numpy arrays, or as python dictionaries.

Set

The `setParam` method of the `PyJapc` class sends data to a parameter for a particular selector. It can be used only on the parameters with write access.

```
import numpy as np
import pyjapc

pjc = pyjapc.PyJapc("PSB.USER.TOF") #Create pyjapc object and initialise it to take
                                        data from the TOF cycle in the PSB

pjc.setParam("BA3.FGVRFH1/Settings#amplitudes",  np.zeros(10)) #Set the voltage for
                                        harmonic 1 in the PSB to be equal to zero
```

Subscription

Subscriptions are used to be notified of parameter value changes. When subscribing to a parameter, every new value that gets published by the corresponding device gets returned to the user. The updates come periodically according to the selector used to create the subscription and to the type of the parameter. For cycle-dependent parameters, a new value is expected at every cycle played for that particular selector, while for cycle-independent parameters, a new value is expected only when it gets changed in the device.

Subscriptions need the implementation of a *callback function*: a so-called ParameterValueListener, which will receive the parameter updates and manage them. The callback function accepts as input

the parameter name for which the value has been published (with the `DEVICE/Property#field` syntax), the value itself, and optionally the the signal's header, containing some additional information (e.g. timestamps).

Another ParameterValueListener, handling the exceptions that may occur during the subscriptions, can be optionally implemented.

```
import pyjapc

def my_callback(name, value, header):  #Create the callback function: in this case
                                           it will just print on the screen what has
                                           been published
  print(f"Got{name} with value {value} at time {header['cycleStamp']}")

pjc = pyjapc.PyJapc("PSB.USER.LHC1A") #Create pyjapc object and initialise it to
                                           take data from the LHC1A cycle in the PSB


pjc.subscribeParam("BR2.BCT-ST/Samples#samples", my_callback, getHeader = True) #
                                           Subscribe to the BCT signal in ring 2

pjc.startSubscriptions() #Start the subscriptions: from this moment the updates will
                                            start to arrive
```

It is also possible to get, set or subscribe to parameters using a different selector from the one passed to the `PyJapc` initialiser. The methods `getParam`, `getNextParamValue`, `setParam` and `subscribeParam` accept an optional parameter, `timingSelectorOverride`, in which a different selector can be specified. This is usually needed when working with cycle-independent signals: being not linked to any PLS user, they need an empty string as a selector.

```
import pyjapc

pjc = pyjapc.PyJapc("PSB.USER.LHC1A") #Create pyjapc object and initialise it to
                                           take data from the LHC1A cycle in the PSB

pjc.getParam("BR.SCOPE33.CH02/Acquisition\#value", timingSelectorOverride="PSB.USER.
                                           NORMHRS") #Get the latest tomoscope
                                           acquisition from the NORMHRS cycle

pjc.getParam("BR.SCOPE33/TriggerCount#value", timingSelectorOverride="") #Get the
                                           number of traces set in the tomoscope (
                                           cycle-independent parameter)
```

3.3 NXCALS

NXCALS (NeXt CERN Accelerator Logging System) logs and stores data coming from the accelerators. It is based on Hadoop Big Data technologies (Hadoop website, 2023) and uses cluster computing power to process and analyse data (NXC, 2023).

NXCALS logs a variety of data from different data sources (called *systems*): the data published by the accelerator devices mainly belong to one of these systems, called CMW (Controls MiddleWare).

In the NXCALS architecture, the data are contained in structures called *entities*, which are, generally speaking, representations of the objects for which state changes over time are stored. In the case of CMW system, an entity represents a `DEVICE/Property` combination. The entities are then organised in *records*, sets of grouped fields, which in the case of CMW correspond to `fields` of that particular device property.

NXCALS architecture

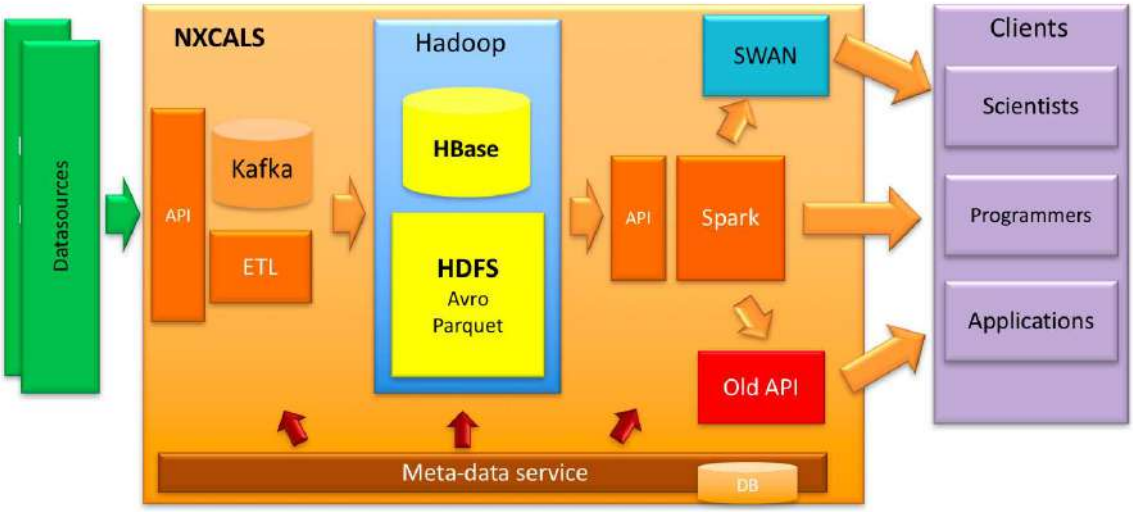


Figure 3.1: NXCALS architecture

NXCALS architecture is quite complex, and the data acquired from the sources go through different stages before being available for extraction.

They are first sent to NXCALS ingestion API, from which they get temporarily stored in Kafka (Kafka website), which acts as intermediate persistent storage buffer for further processing, with a retention time of 2.5 days. In case of processing failures (infrastructure problems, bugs, etc.) this time period allows to fix things and not lose the data.

Then, the data are collected by ETL process and, at the same time:

- they are sent to HBase and nearly immediately made available to users, with a lifetime of 48 hours
- they get compacted, deduplicated and stored in HDFS (Hadoop Distributed File System), process that is usually time consuming

After this stage, the data can be acquired by NXCALS clients through Data Extraction API, which "hides" the origin of the data (HBase or HDFS). The Data Extraction API can be accessed from different interfaces, including a Python API. Combined with Apache Spark (Spark website), it provides means of data extraction and analytics. NXCALS also stores a wide range of meta-data, which are stored in the Oracle database and are available through Meta-data service.

Extraction API

The principal way of accessing logged data is using an *Extraction API*, implemented in JAVA, on top of which a Python library has been built. The Python API is made of a set of python classes that are internally using Py4J, a bridge between Python and Java, enabling Python programs running in a Python interpreter to dynamically access Java objects in a JVM.

NXCALS Data Access API consist of two query builders: *DataQuery* and *ParameterDataQuery*, returning a Spark dataframe as output. It needs specification of time window as input, together with some additional information to identify the data signal, such as the system to get the data from and some key-values pairs, which, in case of CMW system, correspond to device/property specifications.

```
from nxcals.api.extraction.data.builders import DataQuery
from nxcals import spark_session_builder

spark = spark_session_builder.get_or_create(app_name=''spark_session'') # Create
                                            Spark session

start_time = "2022-06-30 09:00:00.000"
end_time = "2022-06-30 10:00:00.000"
spark_df = DataQuery.builder(spark).entities().system("CMW") \
     .keyValuesEq({"device": "BR2.BCT-ST", "property": "Samples"}) \
     .timeWindow(start_time, end_time) \
     .build()  # Query BCT data from ring 2 of the PSB

selected = spark_df.select("samples", "cyclestamp", "selector").where('selector = "
                                            PSB.USER.NORMGPS"') #Select "samples" and
                                            "cyclestamp" field and filter for NORMGPS
                                            selector
pandas_df = selected.toPandas()
```

	samples	cyclestamp	selector
0	{'elements': [0.012314903549849987, 0.01970384...	1656579765665000000	PSB.USER.NORMGPS
1	{'elements': [0.027092786505818367, 0.00369447...	1656579967265000000	PSB.USER.NORMGPS
2	{'elements': [0.0320187471807003, 0.0036944707...	1656579996065000000	PSB.USER.NORMGPS
3	{'elements': [-0.003694470738992095, -0.008620...	1656580111265000000	PSB.USER.NORMGPS
4	{'elements': [-0.0012314902851358056, 0.016009...	1656580168865000000	PSB.USER.NORMGPS
...	...	...	...
120	{'elements': [0.07019494473934174, 0.172408640...	1656582991265000000	PSB.USER.NORMGPS
121	{'elements': [0.14285287261009216, -0.01354639...	1656583020065000000	PSB.USER.NORMGPS
122	{'elements': [0.15024182200431824, 0.124380521...	1656582933665000000	PSB.USER.NORMGPS
123	{'elements': [0.0, 0.05541706085205078, 0.1046...	1656583164065000000	PSB.USER.NORMGPS
124	{'elements': [0.017240863293409348, 0.04556514...	1656583135265000000	PSB.USER.NORMGPS

Figure 3.2: Output of the code for NXCALS data extraction

3.4 Example of some parameters for longitudinal analysis

In order to perform longitudinal analysis, it is necessary to combine and analyse a variety of different signals, depending on the goal one wants to accomplish.

I will present here, as an example, three of the signals I have used the most during my work, that are also among the most used ones in longitudinal analysis routines. They are all relative to the PSB, but in most of the other machines similar devices can be found.

3.4.1 Voltage signals

A range of different signals provide information about the voltage, both programmed and detected, in the RF cavities, along the duration of one cycle. In the PSB, the voltage is given separately for each one of its four rings. Different signals return the total voltage, the voltage of the main harmonic (H1) and the voltage of the secondary harmonic (H2), as a function of the cycle time (C-time), either for specific cavities of the accelerator, or the total one, as a sum of the voltage in each cavity.

Some examples of voltage signals are:

- `BA<ring>.FGVRFH<harmonic>/Settings#amplitudes`, returning the voltage that the user has programmed to be set in the cavities
- `BA<ring>.FGVRFH<harmonic>-SD/Samples#samples`, returning the voltage function that the low-level RF is trying to play in the accelerator (FGVRF stands for Function Generator Voltage RF)
- `BA<ring>.VRFH<harmonic>-SD/Samples#samples`, returning the voltage function that was detected in the cavities.

```
import pyjapc as pj

pjc = pj.PyJapc("PSB.USER.TOF")

total_detected_voltage_R2 = pjc.getParam("BA2.VRF-SD/Samples#samples")
detected_voltage_H1_R2 = pjc.getParam("BA2.VRFH1-SD/Samples#samples")
detected_voltage_H2_R2 = pjc.getParam("BA2.VRFH2-SD/Samples#samples")
```

3.4.2 BCT

BCT stands for Beam Current Transformer, which is the beam instrument measuring the intensity of the beam (i.e. the number of protons) circulating in the accelerator.

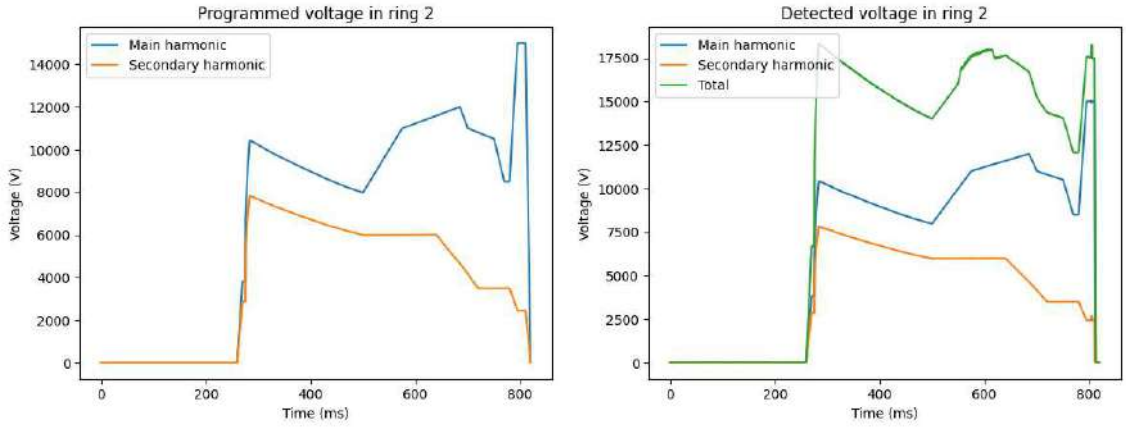


Figure 3.3: Plots of some voltages in the PSB

A Beam Current Transformer works by detecting the magnetic field carried by the beam. In fact, the beam carries an electrical current given by:

$$I_{beam} = \frac{qN_{part}}{t} = \frac{qN_{part}}{l}\beta c, \tag{3.1}$$

which generates, at a distance R from the beam, a magnetic field of intensity

$$B = \mu_0 \frac{I_{beam}}{2\pi R}. \tag{3.2}$$

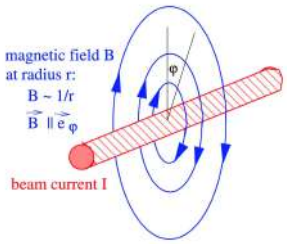


Figure 3.4: The magnetic field generated by the beam (Forck, 2022)

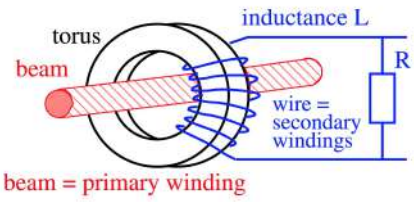


Figure 3.5: Schematisation of a Beam Current Transformer(Forck, 2022)

The beam current can be determined by monitoring the accompanying magnetic field with a current transformer: the beam passes through a highly permeable torus as the 'primary winding'. An insulated wire wound around the torus with N turns serve as the 'secondary winding': measuring the current passing in the secondary winding (or the voltage induced on it), the primary current can be calculated from:

$$I_{sec} = \frac{N_{prim}}{N_{sec}} I_{prim}, \tag{3.3}$$

with $N_{prim} = 1$, due to the single pass of the beam through the torus (Forck, 2022).

After being detected, the signal is converted from current units to number of charges, taking into account the revolution frequency of the beam, and the intensity value can be then returned as a function of the cycle time.

In the PSB, the BCT signal is measured separately for each ring, and it can be accessed through the string `BR<ring>.BCT-ST/Samples#samples`. It is returned as a numpy array, with a unit of 10^{10} particles.

```
import pyjapc as pj

pjc = pj.PyJapc("PSB.USER.TOF")


bct = pjc.getParam("BR2.BCT-ST/Samples#samples")
```

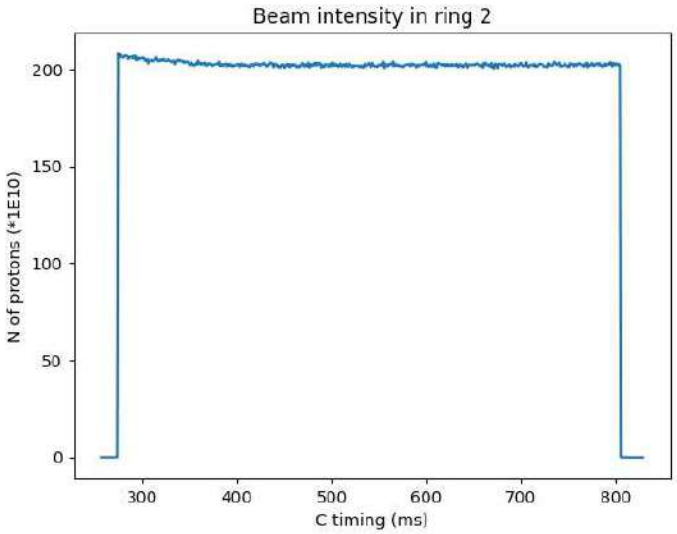


Figure 3.6: BCT in PSB's ring 3, user TOF

3.4.3 Tomoscope

The tomoscope is an application that allows to measure the longitudinal distribution if bunch profiles, i.e. the projection on the x-axis of the longitudinal phase space distribution of the particles, as a function of time (in ns). Starting from a tomoscope signal, and providing some additional information such as voltages and magnetic field, the complete phase space distribution of the particles can be reconstructed, via a routine called *longitudinal tomography* (Hancock and J-L, 2001).

Bunch profiles are measured thanks to a Wall Current Monitor, a device able to return the instantaneous value of the beam current in a specific point of the ring, by measuring the image current that the beam induces on the vacuum pipe.

The signal acquired by the Wall Current Monitor is then digitised thanks to a digitiser triggered by an RF-synchronous clock, and gets then sent to the OASIS stack, FESA class representing a virtual oscilloscope (Fortescue-Beck, 2020b). The tomoscope application, then, subscribes to the correct OASIS channel, and displays the longitudinal beam profile.

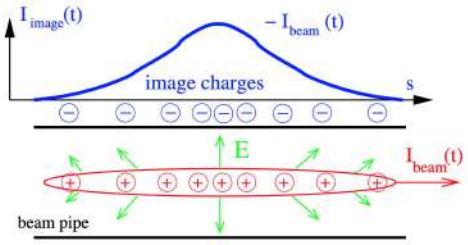


Figure 3.7: The beam current induces a wall current of the same magnitude and time structure but reversed polarity. (Forck, 2022)

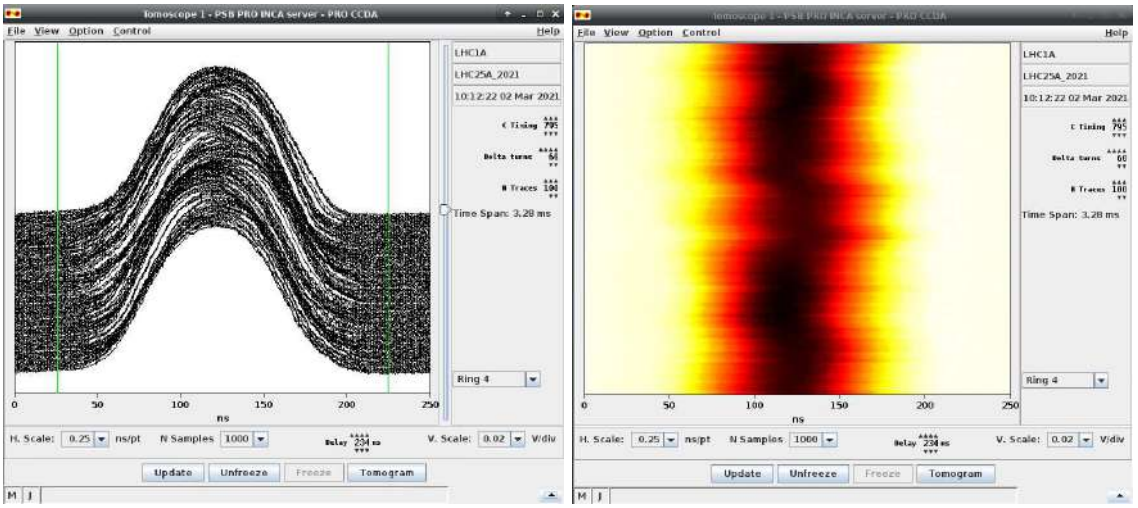
Usually, the tomoscope displays a *waterfall* of profiles, i.e. a succession of profiles of the same bunch, each one separated by a few machine turns. The user can control the vertical and horizontal offsets of the signals, the horizontal scale, the number of traces (i.e. profiles) to acquire, the moment in the cycle in which to start the acquisition, and the separation in time between two traces (in terms of number of turns).

The OASIS FESA class for the tomoscope can be subscribed to: the parameter name is `BR.SCOPE<scope>.CH<channel>/Acquisition#value`, where `<scope>` and `<channel>` are numbers identifying a specific tomoscope. Other properties of the same device allow to access, and change, the tomoscope settings (number of traces, time resolution, offset...)

```
import pyjapc as pj
import numpy as np
import matplotlib.pyplot as plt

pjc = pj.PyJapc("PSB.USER.TOF")

tomo_waterfall = pjc.getParam("BR.SCOPE31.CH01/Acquisition\#value")
```



(a) Mountain Range view *(b) Waterfall view*

Figure 3.8: Screenshots of the tomoscope application displaying 100 profiles of an LHC beam at extraction, from two different views

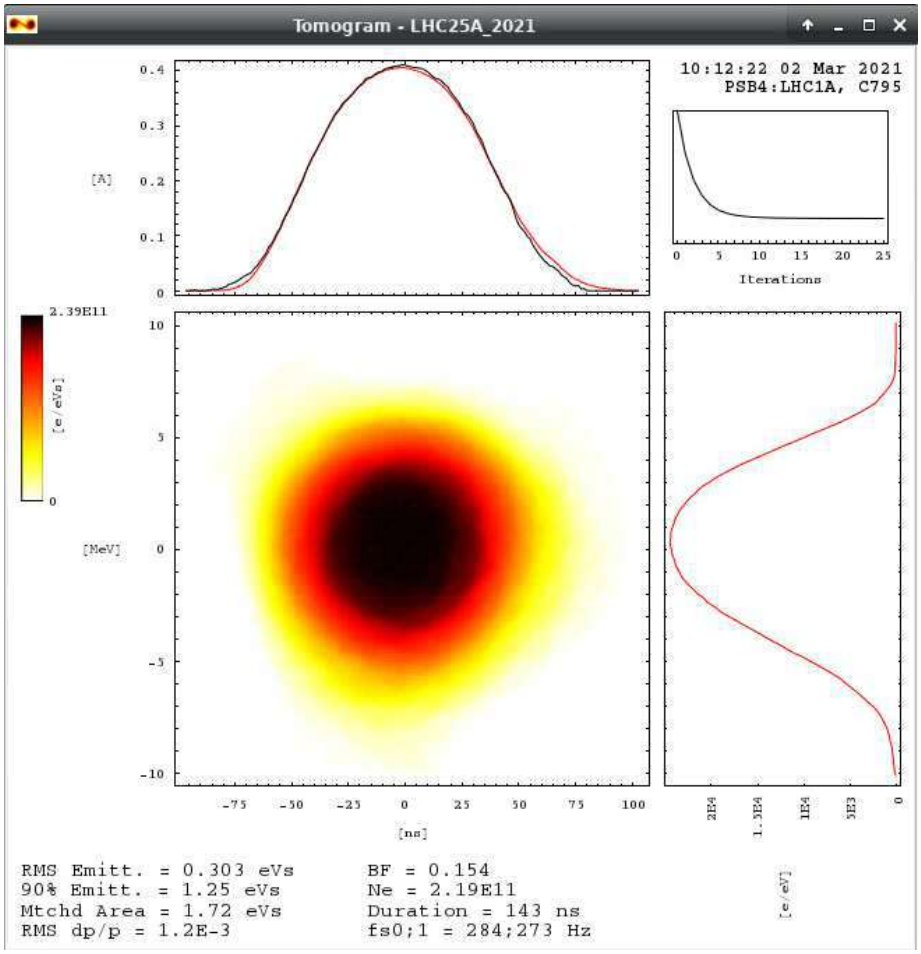


Figure 3.9: Screenshot of the phase space reconstruction of the same beam, made by the tomoscope application

Chapter 4

Data acquisition: the PyDAq library

4.1 Introduction

PyDAq (which stands for Python Data Acquisition) is a library conceived to simplify, unify and standardise the process of acquiring data from the accelerators. Its goal is to provide a simple but flexible interface for data acquisitions and parametric scans, and a well-defined structure for the data saving, which can serve as a starting point for a unified data analysis library. It is built on top of PyJapc and NXCALS, and it aims to unify data acquisition from these two very different sources, providing (as much as possible) the same interface, and offering the possibility to combine data from the two.

4.1.1 Challenges

The majority of data analysis routines are run cycle by cycle: for this reason it is useful to store data in a way that makes it easy to group data relative to the same cycle. In addition to that, in order to be able to run the same analysis on each cycle, it is important to have *all* the necessary data saved for every shot.

Completeness and synchronisation

PyJapc subscriptions return data to the user when the corresponding device publishes them. However, not all the devices publish the data from the same cycle at the same time, some parameters are measured at a single moment of the cycle, and their value is returned straight away, while some other signals need to wait until the end of the cycle before publishing. For this reason, when subscribing to multiple parameters, the signals may not always be returned simultaneously, and the "slower" signals can arrive very close in time with the "faster" ones from the next cycle, making

it difficult to group and save the data for the same shot. Therefore, a "synchronisation"[1] check is needed before saving the data. This can be done by comparing the *cyclestamp* (a timestamp marking the start of each cycle) in each signal's header.

Moreover, for various reasons, it is not guaranteed that all the required signals arrive for each shot: is can happen that one or multiple signals are missing for some cycles. Therefore, completeness has to be checked before saving live data.

In addition to this, not all the signals coming from the accelerator are cycle dependent: settings are usually published (and stored in NXCALS) only when their value is changed in the machines, rather than for every cycle. So, a way to save the current value of settings for every shot, and not just on update, is needed.

Multiple machines

Another feature that PyDAq aims to provide is the possibility to save in a synchronised way (i.e. cycle by cycle) data coming from *multiple accelerators*. This is very important in order to be able to evaluate the impact of operations and optimisations in one machine on the downstream ones, or for tracing back issues and anomalies observed in one machine to the upstream ones. However, despite its importance and usefulness, this has never been implemented in a systematic way.

To assure synchronisation of data across multiple machines, it is necessary to be able to follow the exact same set of particles (i.e.the same *beam instance*) through the various accelerators. However, in this case comparing cyclestamps is not enough, because every accelerator has its own timing system, meaning that the same beam instance has different cyclestamps in different machines. Some other, more general information about the cycles played in the machines is needed, but it has been provided only very recently (during Long Shutdown 2). Now, for the first time, it is possible to implement beam instance tracking through multiple machines.

4.2 Functionality

The main functionality of PyDAq consists of data acquisition, of live and historical data, from a single or multiple machines, and parametric scans. Each of these tasks is implemented in a different class.

4.2.1 Live acquisitions from a single accelerator

The `Logger` class provides the functionality to perform live data acquisitions, implementing synchronisation and completeness checks. It takes care of subscribing to the signals requested by the user, managing the subscriptions, checking synchronisation and completeness of the data that arrive, and saving them in the format specified by the user. Alternatively, instead of being saved, the data can be returned to the user. This can be useful for using them as an input to another application, or for interactive use.

A `Logger` instance has to be initialised by providing a `pyjapc` instance and specifying the machine from which to acquire data, Then, the `add_acquisition` and `add_setting` methods allow specifying

[1] From now on, the word "synchronisation" will be used to refer to grouping data relative to the same shot, both in one and multiple machines.

what signals to acquire. The signals have to be passed together with an *alias name*, which is then used to refer to the signals and to save them.

It is also possible to add *predefined acquisitions*, which automatically add to the `Logger` all the signals needed to run specific routines. For example, by adding the predefined acquisition called `PSBTomoscope`, all the signals needed to perform longitudinal tomography of a PSB beam are automatically added to the `Logger`. These include tomoscope acquisitions and settings, along with information about the measured voltage and magnetic field.

Through the `add_saving_info` method, it is possible to specify the format, path, and name to save the acquired data.

Before starting the acquisitions, the `prepare` method has to be called. After providing a summary of the signals to acquire, it asks the user confirmation to go on with the acquisitions. In case of an affirmative answer, it checks the existence of the required signals and subscribes to them, and then it creates the structure where to later save the data (PyDAq saving structure is explained more in detail in section 4.3).

Then, the `get_n` and `save_n` methods can be called, to respectively have returned, or have saved, a given number of shots. When one of these methods is called, the subscriptions are started, synchronisation and completeness are checked on the data that arrive, and complete and synchronised shots of data are either saved or added to a python dictionary, which is then returned. Optionally, through the `add_validation_function` method, it is possible to provide some additional checks on the data before saving/returning them (for example, a check on the beam intensity to discard empty shots), and through the `add_manipulation` method it is possible to provide some functions to modify the acquired data before saving them.

For each shot, also some metadata are saved, both in the saving structure and in a separate text file, to have a human readable version of them. Cyclestamp and beam destination are automatically saved as metadata, but some more can be added through the `add_metadata` method.

Before starting the acquisitions it is possible to skip a given number of cycles through the `skip_cycles` method.

```
import pyjapc
import pydaq.objects.logger as logger

# Acquisitions specifications
shots_number = 1000
# Creating pyjapc object
pjc = pyjapc.PyJapc("PSB.USER.LHC4")

# Creating logger object
l = logger.Logger(pjc, "PSB", synchronous=True, complete=True)

# Adding to the logger the signals to save
l.add_acquisition("BCT", "BR3.BCT-ST/Samples#samples") #  Intensity in PSB ring 3
l.add_acquisition("VH1", "BA3.VRFH1-SD/Samples#samples")  # Main harmonic voltage in
                                                PSB ring 3
l.add_acquisition("VH2", "BA3.VRFH2-SD/Samples#samples"). # Secondary harmonic
                                               voltage in PSB ring 3

l.add_predefined_acquisition("PSBTomoscope", Ring=3, Scope=34, Channel=1, TNumber=2)
                                                 # Tomoscope predefined acquisition
```

```
# Telling the logger how and where to save the data
l.add_saving_info("npy", "./Results", "Data")

l.prepare()

# Saving the data
l.save_n(shots_number)
```

4.2.2 Historical data extraction from a single accelerator

The `NXCALS_logger` class is dedicated to extraction of data relative to a single machine from the NXCALS database, saving the fetched data shot by shot, or returning them to the user inside a Pandas dataframe.

An instance of this class has to be initialised by providing, besides the machine and PLS user names, the timestamps of the data to extract. There are three ways to do this: providing the start and the end time of the acquisitions ("end" mode), providing the start time and duration of the acquisitions ("dur" mode), or providing a list of single cyclestamps ("lst" mode). The timestamps can be passed both in unix time (in seconds), and as strings, in a human readable ISO format.

After initialisation, the functionality is very similar to the one of the live `Logger`. By calling `add_acquisition` and `add_setting` the required signals are provided, with `add_saving_info` the saving requirements can be specified, then the `prepare` method asks the user confirmation to continue, creates a spark environment, performs checks on the parameters and creates saving folders/files, and finally with the `save_data` and `get_data` methods the data extraction is performed, and the required data are saved or returned.

The data extracted by the `NXCALS_logger` are saved shot by shot, in a tree structure that has the same shape of the one created by the live data `logger`. The only difference is that only the cyclestamp is automatically saved as metadata, and not the beam destination.

The `NXCALS_logger` also gives the possibility to combine data coming from live and historical acquisitions, by saving the data extracted from NXCALS in an already existing saving structure containing data from live acquisitions. In fact, it is very common to use data extracted from NXCALS to integrate the ones previously taken live. This "integration" can be done by initialising the `NXCALS_logger` with the cyclestamps saved in the metadata. Passing to the logger the name of the existing data tree, the `NXCALS_logger` can save the extracted data in the same tree, synchronising the shot numbers with the already existing ones.

```
from pydaq.NXCALS_classes import nxlogger as logger
import pydaq.utils.load_data as ld

# Acquisition specifications
machine = "PSB"
user = "TOF"
mode = "end"  # Providing start time and end time
times = [1653314800.865, 1653315575.065]  # Start and end time of the data we want
                                          save
```

```
# Create logger object
l = logger.NXCALS_logger(mode, times, machine, user)

# Add the signals to save, providing alias name and parameter name as defined in
                                                CCDB
l.add_acquisition("BCT", "BR2.BCT-ST/Samples#samples")
l.add_acquisition("BFIELD", "BR.BMEAS-B-ST/Samples#samples")

# Tell the logger how and where to save the data
l.add_saving_info("npy", "./Results", "nxdata")

# Prepare the logger: create spark environment, perform checks, create saving
                                                folders
l.prepare()

# Query and save the data
l.save_data()
```

4.2.3 Multiple machines acquisitions

As previously mentioned, one of the key features implemented by PyDAq is beam instance tracking, allowing the acquisition of synchronised data from multiple accelerators.

To perform beam instance tracking, it is necessary to link cyclestamps relative to the same beam instance in different machines. This can be achieved by comparing three parameters, provided by the timing system for each beam. They are three different fields of the property identified by the string `<machineletter>X.CZERO-CTML/BeamInfo`, where `<machineletter>` is a letter identifier for a specific machine. Currently, this signal is available for four accelerators: PSB (letter `B`), PS (letter `P`) , SPS (letter `S`), LEIR (letter `E`). In particular:

- The field `bcdPlayStampMs` is the timestamp marking the start of a supercycle
- The field `bcdBeamOffsetBp` describes the time delay of a cycle from the start of the supercycle
- The field `beamId` is a code identifying a specific combinations of selectors

One beam instance, and only it, will have the same value of these three parameters in all the machines. Cyclestamps linked to the same values for these three parameters are relative to the same beam instance.

The `logger_multiple_machines` class makes use of these fields to implement live beam instance tracking. By checking their value for every complete shot of data coming from one accelerator, it is able to save under the same shot number all the data relative to one beam instance.

It has a very similar functionality to the simple `logger` (functions `add_acquisition` , `add_setting`, `prepare`, `get_n`, `save_n`), with the difference that in this case the machine and the user to get the data from are not passed to the class initialiser, but they are parameters of the `add_acquisition` and `add_setting` methods.

Besides the PyJapc instance used to get the data, the `logger_multiple_machines`'s initialiser also accepts a flag, called `complete_multiple`: if it is False, it saves the data relative to every beam circulating in the machines with the passed selector; if it's True, it saves data only for the beams that are programmed to go through *all* the selected machines.

```
import pyjapc
import pydaq.objects.logger_multiple_machines as logger

# Acquisition specifications
shots_number = 3
formt = "HDF5"

# Create pyjapc object
p = pyjapc.PyJapc()

# Create logger object
l = logger.Logger(p, synchronous=True, complete_single=True, complete_multiple=True)

# Add to the logger the signals to save, providing alias name, parameter name as
                                        defined in CCDB, machine and selector
l.add_acquisition("BCT_RING1", "BR1.BCT-ST/Samples#samples",
                  "PSB", "PSB.USER.MD2")
l.add_acquisition("BCT_RING2", "BR2.BCT-ST/Samples#samples",
                  "PSB", "PSB.USER.MD2")
l.add_acquisition("VH1_RING1", "BA1.VRFH1-SD/Samples#samples",
                  "PSB", "PSB.USER.MD2")
l.add_acquisition("BCT", "PR.BCT-ST/Samples#samples",
                  "PS", "CPS.USER.SFTPRO3")
l.add_acquisition("BCT", "SR.BCT.31458-ST/Samples#samples",
                  "SPS", "SPS.USER.SFTPRO1")

# Tell the logger how and where to save the data
l.add_saving_info(formt, "./Results", "Data")

# Prepare the logger: subscribe to signals, perform checks, create saving folders
l.prepare()

# Save the data
l.save_n(shots_number)
```

Beam instance tracking can be performed also with NXCALS data. Although the necessary functionality exists, it has not yet been organised into a Python class. However, there are plans to create a new logger that can perform multi-machine acquisitions from NXCALS, with similar features to the existing loggers.

4.2.4 Parametric scans

A parametric scan consists in setting different values for one or more parameters in the machines, and observing the behaviour of the beam for each set value. PyDAq aims to simplify this process, and make it more automatic without losing control on the values set in the accelerator devices.

PyDAq's `Scanner` class, used in combination with the `Logger`, allows parametric scans with any number of dimensions (i.e. any number of parameters), with the possibility of scanning the parameters in a grid (values evenly spaced between two extremities), or randomly sampling a new value at each iteration.

A `Scanner` instance has to be initialised with the PyJapc instance that will be used to set the accelerator values, and with a flag indicating whether to choose the values in a grid or through random

sampling. If random sampling is chosen, also the number of values to be set is required.

Once the `Scanner` object is created, the parameters to be scanned can be added through the `add_param` method. The method requires passing an alias name, the parameter name with the `DEVICE/Property#field` syntax, and some other information specifying how to scan that parameter. If random sampling is selected, it is enough to pass the limits of the range in which the parameter values will be sampled. If grid scanning is selected, the user can provide the limits and the number of values to be scanned for that particular parameter, or alternatively they can provide a list of custom values to be set. The `Scanner` always computes single values, but some devices need an array as input in the `set_value` function. In this case, it is possible to provide a function to transform the single value computed by the `Scanner` into the array needed by the device.

After adding all the parameters, a `prepare` function has to be called. It checks that all the added settings exist and can be modified and, in case of grid scanning, computes the grid size and the values to be set.

At this point, through the `next_values` property, it is possible to access the values that the `Scanner` is going to set in the machines at the next iteration. The user can create some sanity checks on these values, to prevent potentially harmful settings from being sent to the hardware, and then use the methods `set_value` and `skip_value`, to respectively set the next value in the machines or skip it, before updating the `next_values` property.

It is also possible to access the last and the current values set (through the `last_values` and `current_values` properties), the number of values already set and the ones still to be set (through the `n_sampled` and `to_sample` attributes) and, in case of grid scanning, the current location in the n-dimensional parameter grid. Finally, the `skip_n` method allows skipping a given number of iterations, useful to resume a scan that had been interrupted.

To acquire beam data after every value is set, a `Logger` object can be used. It can be coupled with the `Scanner` object through the `add_scanner` method: in this case, it will save as metadata the values set by the `Scanner`.

```
import numpy as np
import pyjapc
import pydaq.objects.logger as logger
import pydaq.objects.scan as scan


# Function to manipulate the voltage
def voltage_function(input_voltage: float, initial_voltage: np.ndarray):
    peak_id = 10
    inj_ids = range(peak_id+1)
    steps = np.array([0.846, 0.8667, 0.9375, 0.96])

    voltage_func = initial_voltage
    inj_voltage = voltage_func[0][1][inj_ids]

    inj_voltage[2:5] = input_voltage * 0.50
    inj_voltage[5] = input_voltage * 0.45
    inj_voltage[6:peak_id] = input_voltage * steps
    inj_voltage[-1] = input_voltage

    voltage_func[0][1][inj_ids] = inj_voltage
```

```
    return voltage_func


# Acquisition specifications
n_shots = 3

# Create pyjapc object
p = pyjapc.PyJapc("PSB.USER.TOF")

# Get value needed for scan
voltage0 = p.getParam("BA2.FGVRFH1/Settings#amplitudes")

# Create scanner object
scanner = scan.Scanner(True, p)

# Add parameter to scan, providing range limits and the number of values
scanner.add_param("VH1", "BA2.FGVRFH1/Settings#amplitudes",limits= [10E3, 20E3],
                                        nSamples=5,  func=voltage_function,
                                        initial_voltage=voltage0)
# Add parameter to scan, providing custom values
scanner.add_param("VH2", "BA2.FGVRFH2/Settings#amplitudes", custom_values=[10E3,
                                        15E3, 18E3], func=voltage_function,
                                        initial_voltage=voltage0)


# Prepare the scanner: compute scan size, compute values to scan
scanner.prepare()

# Create logger object
l = logger.Logger(p, "PSB", synchronous=True, complete=True)

# Add to the logger the signals to save, providing alias name and parameter name
l.add_acquisition("BCT", "BR2.BCT-ST/Samples#samples")
l.add_acquisition("VH1_detected", "BA2.VRFH1-SD/Samples#samples")
l.add_acquisition("VH2_detected", "BA2.VRFH2-SD/Samples#samples")

# Add the scanner object to the logger
l.add_scanner(scanner)

# Tell the logger how and where to save the data
l.add_saving_info("npy", ".", "Results")

# Prepare the logger: subscribe to signals, perform checks, create saving folders
l.prepare()

# Do the scan, with sanity check
for i in range(scanner.nValues):
    if scanner.next_values["VH1"] < 14E3 or scanner.next_values["VH1"] > 16E3:
        scanner.set_value()
        l.skip_cycles(2)
        l.save_n(n_shots)
    else:
        scanner.skip_value()
```

4.2.5 Acquisitions from input file

All the acquisitions can be also run without creating a `Logger` object. The requirements can be written in an input textfile and then the acquisitions can be started simply by executing an acquisition function, either within a Python script or from the command line.

Currently, JSON is the only format supported for the input file, but more will be added in the future. The JSON input file must include a first dictionary containing general requirements (e.g. saving info, shots number for PyJapc acquisitions, time specifications for NXCALS acquisitions), followed by other dictionaries, one for each machine to get the data from, specifying what signals to acquire. Some examples of input files can be found in the appendix A.

4.3 Saving structure

The data are saved in a well-defined structure, which is preserved across different formats. Currently, two different formats are available: "npy" and "HDF5" , and more formats will be added in the future.

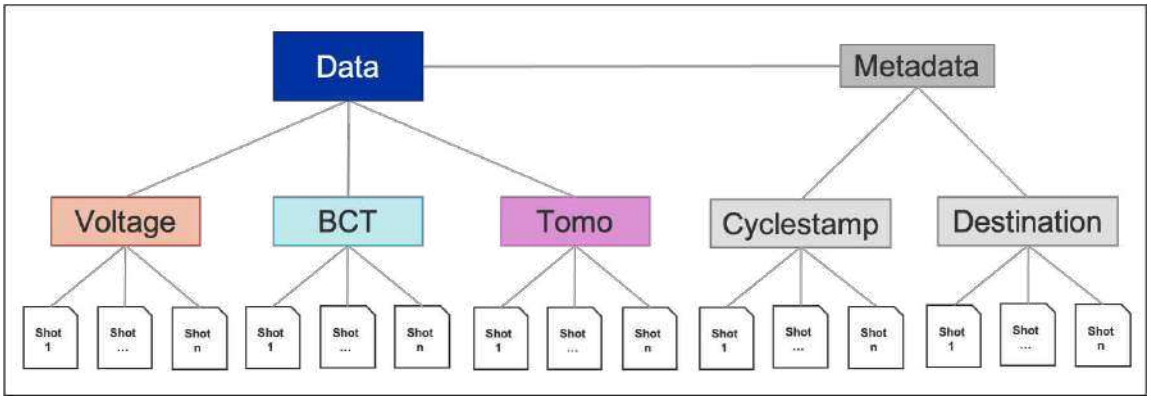


Figure 4.1: Saving structure for single machine acquisitions

In "npy" format, the data are stored in `.npy` files, and the name of each file includes the shot number from which the data were taken. The files are contained in directories named after the alias name of the signals they come from, and these directories are contained in a main directory, whose name corresponds to the one specified by the user when providing the saving information. Inside the main directory, also the `METADATA` directory can be found.

In "HDF5" format, the saving structure is the same, but the containers change: the main directory becomes a `.hdf5` file, the subdirectories become HDF5 groups, and the data are stored inside HDF5 datasets (see fig. 4.1).

In the case of saving data from multiple machines, the saving structure is almost the same: only one more layer is added, specifying the machine and the PLS user of the data. (see fig. 4.2).

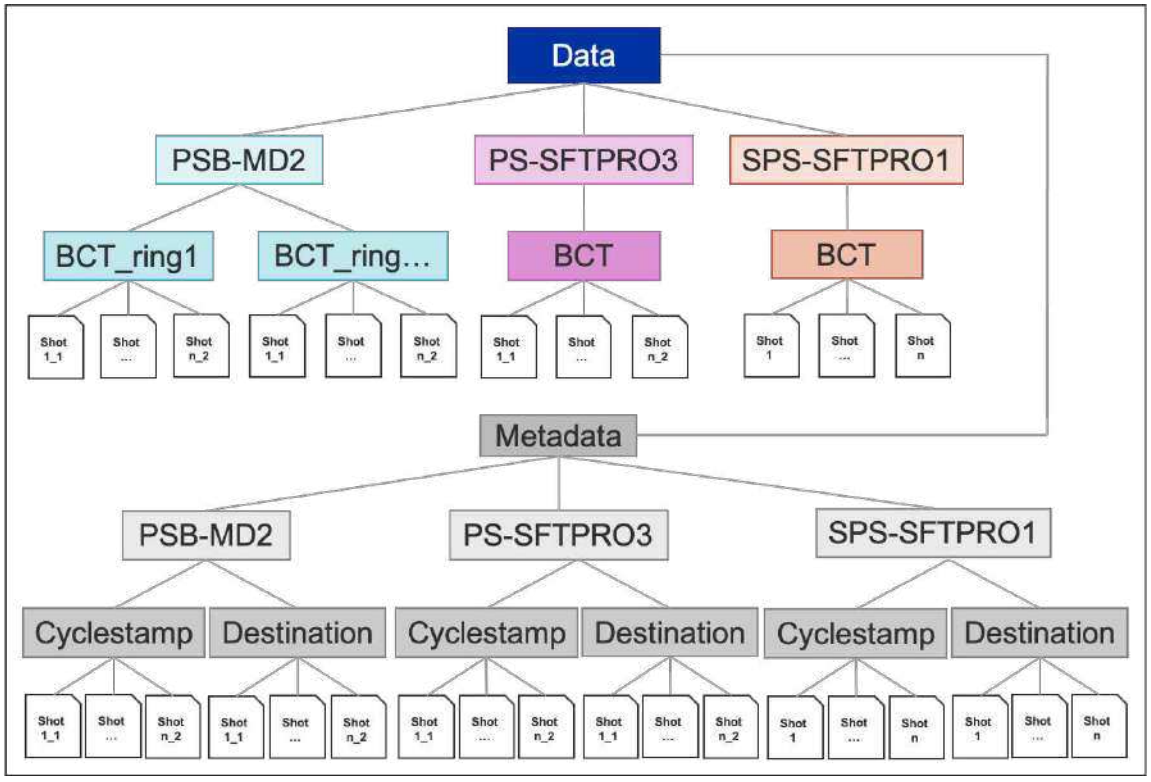


Figure 4.2: Saving structure for multiple machines acquisitions

4.3.1 Logging messages

PyDAq's loggers store records of all their operations in a textfile, by sending logging messages. This is very useful for diagnosing problems during acquisitions. Logging messages are automatically enabled when creating any `Logger` object, while for other uses of PyDAq they have to be manually enabled, through the dedicated module (`pydaq.utils.logging_management`) or through the methods `enable_file_logging` and `disable_file_logging`, present in every PyDAq's class.

4.4 Code architecture for live acquisitions

PyDAq is based on a collection of classes, each one with specific functionality and tasks. The architecture for live data acquisitions is quite complex, with several classes working together and nested within each other.

In the following sections, a description of the classes involved in PyDAq's live acquisition process will be given.

4.4.1 Parameter and device dataclasses

The most elementary class is the `parameter` dataclass. Every `parameter` instance represents a specific property of an accelerator device, and the `parameter` dataclass's fields are named after the fields that the real device's property has. It is in each of these fields that the values coming from the subscriptions to the accelerator devices get stored. Every `parameter` instance inherits from an abstract dataclass called `parameter_base`.

The `parameter` objects are contained in the fields of `device` dataclasses, which represent accelerator devices. Also `device` instances inherit from an abstract dataclass, called `device_base`: its main method, called `initialize`, subscribes to the signals corresponding to the properties that the `device` instance contains in its fields (but without starting the subscriptions). For every subscription, the correct callback function is provided, storing the published values in the respective `parameter` fields. The `device_base` contains also some methods to set and get single values from the devices.

Specific `device` instances can be automatically generated using PyDAq's `device_generation` module. The `generate_device` function returns a string with the lines of code needed to create a particular `device` object; this string can be then saved in a .py file through the `save_device` function, becoming a module that can be imported. The `generate_device` function requires a list of device properties for that particular device: this list can be created manually or looked up in pyccda[2] using the `find_properties_and_fields` function, also contained in the same module.

4.4.2 Acquisition control

The `device` objects can be contained in an `acquisition_control` class, responsible for managing subscriptions to signals and validating the latest acquisition. Through the `add_device` method, `device` objects can be added to the `devices` data member of an `acquisition_control` instance. When a `device` is added to an `acquisition_control` object, the `initialize` method gets called and the subscriptions to its properties are made. At this point, through the `acquisition_control`'s methods `start_subscriptions` and `stop_subscriptions`, the subscriptions can be started and stopped.

In addition to managing subscriptions, the `acquisition_control` class can also include some validation functions that perform checks on the latest data and return a boolean value. Through the `valid` property, the validity of the currently stored values can be returned (`True` is returned if all the validation functions return `True`, `False` is returned otherwise).

The `acquisition_control` class contains also some methods to get the latest values that have arrived (the ones that are currently stored inside the `device` objects), or to get a specific value.

4.4.3 Single machine logger

The `acquisition_control`'s functionality is then used by the `Logger` class: every `Logger` instance contains an `acquisition_control` object as a data member, which gets created when calling the `Logger`'s `prepare` method.

[2]pyccda is the Python Controls Configuration Api, a python interface to the CCDB (Controls Configuration Data Base), which stores the controls information (Fortescue-Beck, 2020c) (Urbaniec and Burdzanowski, 2022).

When calling `get_n` or `save_n` in the `Logger`, the subscriptions to the signals are started from inside the `acquisition_control`, and the data get saved (or returned) only after being validated by it. Completeness and synchronisation of the data are checked through some validation functions that the `Logger`'s `prepare` method adds to the `acquisition_control`.

4.4.4 Multiple machines logger

Finally, the multi-machine `Logger` contains one simple `Logger` instance for every machine involved in the acquisitions. Each simple `Logger` works separately to get and validate data from their own machines and, when the data in one logger are valid, the multiple machines `Logger` saves them with the correct shot number.

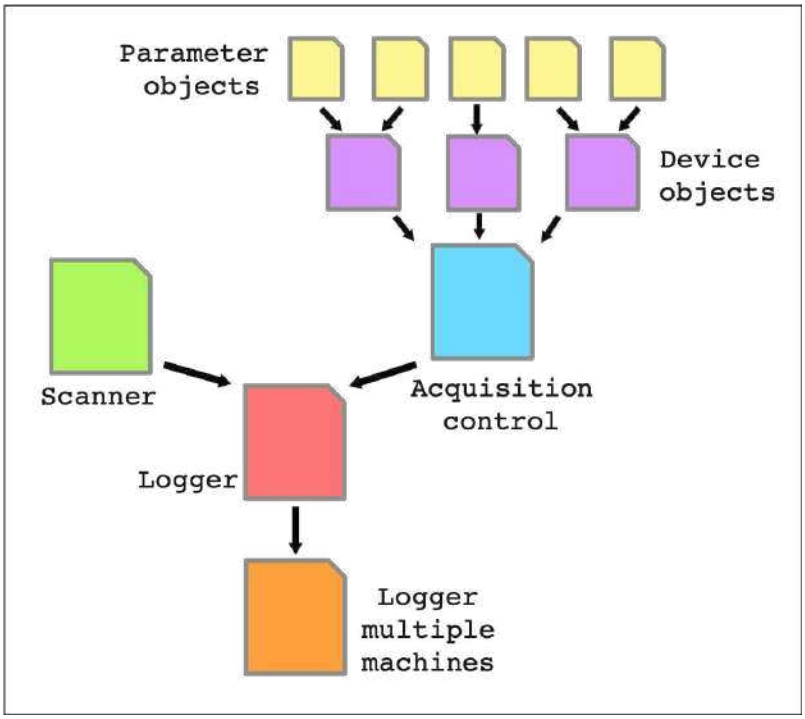


Figure 4.3: Scheme of how the different classes interact. Every arrow means "is contained in" or "can be contained in"

A scheme of how the different classes work together is shown in fig. 4.3.

4.4.5 Example of a multiple machine acquisition

To put ideas together, in the following paragraphs I will try to explain what actions are taken during a multiple machines acquisition.

When a `logger_multiple_machines` instance is initialised, a private data member called `_loggers` is created: it will contain a list of single machine `Loggers`, one for every accelerator involved in the

data acquisitions.

As signals from new machines are added (through the `add_acquisition` and `add_setting` methods), a simple `Logger` for the specified machine gets created, initialised and added to the `_loggers` parameter of the multiple machines logger. The signal information gets added to the simple `Logger`, inside its `acquisitions` or the `settings` data members. During the initialisation of the simple `Logger`, an `acquisition_control` instance is created and assigned to the `Logger`'s `_acquirer` data member.

When calling the `prepare` method on the multi-machine logger, `prepare` is called on each simple `Logger` contained in the multi-machine one. At this point, every `Logger` creates a `device` instance for every signal, and adds it to its `acquisition_control` object through the `add_device` method. Every `device` instance, when added to the `acquisition_control` object, creates the subscriptions to the corresponding signals, with the correct callback function, which will fill the fields of the `parameter` objects contained in the `device` instances. It is also during this phase that the validation functions to check completeness and synchronisation are added to the `acquisition_control` objects.

When calling the methods `get_n` or `save_n`, every `Logger` starts the subscriptions through its `acquisition_control`'s `start_subscriptions` method. The signal values start to be published and stored in the corresponding `parameter` instances. When one `acquisition_control` object validates its data (meaning that they are complete and synchronised), the multi-machine logger saves them with the correct shot number.

4.5 Practical uses

PyDA has already been used for some projects. Here, I will describe some of them.

Machine learning agent training

PyDAq's parametric scan functionality has been used to make convexity checks and train a machine learning agent. Different values of voltage for the main and secondary harmonics in the PSB, at flat bottom[3], have been scanned, and the saved data have been used to generate heatmaps of emittance and transmission of the beam after injection (see fig. 4.4).

The obtained results have been used to shape the cost and reward function of a machine learning agent aimed to optimise the injection parameters in the booster.

Loss of Landau damping studies

Landau damping mitigates coherent beam instabilities, and it is generated by the synchrotron frequency spread caused by the nonlinearities of the RF fields. Under some conditions, a loss of Landau damping is observed, with the instabilities not being mitigated anymore (Karpov et al., 2021).

The phenomenon is being studied within the BR section, in order to create a model describing it: to do so, measurements from the accelerators are compared with numerical simulations. At the

[3]Flat bottom is the very first part of the cycle, straight after injection, before the acceleration starts, when the magnetic field is still flat.

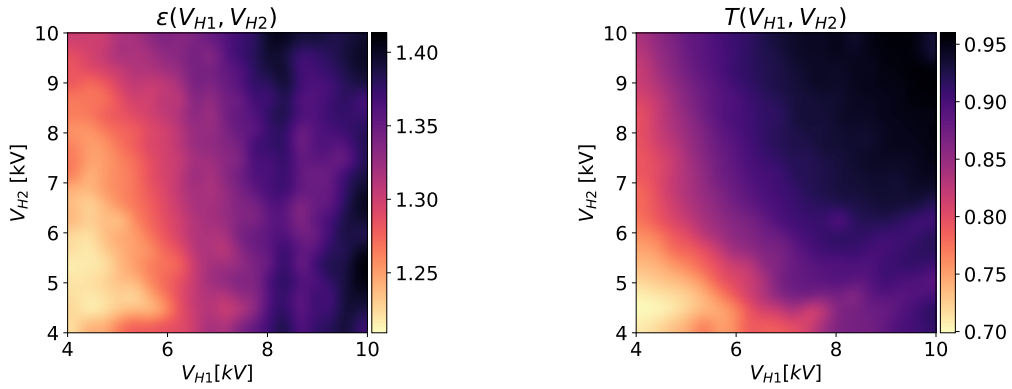


Figure 4.4: Emittance and transmission at flat bottom for different voltage values, generated with data taken through PyDAq.

moment, loss of Landau damping is studied in the PS and SPS for double-harmonic systems: after injection, the beam gets excited by a phase kick, and the resulting bunch oscillations are observed. If Landau damping is present, the oscillations are damped. This procedure has been repeated for different intensity values, and part of the data have been saved using PyDAq. An example of the results obtained is shown in fig. 4.5.

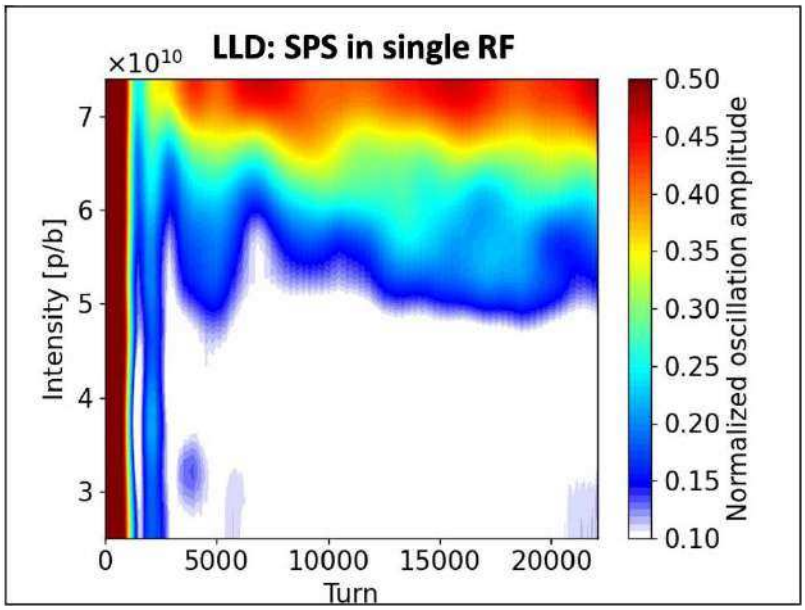


Figure 4.5: Hilbert transform of the time evolution of the bunch offset amplitude after a kick, for different intensity values, in the SPS. It is possible to observe that above a certain intensity Landau damping is lost.

Display LEIR LLRF diagnostic information

PyDAq's `acquisition_control` class can be used also outside the `Logger`: for example, it has been used to handle the subscriptions to a group of signals coming from LEIR's low-level RF (LLRF) system, in order to display their current values on a diagnostic panel. In order to do this, 5 `device` objects have been created (though PyDAq's `device_generation` module) and added to an `acquisition_control` instance, which manages the subscriptions and returns the latest values of the signals, allowing them to be displayed on the panel. A screenshot ot the panel is reported in fig. 4.6.

Figure 4.6: A screenshot of the LEIR LLRF diagnostic panel, fed by information coming from PyDAq.

Voltage calibrations in the PSB

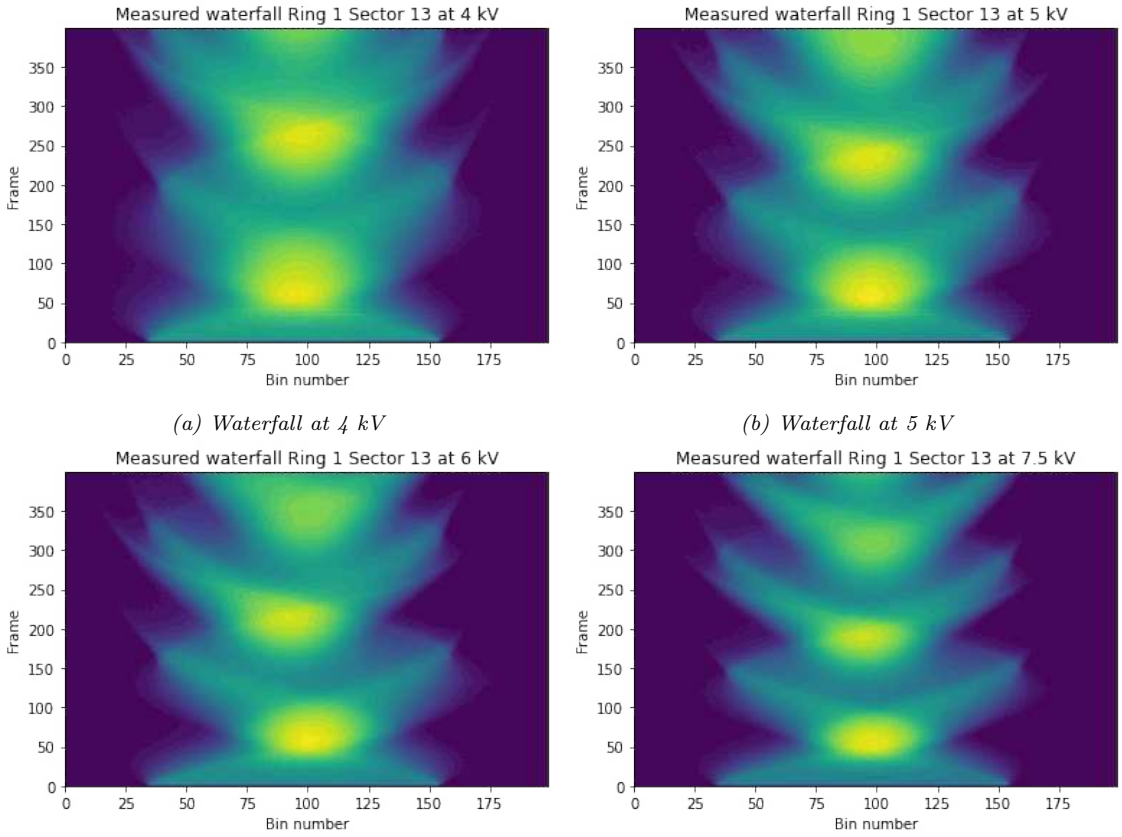
PyDAq's `logger` is being used to perform voltage calibrations in the PSB, which consist of a comparison between the voltage set in the booster's RF cavities and the one experienced by the beam (obtained by phase space reconstruction, done with tomography), in order to calibrate the voltage set in the accelerator(Quartullo et al., 2022).

The PSB voltage calibrations performed in 2023 used PyDAq for data acquisitions. In particular, the `logger`'s `PSBTomoscope` predefined acquisition were particularly useful, allowing to automatically save all the data needed to reconstruct the phase space distribution through tomography. Some of the acquired waterfalls are reported in fig. 4.7.

4.6 Performance and limits

Performance tests are currently ongoing to understand the limits of PyDAq, especially regarding the part of live data acquisitions: since everything happens in real time, speed plays an important role.

In particular, the major element of concern is the time taken by PyDAq to save the acquired data once they get validated. To ensure completeness and synchronisation, it is essential that all the data relative to one shot are saved before new data, from the next shot, arrive. The success of this



(a) Waterfall at 4 kV

(b) Waterfall at 5 kV

(c) Waterfall at 6 kV

(d) Waterfall at 7.5 kV

Figure 4.7: Tomoscope waterfalls acquired with PyDAq's predefined acquisitions for PSB voltage calibrations, from ring 1, sector 13.

process depends on many factors: the time interval between two cycles with the same PLS user, the saving format and the number of signals to be saved.

Profiling

The amount of time taken by PyDAq to save the acquired data, after they get validated, has been measured. The tests have been performed under different conditions: different kinds of beams, different saving formats, different signals to be acquired.

For the first test 10 shots of two BCT signals and one voltage signal have been acquired from different PSB beams, resulting in 30 arrays to be saved for each test. Four saving formats have been compared: "npy", "HDF5", and "HDF5" with `gzip` data compression, for two different compression levels (level 4 and level 9). From the tests it emerges that the "npy" format is the fastest, taking on average 20% less time to save the 30 data points, with respect to the "HDF5" format (uncompressed), and it also takes less storage space than the uncompressed HDF5 file. With compression, it is

	Average saving time (s)	Average storage space (kb)
npy	0.153	264
HDF5 (uncompresssed)	0.194	328
HDF5, compression level 4	0.287	225
HDF5, compression level 9	0.277	224.8

Table 4.1: Results from the first profiling test: average time taken to save 30 data points, and the storage space occupied by them, in different formats

	Average saving time (s)	Average storage space (Mb)
npy	0.635	6.650
HDF5 (uncompresssed)	3.198	6.850
HDF5, compression level 4	3.777	3.900
HDF5, compression level 9	4.371	3.875

Table 4.2: Results from the second profiling test: average time taken to save 180 data points, and the storage space occupied by them, in different formats

possible to reduce the file size of about 30%, but at the price of a slower saving. The results are shown in table 4.1.

For the second test, the `PSBTomoscope` predefined acquisition has been included: in this way, a greater amount of signals were taken (18, instead of just 3), including also a heavier signal, the tomoscope waterfall, consisting of an array of 400'000 ints. In this case, the differences in speed between "npy" and "HDF5" were much more evident. The results are shown in the table 4.2.

Large number of signals

The second test was aimed at understanding how many signals could be saved with PyDAq before data from a new shot start arriving.

In this case only sampler signals (consisting of arrays of floats, with a length in the order of 10^3) were used, and the evaluation was made by looking at the logging messages saved during the acquisitions. Two different saving formats (npy and HDF5 uncompressed) were compared, for two different PLS user: the first PLS user was present only once in the supercycle; the other user is LHC1A which was present four times in the supercycle, with only two other cycles between them. This means that from the start of one LHC1A cycle to the start of the following one only 3.6 seconds would pass.

The results showed that in the case of only one cycle in the supercycle, problems started arising when taking 100 signals for the HDF5 format, and 180 signals for the npy format. For the LHC1A case, instead, some acquisitions started failing with 70 signals for HDF5, and 100 signals for npy.

It has been also observed that the behaviour was not always the same when repeating the same acquisitions, so more tests have to be done in order to have a larger set of data. Moreover, it has been observed that in the majority of the tests that were failing, only the acquisition of the first shot was showing issues. Also this observation will need further investigation.

In any case, it is already quite clear that the saving format plays an important role in live data acquisition, and the different performances of different formats have to be taken into account when starting a live acquisition with PyDAq.

4.7 Possible improvements and future work

As mentioned before, not all the features of PyDAq have been completely developed and released yet, with the NXCALS multi-machine logger being still in the creation phase. Therefore, the first priority on the to-do list is to complete its development and include it in the next release of the library.

Additionally, a lot of small improvements and fixes are planned to be implemented, along with some additional features, such as time-based, and not shot number-based, live acquisitions specifications (e.g. save one hour of data, or save data once every hour), or enabling custom scanning interacting with an external object.

Moreover, new formats are planned be supported, both for the input file (e.g. yaml) and for the output saving structure (e.g. parquet), and new predefined acquisitions will be added.

Finally, the aim is to extend multiple machine acquisition and beam tracking to every accelerator in the CERN complex (currently only four machines are involved, PSB, PS, SPS and LEIR). However, this will be possible only if the timing information about the cycle and supercycle gets published also for the other accelerators. In particular, it will be very useful to include the LHC in the beam instance tracking: this will allow having a way to evaluate the effectiveness of beam operations in the upstream accelerators by checking the beam quality in the LHC.

Chapter 5

Data analysis: the PyLAn library

5.1 Introduction

PyLAn stands for Python Longitudinal Analysis, and it's a library created to unify the existing longitudinal analysis libraries and functions into a single, common and collaborative library for longitudinal beam dynamics routines and functions.

Being a collaborative library, users will be able to add their own functions, according to their particular needs and purposes, and the added functions can be also made public, so that other people with similar needs can benefit from them. This can be very useful, especially for some analysis routines that are quite standard and represent a starting point for many different studies (bunch profile analysis, longitudinal tomography, bunch fitting...).

Of course, the collaborative nature of the library carries the risk of an uncontrolled growth of the library, with the presence of duplicates of functions, that may perform similar analyses in different ways.

My goal is to build in PyLAn a framework of functions and classes providing:

- The possibility to chain functions one after the other, and to run the same sequence of functions on many shots of data, in order to simplify and automate the data analysis process
- A system to compare similar functions (or sequences of functions), in terms of speed and output, in order to help prevent the uncontrolled growth previously mentioned

PyLAn is conceived to be a continuation of the PyDAq library: it can accept any data as input, but it works most effectively with data saved using PyDAq.

The library is still in its early stages of development, with only a first part of the planned functionality currently available. A test version is available for a small group of beta testers.

5.2 Pipeline and Data Manager

A typical data analysis routine involves running a sequence of functions one after the other, each function performing a specific analysis task and returning a value needed as input for the next function. Additionally, the same sequence of functions is often run on multiple shots of data, either data with the same initial parameters, in which the final results get usually averaged, or data with different initial parameters, for example coming from parametric scans, in which the final results get compared.

In order to simplify and automate this process, PyLAn offers two classes: the `Pipeline` class, to chain functions one after the other, and the `DataManager` class, to run function sequences on large sets of data.

5.2.1 The Pipeline class

PyLAn's `Pipeline` class allows users to chain functions together and run them in a specific order.

The functions can be added to the `Pipeline` through the `add_function` method, by providing the function to add, a list of alias names for the parameters that the function requires as input, and a list of alias names for the values that the function returns as output. The order of these aliases must match the order of the function's arguments and return values. Optionally, a dictionary containing kwargs can also be provided.

Each function added to the `Pipeline` is stored as a `Function` dataclass, containing the information about the function's input, output, and kwargs. These `Function` dataclasses are then added to the `Pipeline`'s `functions` data member in the order in which they were added, which is the same order in which the `Pipeline` will run them.

The `Pipeline` object is callable: it can be executed by passing a dictionary as an input. The keys of the dictionary need to match the input aliases of the functions in the `Pipeline`, and the dictionary values have to contain the input data. When the `Pipeline` is called, every function it contains is executed, with the input values for each function coming from the input dictionary. The output of each function is then added to the dictionary, using the alias names stored in the corresponding `Function` object. In this way, the input dictionary for the following function in the pipeline contains both the original input data and the data returned by the previous functions.

The output of the `__call__` method is a dictionary containing both the input and the output values of all the functions in the pipeline, with the keys of the dictionary corresponding to the alias names provided during the function addition. There is also the option to exclude certain function outputs from the final output dictionary. This can be done by passing an additional argument, `transient_outputs`, to the `add_function` method, specifying the alias names of the unwanted outputs. In this way, the transient outputs are only used as input for the following functions in the pipeline, but they are not included in the `Pipeline`'s output dictionary.

The `Pipeline` contains also two alternative methods to run the chain of functions on a dictionary, catching potential exceptions raised during the run of the functions. This prevents the whole pipeline run to be blocked in case of an exception. The exceptions raised during the run get printed to screen or returned in a dictionary.

Besides being callable, the `Pipeline` class is also iterable, addable, indexable and appendable.

5.2.2 The Data Manager class

The `DataManager` class provides an interface for handling and organising data, including loading and saving them, and to run `Pipeline`s over them.

The data are stored in a pandas dataframe within the `data` data member, with values from the same cycle stored in the same row.

The `load_from_pydaq` method loads data that were saved using the PyDAq library (or, anyway, that are saved in the PyDAq's saving structure): in this case, the column names of the `data` dataframe correspond to the alias names that were used when saving the signals.

The `DataManager` class also provides methods to add or remove single rows or columns from the `data` dataframe.

The main feature of the `DataManager` class is the `run_pipeline` method, which runs a `Pipeline` on the data stored in the `DataManager` instance. To do so, the column names in the `data` dataframe must match the alias names required by the `Pipeline`'s functions. The provided `Pipeline` is run on each row of the `data` dataframe, and the output values are added to the `data` dataframe in new columns.

After running the `Pipeline`, the new dataframe with the added columns can be then saved to disk through the `save_data` method: the PyDAq saving structure is used, with the same available formats.

If, during the `Pipeline` run, some exceptions are raised, they are caught and stored in the `raises` data member. This prevents a single exception from blocking the whole `Pipeline` run: the shots not affected by the exception are still giving a result. For the shot (or shots) affected by an exception, a NaN value is added to the `data` dataframe. Optionally, the user can indicate another value to be returned in case of an exception, passing it as a parameter to the `run_pipeline` method.

```
import numpy as np
from pylan.classes import data_manager as dm, pipeline
import myfunclib  # Library containing the analysis functions that will be used in
                                          the pipeline


datamanager = dm.DataManager()  # Creating DataManager instance

input_data = "../input_data"
# Loading intensity data and bunch profiles
datamanager.load_from_pydaq("HDF5", input_data, "pydaq_data",
                            alias=["BCT", "Tomo_Data", "Tomo_HScale", "Tomo_NSamples
                                                                      "])

pipe = pipeline.Pipeline()  # Creating Pipeline instance

# Adding function returning the integral of the bunch profiles
pipe.add_function(np.trapz,
                  input_aliases=["Tomo_Data"],
                  output_aliases=["integral"],
                  transient_outputs=["integral"])
```

```
# Adding function that normalises the bunch profiles
pipe.add_function(myfunclib.normalise,
                  input_aliases=["Tomo_Data", "integral"],
                  output_aliases=["normalised"],
                  transient_outputs=["normalised"])

# Adding function that multiplies the normalised bunch profiles with the beam
                                        intensity at injection
pipe.add_function(myfunclib.bunch_in_npart,
                  input_aliases=["normalised", "BCT"],
                  output_aliases=["final"])

res = datamanager.run_pipeline(pipe)

datamanager.save_data("npy", ".", "analysis_results")
```

5.3 Pipeline evaluation

In order to prevent adding functions and pipelines with the same functionality to the PyLAn library, similar pipelines can be compared and evaluated, in terms of speed and result. In this way, any newly added routine can be compared to similar ones already present in PyLAn.

PyLAn offers two ways to compare `Pipeline`s: a simple function for basic comparisons, and a more advanced class for complex cases.

5.3.1 The `eval_speed` function

To compare `Pipeline`s requiring the same input data, the `eval_speed` function, contained in PyLAn's `evaluation` module, can be used.

This function accepts as input a list of the `Pipeline`s to be compared, a dictionary containing the input data for the pipelines and, optionally, the number of times each `Pipeline` should be run during the speed evaluation (the default value is 100).

When the `eval_speed` function is run, every given `Pipeline` is run the specified number of times, and the average run time is computed and stored, together with the `Pipeline`'s results, inside a named tuple. The output returned by the `eval_speed` function is a list containing all the generated named tuples, one for each pipeline.

The results of the pipelines can be then compared by the user to evaluate the accuracy of each `Pipeline`.

5.3.2 The `Profiling` class

For more complex comparisons, a `Profiling` class can be created by inheriting from the `ProfileBase` base class, included in PyLAn's `evaluation` module. The functionality of the `ProfileBase` class takes inspiration from Python's unit test libraries and includes test fixture, cases, and a test runner.

The functions to be compared can be added to the `Profiling` instance as class methods, and they need to have the following characteristics:

- the method name has to start with the prefix "`profiling_`"
- the output has to be composed of a runtime and a numerical result.

The `run` method, the main method of the class, is implemented in the `ProfileBase` class. When it gets called, all the methods with names starting with "`profiling_`" are run for a specified number of times. For each method, the mean runtime and result, along with their standard deviations, are stored in a dictionary that is then returned to the user after all the profiling methods are executed.

There are different ways to provide the number of times a profiling method should be run. A default iteration value can be optionally passed in the class initialiser (if nothing is provided, the default is 100), but for every added profiling method a different iteration number can be specified with an `iterate` decorator to which the specific number of iterations is passed.

Inside a `Profiling` instance it is also possible to implement a `SetUp` and a `TearDown` method. These methods are run respectively before and after every iteration of the profiling methods. Inside the `SetUp`, for example, the input data needed by the profiling functions can be loaded and prepared.

During the `run`, every raised exception gets caught and printed on the screen.

5.4 Future work

PyLAn is still in its early stages of creation, and not all of the planned functionality has been implemented.

Some of the next features that are planned to be added to PyLAn include the possibility to add aggregation functions to the `Pipeline`s and run `Pipeline`s on the Spark cluster or on the batch service.

In addition to these features, some tests on the computer memory employed during the run of a `Pipeline` will be held, in order to better understand the impact of a pipeline execution on memory and and consequently optimise it.

Aggregation functions

Currently, a `Pipeline` only accepts functions that operate on data relative to a single shot, and return results relative to that same shot. However, many analysis routines also require aggregation functions such as averages, sums, standard deviations, moving averages, etc. For this reason, the possibility to add aggregation functions to the `Pipeline`s is planned.

One of the main challenges of this feature consists of understanding how the input data needed by the aggregation functions can be passed to the `Pipeline`, especially when this latter is run by a `DataManager` object.

Analysis on the Spark cluster or on the batch service

Many analysis routines can be computationally heavy and require significant processing power. To improve performance, it would be useful to take advantage of external resources, such as the Spark cluster or the batch service.

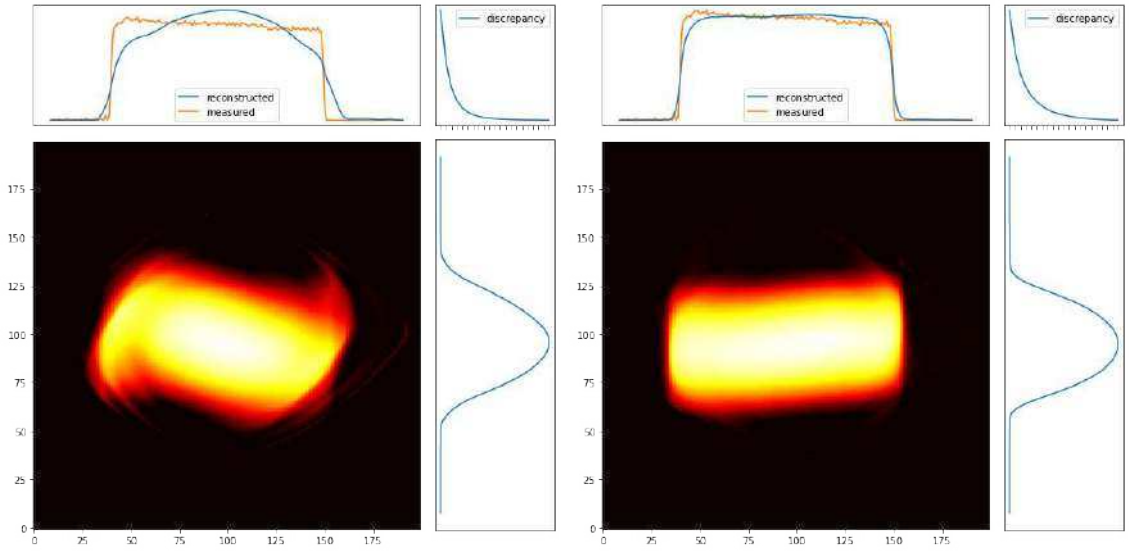
Thanks to its optimised performance and scalability, the Spark cluster is an ideal resource to run data analysis on large data sets. To enable the use of Spark, new features will need to be added to the `DataManager` class, such as allowing data to be loaded and managed into a Spark dataframe, instead of a Pandas dataframe, and running the `Pipeline`s on the Spark cluster.

Another plan is to break pipelines into smaller jobs that can be run in parallel on the batch service. This will allow using even larger memory capacity and more powerful CPUs, and possibly GPUs.

5.5 Uses

PyLAn is currently being utilized to run voltage calibrations in the PSB. To accomplish this, longitudinal tomography and voltage optimization functions have been added to a `Pipeline` object. This object has then been executed on a dataframe that contains data obtained from live acquisitions performed using PyDAq.

The results obtained from the analysis will then be used to study and optimise the performance of the PSB.



(a) Reconstruction pre-optimisation *(b) Reconstruction post-optimisation*

Figure 5.1: Phase space reconstructions of the beam at flat bottom, at 6 kV, before and after voltage optimisation, for PSB ring 1, sector 13. The analysis of the data has been made with PyLAn

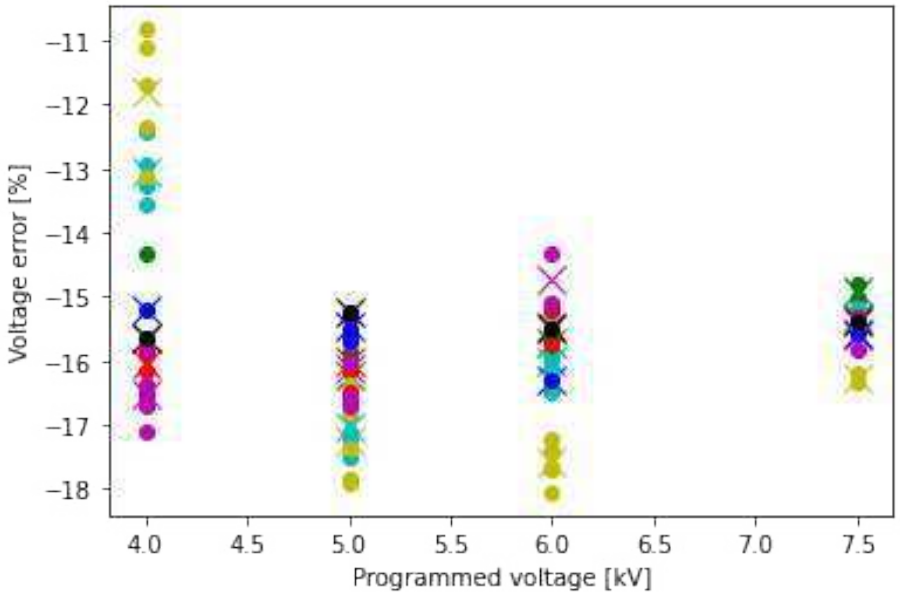


Figure 5.2: Voltage errors for different values of the programmed voltage. The different colours correspond to different settings in the PSB. The analysis has been made with PyLAn.

Conclusions

In this work, I have outlined the basis for two Python libraries designed to simplify and standardise the acquisition and analysis of beam data from CERN's accelerators, in order to improve the performance of beam studies, especially regarding the longitudinal plane.

The first library, PyDAq, is focused on data acquisition, and offers functionality to save, or return to the user, live or logged data, and to perform parametric scans. To do this, some Python classes are provided, but it is also possible to launch the acquisitions directly from the command line, after writing the requirements in an input text file. The acquired data are grouped by cycle, and they can be saved in a well-defined and standardised structure, or returned to the user. Multiple formats are supported both for input and output, and more can easily be added, increasing the code's flexibility. Some built in functions are also available to change the format of the saved data.

The possibility to track a single beam instance through different accelerators has been implemented, opening the way to a new series of measurements and studies, that can be of great help to improve the beam quality in the accelerators, including the LHC.

A first complete version of the library is almost ready, needing the implementation of only one more class, and it will be released soon. A test version is already available, and it has been used to perform various tasks, in different machines, receiving overall positive feedback.

The second library, PyLAn, is a collaborative library for longitudinal analysis, and it is optimised to work well with data saved through PyDAq. This library is still in its early stages of creation, but it already contains a framework to automate the process of running sequences of functions on large sets of data, along with tools to evaluate the speed and accuracy of functions, or sequences of functions. Further studies will be made to increase the performance of data analysis, also making use of external tools such as the Spark cluster or the batch service.

Glossary

AD Antiproton Decelerator, synchrotron used to decelerate and store antiprotons. 6, 10, 14, 15

ALICE A Large Ion Collider Experiment, a LHC experiment dedicated to heavy-ion physics.. 10, 12

ATLAS General-purpose LHC experiment. 10, 11, 12

AWAKE Advanced WAKEfield Experiment, proof-of-principle experiment investigating wakefield plasma acceleration of electrons using a proton bunch as a driver. 6, 8, 13

BCT Beam Current Transformer, the beam instrument measuring the intensity of the beam. 36, 38, 56

bunch steady longitudinal distribution of particles around the synchronous particle. 8, 13, 19, 22, 30, 54, 59

cavity Radio-frequency (RF) cavity are electric resonators used to accelerate the beam in the longitudinal plane and to manipulate the longitudinal motion of the beam. 4, 7, 8, 9, 17, 19, 22, 28, 55

CERN European Organization for Nuclear Research, international organisation dedicated to the development of particle physics and nuclear physics. 4, 5, 7, 8, 10, 11, 14, 17, 21, 31, 58, 66

CLEAR CERN Linear Electron Accelerator for Research, independent linear accelerator at CERN, hosting different experiments. 8

CMS Compact Muon Solenoid, general-purpose LHC experiment. 10, 12

cycle A complete set of accelerator parameters that have to play through in order to produce a given beam. 15, 16, 24, 31, 32, 36, 37, 39, 41, 42, 43, 45, 56, 57, 58, 61, 66

East Area Experimental area situated at the PS. 6, 14

ELENA Antiproton deceleration ring, situated after the AD. 10, 11, 14, 15

emittance Area in the phase space occupied by the beam. 7, 10, 15, 25, 53

FESA Front-End Software Architecture, a real-time application framework that allows accessing the Front End Computers, through which almost all the accelerators control is done. 31, 38, 39

HiRadMat Installation dedicated to the test of material samples and accelerator components under beam impact. 6, 13

intensity The number of particles composing a beam. 7, 13, 15, 36, 37, 43, 54

ISOLDE Isotope mass Separator On-Line DEvice, CERN's facility for nuclear physics. 5, 14, 15, 32

LEIR Low Energy Ion Ring, synchrotron accelerating ions from Linac3, before injecting them in the PS. 6, 7, 10, 15, 45, 58

LHC Large Hadron Collider, the biggest accelerator at CERN, it collides particles in four points, where experiments are located. 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 22, 58, 66

LHCb Large Hadron Collider beauty, an LHC experiment specialised in studying the decay of particles containing B mesons. 10, 12

Linac3 Linear accelerator, performing the first step of ion acceleration in CERN's accelerator complex. 6, 7, 10

Linac4 Linear accelerator, performing the first step of proton acceleration in CERN's accelerator complex. 5, 7, 9, 15

low-level RF system which provides the RF cavities with the necessary small-amplitude signals at the correct frequency and phase. 8, 17, 36, 55

MD Machine Development beam, beam played not to be provided to physics users, but to perform research activities and machine studies.. 16, 18

North Area Experimental area situated at the SPS. 6, 13, 32

nTOF Experimental facility studying neutrons. 6, 14, 32

NXCALS NeXt CERN Accelerator Logging System, logging system storing data coming from the accelerators. 31, 33, 34, 35, 41, 42, 44, 46, 49, 58

PLS user a "container" of the collection of parameters that must be applied in oder to produce a specific beam. 15, 16, 32, 33, 44, 49, 56, 57

PS Proton Synchrotron, accelerator in CERN's accelerator complex. 5, 6, 9, 10, 14, 15, 16, 21, 45, 54, 58

PSB Proton Synchrotron Booster, accelerator in CERN's accelerator complex. 5, 7, 9, 10, 14, 15, 16, 21, 32, 36, 38, 43, 45, 53, 55, 56, 58, 64

PyDAq Library developed for data acquisition. 4, 41, 42, 43, 45, 46, 50, 51, 53, 54, 55, 56, 57, 58, 59, 61, 64, 66

PyJapc Python interface to JAPC (Java API for Parameter Control), a unified Java API to access different types of devices present in the CERN control system. 31, 32, 41, 45, 46, 49

PyLAn Library developed for longitudinal analysis. 4, 59, 60, 62, 63, 64, 66

RFQ RF Quadrupole, RF cavity acting also on the transversal plane, focusing the beam. 8

SPS Super Proton Synchrotron, accelerator in CERN's accelerator complex. 6, 10, 13, 15, 16, 32, 45, 54, 58

supercycle A pre-defined sequence of cycles to be played in a machine, one after the other. 15, 16, 45, 57, 58

synchrotron circular accelerator in which the particles travel in a fixed closed-loop trajectory. 7, 9, 10, 19, 21, 28

tomoscope application that allows to measure the longitudinal distribution if bunch profiles. 38, 39, 43, 57

Appendix A

Examples of input files for PyDAq acquisitions

As mentioned in 4.2.5, PyDaq's acquisitions can be directly launched by calling a function or from the command line, after providing an input text file containing the requirements for the acquisitions. The plan is to have many supported formats for the input file, but at the moment only JSON is supported. In the following paragraphs, how to write a JSON input file will be explained, and some examples will be shown.

The JSON input file

In JSON format, information is stored inside a dictionary, which can contain various levels of sub-dictionaries. The JSON input file for PyDAq acquisitions is expected to contain one sub-dictionary called "Global", with the acquisition specifications, followed by one subdictionary for every machine-PLSuser pair involved in the acquisitions, with the names of the needed signals. This means that for single machine acquisitions the JSON file will contain two sub-dictionaries, the Global one and the one for signals specifications, while for multiple machine acquisitions three or more sub-dictionaries will be present.

The "Global" sub-dictionary

The "Global" subdictionary contains various information. First of all, the source of the data (pyjapc or NXCALS) has to be specified, then some acquisition specifications, dependent on the source of the data, have to be given inside a dictionary called "specifications". For pyjapc acquisitions, the shots number is expected, while for NXCALS acquisitions, information about the time window of the required data must be provided. Lastly, if the acquired data are intended to be saved, the information about how and where to save them has to be given, inside a dictionary called "Save".

The machine sub-dictionary

The specifications about the signals to acquire have to be contained in a sub-dictionary having, as a key, the name of the accelerator to take the signals from.

The sub-dictionary requires information about the PLS user, and then about the signals themselves, divided in *acquisitions* and *settings*. While for the acquisitions a new value is expected at every cycle, settings can be cycle-independent. Completeness and synchronisation is checked only on the acquisitions, while for the settings the most recent value is saved regardless of the cyclestamp. The signal names have to be provided with the `DEVICE/Property#field` syntax, together with an alias name and, optionally, a timing selector override.

Below, some examples of JSON input files are shown.

```
{
    "Global": {
        "Source": "pyjapc",
        "Specifications": {
            "Shots Number": 5
        },
        "Save": {
            "Format": "npy",
            "Path": "Results",
            "Name": "DataSM"
        }
    },
    "PSB": {
        "User": "TOF",
        "Acquisitions": {
            "BCT_R1": [
                "BR1.BCT-ST/Samples#samples",
                null
            ],
            "BCT_R2" : "BR2.BCT-ST/Samples#samples",
            "BCT_R3" : "BR3.BCT-ST/Samples#samples",
            "BCT_R4" : "BR4.BCT-ST/Samples#samples"
        }
    }
}
```

```
{
    "Global": {
        "Source": "pyjapc",
        "Specifications": {
            "Complete_multiple": true,
            "Shots Number": 5
        },
        "Save": {
            "Format": "npy",
            "Path": "Results",
            "Name": "DataMM"
        }
    },
    "PSB": {
        "User": "MD2",
        "Acquisitions": {
```

```
            "BCT_R1": [
                "BR1.BCT-ST/Samples#samples",
                null
            ],
            "BCT_R2" : "BR2.BCT-ST/Samples#samples",
            "BCT_R3" : "BR3.BCT-ST/Samples#samples",
            "BCT_R4" : "BR4.BCT-ST/Samples#samples"
        }
    },

    "PS": {
        "User": "SFTPRO3",
        "Acquisitions": {
            "BCT": "PR.BCT-ST/Samples#samples"
        }
    },
    "SPS": {
        "User": "SFTPRO1",
        "Acquisitions": {
            "BCT_SPS": "SR.BCT.31458-ST/Samples#samples"
        }
    }
}
```

```
{
  "Global": {
    "Source": "NXCALS",
    "Specifications": {
      "Start time": "2022-11-04 08:30:00.000",
      "End time": "2022-11-04 17:30:00.000",
       },
    "Save": {
      "Format": "npy",
      "Path": "Results",
      "Name": "MTEData"
      }
    },
  "PSB": {
    "User": "NORMGPS",
    "Acquisitions": {
      "BCT_RING1": "BR1.BCT-ST/Samples#samples",
      "BCT_RING2": "BR2.BCT-ST/Samples#samples",
      "BCT_RING3": "BR3.BCT-ST/Samples#samples",
      "BCT_RING4": "BR4.BCT-ST/Samples#samples"
    }
  }
}
```

Bibliography

NXCALS documentation, 2023. URL https://nxcals-docs.web.cern.ch/current/.

G. Aad, T. Abajyan, B. Abbott, et al. 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, sep 2012.

D. Alesini. *Proceedings of the CERN-Accelerator-School course: Introduction to Particle Accelerators*, pages 62–92. CERN, 2022. doi: 10.48550/ARXIV.2103.16500. URL https://arxiv.org/abs/2103.16500.

Atlas website. URL https://atlas.cern/Discover/Detector.

BR webpage. URL https://sy-dep-rf.web.cern.ch/content/radio-frequency-group.

Cern website. URL https://home.cern/science/accelerators/large-hadron-collider.

S. Chatrchyan, V. Khachatryan, A. Sirunyan, et al. Observation of a new boson at a mass of 125 gev with the cms experiment at the lhc. *Physics Letters B*, 716(1):30–61, sep 2012.

H. Damerau. *Proceedings of the CERN-Accelerator-School course: Introduction to Particle Accelerators*, pages 641–671. CERN, 2022. doi: 10.48550/ARXIV.2108.06237. URL https://arxiv.org/abs/2108.06237.

P. Forck. *Proceedings of the CERN-Accelerator-School course: Introduction to Particle Accelerators*, pages 692–747. CERN, 2022. doi: 10.48550/ARXIV.2009.10411. URL https://arxiv.org/abs/2009.10411.

S. D. E. Fortescue-Beck. Introduction to the be-co control system, 2020a. URL https://cds.cern.ch/record/2748122?ln=it.

S. D. E. Fortescue-Beck. *Introduction to the BE-CO Control System*, pages 158–161. 2020b.

S. D. E. Fortescue-Beck. *Introduction to the BE-CO Control System*, pages 147–148. 2020c.

E. Gschwendtner and P. Muggli. Plasma wakefield accelerators. *Nature Reviews Physics*, 1(4):246–248, 2019. doi: 10.1038/s42254-019-0049-z. URL https://doi.org/10.1038/s42254-019-0049-z.

E. Gschwendtner, K. Lotov, P. Muggli, et al. The awake run 2 programme and beyond. *Symmetry*, 14(8), 2022. ISSN 2073-8994. doi: 10.3390/sym14081680. URL https://www.mdpi.com/2073-8994/14/8/1680.

Hadoop website, 2023. URL https://hadoop.apache.org/.

S. Hancock and S. J-L. A pedestrian guide to online phase space tomography in the cern complex. 01 2001.

JAPC user guide, 2023. URL https://readthedocs.web.cern.ch/display/RADE/JAPC+User+Guide.

Kafka website. URL https://kafka.apache.org/.

I. Karpov, T. Argyropoulos, and E. Shaposhnikova. Thresholds for loss of Landau damping in longitudinal plane. *Phys. Rev. Accel. Beams*, 24(1):011002, 2021. doi: 10.1103/PhysRevAccelBeams.24.011002.

S. Y. Lee. *Accelerator Physics*. World Scientific Publishing, 2019.

D. Quartullo, S. Albright, and H. Damerau. Frequency-Dependent RF Voltage Calibration Using Longitudinal Tomography in the CERN PSB. *JACoW*, IPAC2022:845–848, 2022. doi: 10.18429/JACoW-IPAC2022-TUPOST006.

F. Roncarolo, G. Bellodi, E. Bravin, et al. Overview of the cern linac4 beam instrumentation. 01 2010.

Spark website. URL https://spark.apache.org/.

SY-RF website. URL https://sy-dep-rf.web.cern.ch/content/radio-frequency-group.

SY website. URL https://sy-dep.web.cern.ch.

F. Tecker. *Proceedings of the CERN-Accelerator-School course: Introduction to Particle Accelerators*, pages 351–367. CERN, 2022. doi: 10.48550/ARXIV.2011.02932. URL https://arxiv.org/abs/2011.02932.

B. Urbaniec and L. Burdzanowski. Cern controls configuration service - event-based processing of controls changes. In *Proc. ICALEPCS'21*, number 18 in International Conference on Accelerator and Large Experimental Physics Control Systems, pages 253–256. JACoW Publishing, Geneva, Switzerland, 03 2022. doi: 10.18429/JACoW-ICALEPCS2021-MOPV043. URL https://jacow.org/icalepcs2021/papers/mopv043.pdf.

H. Wiedemann. *Longitudinal Beam Dynamics*, chapter 9. Springer International Publishing, Cham, 2015. ISBN 978-3-319-18317-6. doi: 10.1007/978-3-319-18317-6_9. URL https://doi.org/10.1007/978-3-319-18317-6_9.