

Bachelor Thesis:

A Python Library for Longitudinal Beam Tomography

Christoffer Hjertø Grindheim

Abstract

Measuring the longitudinal phase space distribution of particle beams is needed for a myriad of purposes. Among these are machine calibration and bunch quality assessment. Since the tomographic reconstruction algorithm was developed, it has been a vital part of operation at CERN.

The current program is outdated, complex and hard to develop further. Its functionality should be able to be customised for each machine, and for the needs of the individual users. For this reason, a new version must be developed, able to serve the wide range of requirements the future might bring.

This dissertation describes the translation from a program written in highly optimised Fortran95 code, to a modular Python library. It also addresses experiments with a hardware accelerated implementation of the tomographic reconstruction routine. Finally, the code is profiled with respect to both output quality and run time, and it is confirmed that it is ready for operational usage.

Contents

1	Introduction	4
1.1	Contracting Entity	4
1.2	Problem Description	4
1.2.1	A Brief Introduction to Accelerator Physics	4
1.2.2	The Background for the Problem	7
1.2.3	The Problem	8
1.3	Main Idea of Solution	8
2	Problem Analysis	9
2.1	Specification of Requirements	9
2.2	Description of Possible Solutions	10
2.2.1	Solution Alternatives	11
2.2.2	Issues Regarding Tools and HW/ Software (SW) Components	11
2.3	Problem Analysis Conclusion	11
3	Realisation of Selected Solution	13
3.1	The Fortran 95 Source Code	13
3.1.1	About the Program	13
3.1.2	Running the Fortran Program (Input and Output)	13
3.1.3	Program Modules and Structure	14
3.1.4	Program Flow	14
3.2	Python Translation	20
3.2.1	Goals for the Translation	20
3.2.2	First Translation - Program Flow and Structure	20
3.2.3	Algorithmic Changes	21
3.2.4	Restructuring from Program to Library	25
3.2.5	Floating Point Error	28
3.3	Hardware Acceleration	30
3.3.1	A Quick Introduction to GPUs	30
3.3.2	Tools Used	31
3.3.3	Accelerating the Particle Tracking	31
3.3.4	Accelerating the Reconstruction Routine.	34
4	Testing	36
4.1	Output Quality	36
4.2	Run Time: Full Program	38
4.3	Run Time: Minimal program	39
4.4	Unit Tests and Documentation	40

5	Discussion	41
5.1	Planning	41
5.2	Translation Process	41
5.3	Speeding Up the Code	42
5.4	Run Time and Output Quality	43
6	Conclusion	44
6.1	Further Improvements	45
	Appendices	48
A	Abbreviations and Dictionary Explanations	49
B	Example Fortran Input File	51
C	Table of Optimal Reconstruction Settings for Reference Files	54
D	Program Documentation	55

Preface

First and foremost, I would like to give a large thank you to my supervisor Dr. Simon Albright for his excellent guidance, and for giving me the opportunity to work with him in this project. Also, I would like to thank him for everything he has taught me during the last 13 months. I would further like to extend my gratitude to Konstantinos Iliakis for overseeing this project in relation to the field of computer science. His advice has tremendously enhanced the project. I would also thank Dr. Alexandre Lasheen for the interesting discussions and important input. To all the members of the BE-RF-BR for keeping up with me, and sharing your expertise and insight. At last, I would like to give a special thank you to Lukas Matter and Dhiren Jhugroo for their comments in the writing of this dissertation.

Chapter 1

Introduction

1.1 Contracting Entity

CERN (European Organization for Nuclear Research) is a European research institution, located near Geneva, Switzerland. Over 12 200 scientists of 110 nationalities work together, with the mission to uncover what the universe is made of and how it works. This is done by providing particle accelerator facilities to researchers [1].

BE-RF-BR (Beams Department, Radio-Frequency Group, Beams & RF Studies Section) deals with the interactions of beams with radio frequency (RF) systems and their environment, primarily in the longitudinal plane. The BR section members are strongly involved with performance and upgrades of existing accelerators. The section is able to measure and calculate the impedance of various machine elements using many different methods, including beam measurements. Beam diagnostics in the longitudinal plane (bunch profiles, phase space tomography, Beam Quality Monitors, Schottky spectra...) are also a field of interest and support. The section is developing the Beam Longitudinal Dynamics code (BLonD) [2] which is heavily used in numerous particle simulations, vital to support beam measurements, RF manipulations and to predict beam performance after upgrades [3].

1.2 Problem Description

1.2.1 A Brief Introduction to Accelerator Physics

CERN is a laboratory for High Energy Physics. Here, accelerators are used to rise the energy of particles to high energies. The particles are then used for experiments. This is done by sending the particles through a chain of accelerators. The stream of particles is called the beam. Normally, This consists of one or more bunches, which are smaller groups of particles. As the beam travels through the accelerators, they gain more and more energy until they are at a sufficiently high energy to be used for experiments.

The accelerator chain starts with a linear accelerator, followed by a series of circular accelerators. This can be seen in figure 1.1. The first circular accelerator, the Proton Synchrotron Booster (PSB), receives a beam from the linear accelerator. As the particles gain more energy, they will be injected in to the next accelerators in the following order: the Proton Synchrotron (PS), the Super Proton Synchrotron (SPS), and finally, the Large Hadron Collider (LHC). Each accelerator has a larger circumference than the preceding in order to cope with the particles increase in energy.

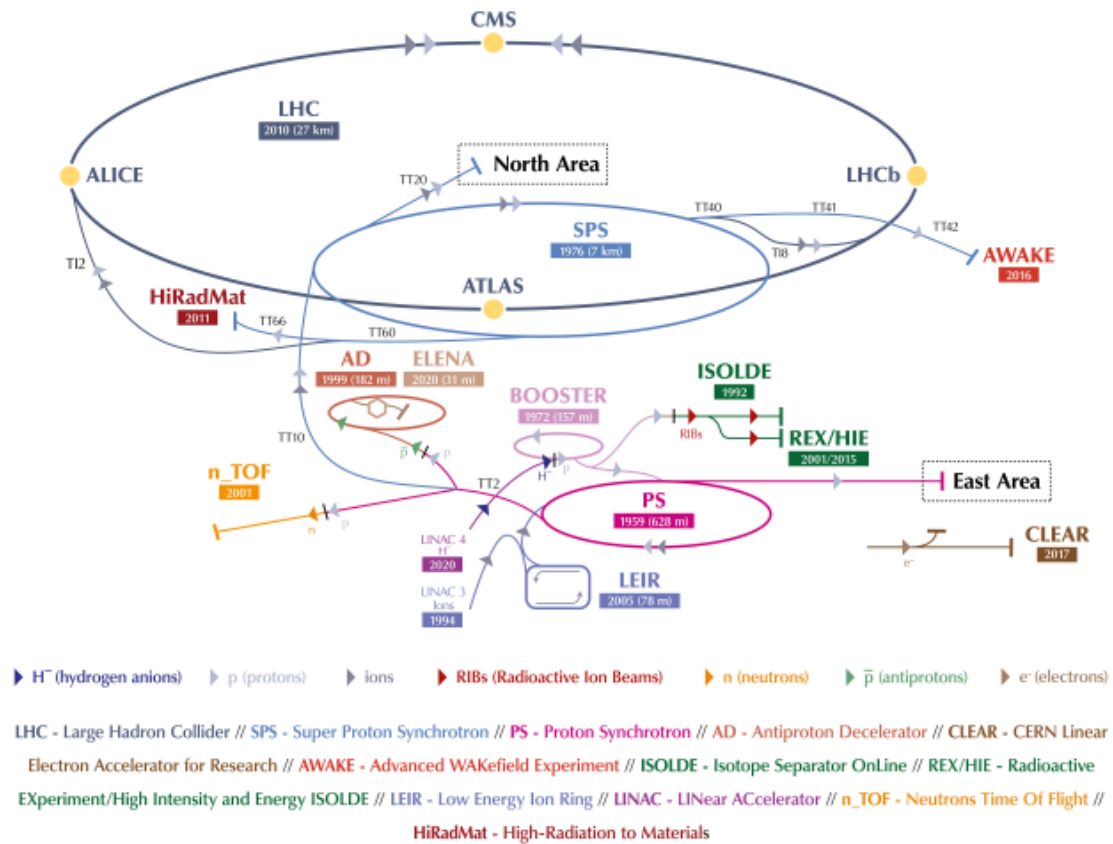


Figure 1.1: The CERN accelerator complex. CC: CERN

Most of the accelerators at CERN are synchrotrons. These consist of a series of magnets bending the particles to follow a circle, and radio frequency (RF) stations changing the energy of the particles. Every time the particles passes the RF stations, they gain an energy kick. Then, as the particles travel trough the ring, the particle phase will drift. This movement in phase and energy is called the synchrotron motion. In figure 1.2 a simple example of a particle accelerator can be seen.

A theoretical particle named the synchronous particle can be introduced in order to simplify the calculation for the particles change in energy and phase. Here, the synchronous particle, set at the ideal phase relative to the RF voltage, is used as a reference for the other particles phase and energy coordinates. The phase between the synchronous particle and the RF voltage determines the energy gain for the particle bunch at each turn. In figure 1.3a, the synchrotron motion relative to the synchronous particle can be seen. Here, the synchronous phase is zero, which means that no energy is gained for the bunch at each turn. However, the RF voltage still changes the energy for the particles at an individual level within the bunch. Using the synchronous particle is not strictly needed in order to calculate the synchrotron motion, but it is used as a convenient tool for the tomography program.

When looking at the synchrotron motion below transition crossing, particles preceding the synchronous particle will see a negative voltage. In figure 1.3a, the gray dot is at such a position. This will have an early arrival, which makes it see a negative voltage, and thereby lose energy. The particle will then drift

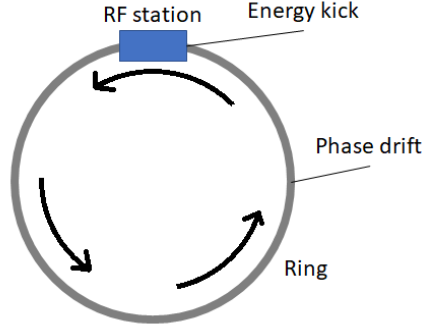


Figure 1.2: Simple accelerator. Bunches receives an energy kick when passing by the RF station. A phase drift happens in the rest of the ring.

backwards in phase. When the particle is at the position of the blue dot in the same figure, the phase will be the same as for the synchronous particle, but the energy will still be lower. This will make it continue to drift until it is at the position of the green dot. Here, it will be at the same energy as the synchronous particle, but because of the phase drift, it will have a late arrival resulting in seeing a positive voltage. This gives the particle more energy, and it will start to drift in phase towards again being a leading particle. For a stable bunch, such a process is repetitive, and results in an elliptical movement in phase space.

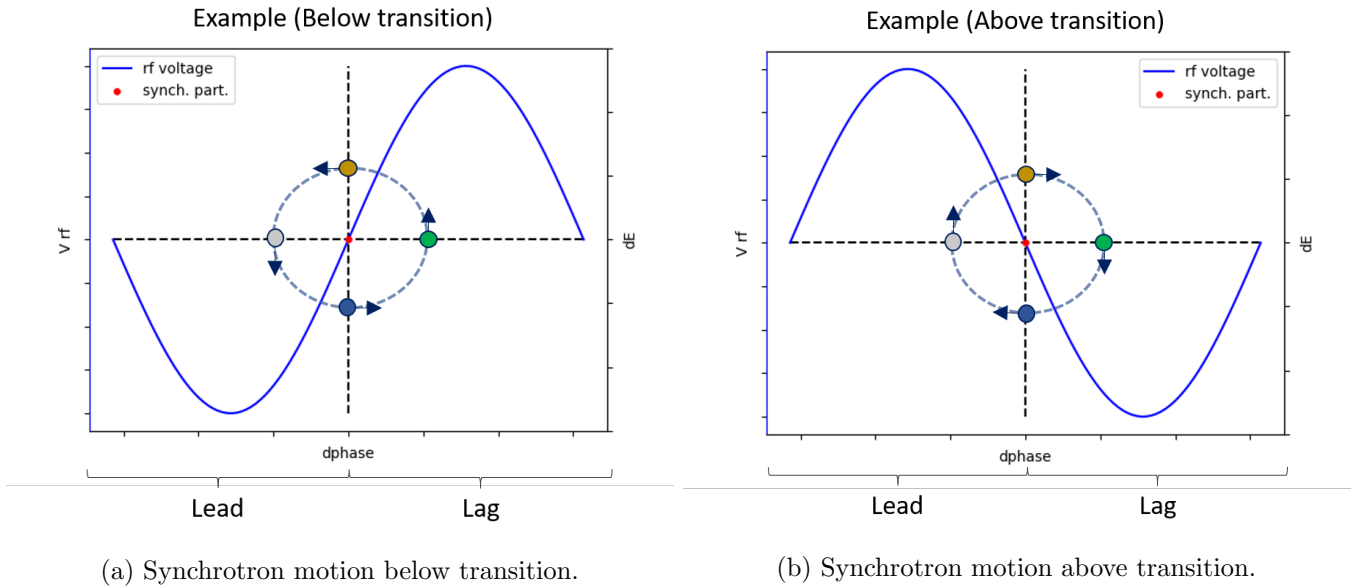


Figure 1.3: Synchrotron motion with with synchronous particle at zero.

When the particles are at energies below transition, higher energies lead to higher revolution frequency. Meanwhile, above transition crossing the ratio between rise in mass and rise in velocity changes in the benefit of mass. This means that the trajectory of the particles with the highest energies will be longer than the ones with lower energies. As the gain in velocity is not sufficient to keep up with the larger orbit radius, the higher energy particles have a lower revolution frequency and will start to lag behind (see figure 1.3b for the synchrotron movement above transition). This means that the movement above transition is the opposite of the movement below transition. Also, this means that a bunch at an RF frequency stable below transition

will be unstable above transition.

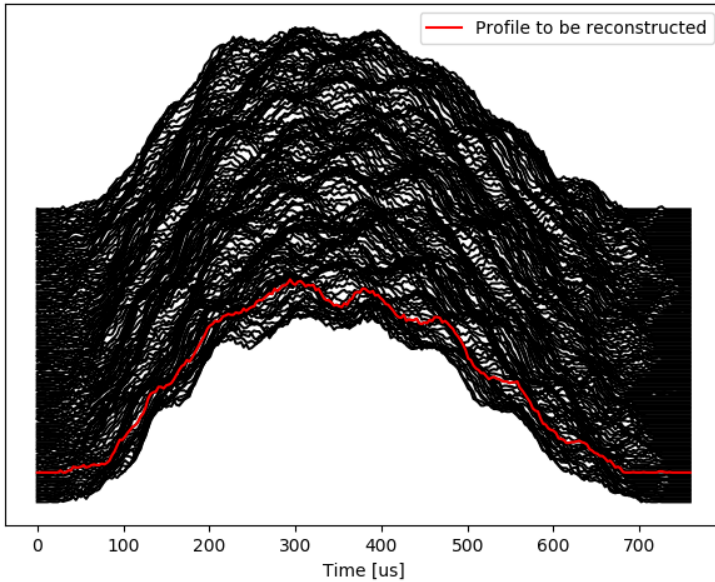
Particles having a too large energy or phase difference relative to the synchronous particle will be lost as the energy gain slips out of control. The boundary in phase and energy where the particles goes from stable to unstable motion is called the separatrix. The area within the separatrix is one way to define the bunch emittance, an important measure for the bunch.

1.2.2 The Background for the Problem

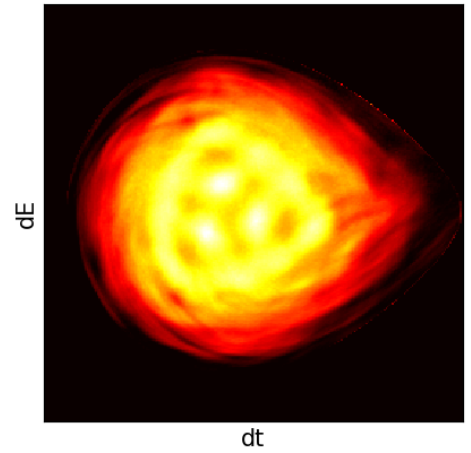
Tomography is the method of reconstructing a n -dimensional object from its $(n-1)$ -dimensional projections. The underlying principle of tomography is to combine the information in a sufficiently large number of profiles to be able to reconstruct unambiguously the fuller picture with the extra dimension reinstated. For example, many one-dimensional profiles of x-ray transparency taken from different angles can give doctors an image of a two-dimensional slice through a patient [4].

The same principle may be used for longitudinal beam tomography, but instead of using x-rays this method is based on a series of one-dimensional measurements of the bunch profiles. The bunch profile measurements are snapshots of the bunches, measured in induced current or voltage as they pass by a sensor, often a wall current monitor [5]. The technique takes advantage of the fact that the bunches are rotating in phase space, which gives them a small change of angle for each measurement. The algorithm currently utilised and developed by CERN can from this data create a two-dimensional image of the time and energy distribution of a bunch in phase space.

While accelerating particles in a synchrotron, an overview of the phase space density distribution it is desired. This is important to understand the qualities and the behaviour of the bunch during the cycle, and that the bunches fit their requirements. An example of such a quality assessment can be found when injecting a bunch from one accelerator to the next. By analysing the featured of the received bunch the following accelerator, one can determine that the transfer was done correctly. In this way, it can be assured that the quality of the bunches delivered are following the standards needed by the experiment.



(a) Mountain range of 1D profile measurements (input).



(b) Reconstructed phase space distribution (output).

Figure 1.4: Input and output of the tomography program.

In figure 1.4a one can see the measured input, presented as a mountain range of one dimensional profiles.

Each profiles corresponds to the measurement of a passing bunch. The profile marked in red is the profile to be reconstructed by the tomography program. In figure 1.4b the resulting phase space can be seen. This is the output of the program, and shows the particles distribution in phase space. In this example, a high harmonic RF system is phase modulated at a frequency that resonates with particles in the bunch. This resonance causes the structure seen in the figure.

1.2.3 The Problem

The original program used for tomography was written in the language Fortran95, and has been optimised for the computing tools and languages available at the time it was written. A new and more modern version of the program is desired for further development and easier maintenance. An important factor is that the new version of the program should at least be as fast as the old version.

Today, CERN is in the period of the Long Shutdown 2 (LS2). Here, the accelerators are not operative and are going through large upgrades. There is an intent to investigate the possibility of using the program in all seven synchrotrons located at CERN. Since there are different needs for the different synchrotrons it is yet to be decided which demands for the program have the highest priority. It is also a necessity that the program can run seamlessly with computing environment used in the CERN system.

A possible extension is to get the run time of the program small enough to make one recreation per machine cycle. In the PSB, the smallest machine at CERN, this is less than 0.5 seconds. Such a short run time is needed in order to do online measurements of the acceleration process, and to perform reconstructions at both injection and extraction of the machine. Another possible extension is to combine the tomography program with a newly developed method for reading bunch profile data and measuring bunch length [6].

1.3 Main Idea of Solution

The solution of the problem is divided into two steps. First, to translate the current program from Fortran95, to an object-oriented Python version for easy experimentation with the code. After this, the finished Python version will be written fully or partially into C++ for greater speed. When the final Python/C++ version is finished and running correctly, the priorities for extending the programs functionality or further improving the execution time will be handled.

Chapter 2

Problem Analysis

In this chapter, the demands for the program are presented and analysed. Different solutions are proposed and evaluated in order to find the most expedient solution.

2.1 Specification of Requirements

- **A complete solution with all functionality translated from Fortran95 to Python/C++**

The program should be translated into a functionally identical copy of the original program. This means that a given input should produce a very similar output, and that all the various extra functionality in the program should be retained.

- **Must be suitable for use with the operational environment at CERN**

The final version shall be run on the computing environment used in the CERN system.

- **Must be well documented**

The program must be well documented in order to keep a high degree of maintainability and for easy understanding of the functionality of the program. The documentation will consist of both comments in the code, and a separate file mentioning decisions taken during the development of the program. There should also be a user manual for installation and usage of the program.

- **Must have a complete suite of unit tests**

It is important that the unit tests cover all code. Furthermore, the program should provide easy-to-understand exceptions.

- **Run time must be less than or equal to the run time of the original version**

The original program is fast, and it is important that the Python/C++ version can keep up with the current run time. A possible extension, and a long time goal, is to reduce the run time to under 0.5 seconds for an optimal, online tomography.

- **Program input and output should come with different options: pipe-lined and File IO**

For different usages of the program, different ways of providing input and retrieving output will be a necessity. The input should be able to be pipe-lined to and from other programs in a command prompt. It should also be able to use data from files on the disk. A possible extension is to receive data serially, while performing online measurements.

- **A GPU implementation shall be tested to see if it is a usable solution**

For the long-time goal of a run time in less than 0.5 seconds it may be relevant to examine the possible use of a hardware (HW) accelerated solution. After the Python/C++ version is finished, documented and tested; implementing and testing of an accelerated version using a Graphical Processing Unit (GPU) will show if there is potential for such a solution.

2.2 Description of Possible Solutions

After analysing the demands in chapter 2.1, a solution can be created. The solution must fulfil all the demands, but an evaluation must be made if it can be done with the resources and time available. As some of the demands might contradict one another, compromise solutions might be evaluated for the best possible outcome.

The execution time will be of importance as doing tomography in real time during operations is desirable. Since Python is an interpreted language it can be relatively slow when working with large amounts of data. Therefore, it will be necessary to speed up the code when the first translation is written.

There are different ways of speeding up a program, one solution is to rewrite the full program into a precompiled language such as C++. This may make the program a lot faster, but it will also make the program more complex and rigid. Another way to speed up the program is by using Python libraries. Some libraries let your Python script call back and forth to C and C++. An example of such a library is the cython library [7]. In this way, the computationally demanding parts of the program can be outsourced to C++, while the framework remains in Python. This solution makes the code both faster, and easy to maintain. How much of the program is needed to be re-written to C++ depends on the gain by speeding up the code in relation to the loss in flexibility. Another library is Numba, this contains some easy to use decorators to be wrapped around resource demanding Python functions. These decorators will tell the Numba compiler to compile the function to machine-code before execution, which, with the right use, can speed up the code significantly.

When the program is working and optimised, there may be a need of speeding up the program even further if the long-time goal of a run time under 0.5 seconds is to be achieved. In this case, two solutions may be relevant: hardware acceleration using GPUs or field-programmable gate arrays (FPGAs).

FPGAs are fast, but they do not have much support for floating point calculations. This may be a problem as the tomography program is heavily reliant on floating point operations. However, methods of working around this may be investigated further. In addition to this, FPGAs have a somewhat steep learning curve and a relatively small community. Due to lack of earlier experience with such devices there is a concern about the amount of time needed to learn how to use FPGAs, and later to develop such a solution.

A more suitable solution may be the usage of GPUs. Here, libraries for Python and C++ can be easily applied to create a GPU implementation. In addition to this, there is a large community, a notable amount of literature, and numerous courses in GPU programming which will help setting up a solution within a reasonable time frame. GPUs are fast and are able to perform a very large number of floating-point operations each second. On the other hand, a notable amount of time is needed to move data from the CPU memory to the GPU memory, which often have a limited size. Furthermore, GPUs start and stop slower than CPUs. This means that the time spent loading the data into the GPU memory must be compensated by the amount of time saved on the acceleration of the relevant functions.

In addition to speeding up the code, the possibility to perform future modifications of the program is of high importance. For this reason, it is important to write unit tests, custom exceptions and to write a detailed documentation. This will ensure that the program runs smoothly, gives reasonable error messages and contributes with necessary debugging information. Setting up a repository with automatic pipe-lined

testing is a an effective tool to ensure code correctness. Many easy-to-use libraries exists for unit testing in Python and is another reason to keep Python as a framework language for the program. These measures will ensure that coming developers can efficiently start working on the program.

2.2.1 Solution Alternatives

- **Solution alternative 1:** Python accelerated with Python libraries and GPU.
- **Solution alternative 2:** Python with demanding parts written in C++.
- **Solution alternative 3:** Python with demanding parts written in C++ and HW accelerated with GPU.
- **Solution alternative 4:** Full program in C++ and HW accelerated with GPU.
- **Solution alternative 5:** HW Acceleration using FPGAs.

2.2.2 Issues Regarding Tools and HW/ Software (SW) Components

2.2.2.1 Development Environment

Many computers at CERN use Linux based operating systems. Therefore, it is necessary to get an understanding of the Linux environment. The version of Linux on the provided computer is CERN CentOS7. Having no experience with a Linux system, it is expected to spend a week of getting comfortable and learning the basics of the operating system (OS), its commands and features.

For developing and debugging Python, a suitable integrated development environment (IDE) will be needed. It is important that this IDE has debugging tools, works with online repositories such as GitLab and runs on Linux systems. An IDE that fulfils these criterions is JetBrains IDE: PyCharm [8].

For retrieving numbers in order to compare Python output with the original Fortran code an IDE with a Fortran debugger will be useful. The IDE must be available for Linux systems. An IDE that fulfils these criteria is Eclipse by The Eclipse Foundation [9].

A fitting IDE for C++ programming is Visual Studio Code [10]. This is a familiar programming environment for the developer, and works seamlessly with the online repository used at CERN. Visual Studio Code also has an integrated debugger and runs on Linux systems.

The online repository utilised by CERN for this project is GitLab. Having a very limited experience with online repositories, as well as a lack of experience with CentOS and PyCharm, at least one day to learn the basics and get it all to work.

2.2.2.2 Hardware

CERN holds numerous GPUs. A batch service may be used to test the code on many different GPUs, and make it possible to experiment with them in order to find which are contributing with the best results in speed and scalability. For this reason, one cannot decide which GPU is the most suitable for the program prior to testing. However, since this is a scientific computational problem, it is possible to say that some relevant GUI libraries and programming languages to examine will be PyCUDA [11], CUDA [12] and OpenCL [13]. The time expected to get basic knowledge of GPU programming is at least one week.

2.3 Problem Analysis Conclusion

Solution 1, written in Python is regarded as a good start, but experimenting with functions in C++ will tell if optimised C++ code will run faster than precompiled Python code. Seeing this as likely, the solutions 2,

3 and 4 will be preferable. Solution 5, which includes hardware acceleration using FPGAs is seen as very time consuming in comparison to the other options and is therefore disregarded at this point.

The solutions 2, 3 and 4 all includes translation to C++. Together with the choice between these two solutions comes the question of maintainability and easy change and experimenting with the code, as opposed to execution speed. Writing the full program in C++ can reduce the run time of the program, but also make it more rigid and complex. For this reason, the option of writing only the computationally heavy parts to C++ (solution 2 and 3) seems favourable. The choice will be solution 2, with the possibility of expanding the program by introducing GPUs ending up on solution 3.

Chapter 3

Realisation of Selected Solution

In this chapter, the developed solution is described. First, the original program and its algorithms are explained. Secondly, the Python/C++ translation is described focusing on functionality and structure of the code. Also, a difference due to the use of different floating points in the two versions are compared. Lastly, an implementation using a GPU in an attempt to reduce the run time is described. The chapter should provide the reader with sufficient information to understand the logic and functionality of the old and new version.

3.1 The Fortran 95 Source Code

The following is a description of the original program. The description focuses on the algorithm and the structure of the program. Also, the input, and output files and their formatting are described.

3.1.1 About the Program

The original tomography program is written in the compiled high-level language Fortran 95, in a procedural oriented manner. This means that the code is divided into procedures which can be called multiple times. The program is optimised using the High-Performance Fortran (HPF) extension, which includes tools for easy vectorisation and parallelisation of the code. The program is written with the goal of having the shortest possible execution time and the optimal usage of the memory capacities computers had at the time when the program was written. All this adds up to an effective program which produces phase space images of a good quality at a short run time.

3.1.2 Running the Fortran Program (Input and Output)

3.1.2.1 Requirements

In order to run the program, a Fortran 95 compiler is needed. The recommended compiler is the Portland Group Compiler (PGI) [14] as the program is optimized for PGI's "high performance Fortran compiler" (pghpf).

3.1.2.2 Program input

The necessary inputs are the machine parameters, settings for the reconstruction, and the measured data. All of these should be given as text in a file like the example input in seen in appendix B. These files are written in a standardised format, where each line holds the value of an input variable. The measured raw

data can be stored as a separate text file or pipe-lined in the same file. If the measured data is placed in a separate file, the path to this file must be provided in the input file.

3.1.2.3 Program output

The output of the program is provided in two ways; through the systems standard-output, and as files to a specified directory. Via the standard output, information about calculated parameters and the state of the program is provided. This output is needed by a program named 'tomoscope' [15], an operational graphical user interface (GUI) for tomography, in order to perform further analysis and presentation of the recreated phase space distribution. The following table lists all the output files.

imageXXX.data	The reconstructed phase space as a one-dimensional list of floats. XXX is the number of the profile to be reconstructed.
dXXX.data	The discrepancy of the reconstructed phase space for each iteration of the reconstruction. XXX is the number of the reconstructed profile.
jmax.data	Limits of the reconstructed area in the j-axis (energy axis)
ProfileXXX.data	Profile to be reconstructed after rebinning and baseline cut. XXX is the number of the reconstructed profile.

Table 3.1: Fortran Tomography Output Files.

3.1.3 Program Modules and Structure

The program is structured in four different modules, which contain subroutines and functions to be called from the main program. The program modules are listed in table 3.2.

tomo	The main program. This module orchestrates the function and subroutine calls.
long	Module with longitudinal phase space specific routines. Here data is loaded, profiles are created, and particles are tracked.
tomosubs	Module which stores all the subroutines needed for the Tomographic reconstruction.
nrstuff	Numerical formulas from the book Numerical Recipes [16].

Table 3.2: Fortran Program Files.

3.1.4 Program Flow

The program is based on the Algebraic Reconstruction Technique (ART) [17]. However, since this technique is a linear technique it cannot be used directly without running the risk of significant loss of quality. The quality is lost due to non-linear motion of large amplitude particles. However, this is solved by introducing particle tracking to the algorithm. Test particles are spread out in a grid structure which represents the phase space. The initial distribution is homogeneous, where each cell is populated by an equal amount of test particles. The particles movement are tracked trough all machine turns of the measurement. Their positions are saved at each turn where a profile measurement has been made, in coordinates of phase and energy. When all the particles are tracked, one can count how many of them ended up in each bin. this, combined with the measured profiles results in a weight factor given to each of the cells in the grid structure. These

weight factors will be adjusted in the iterations of the tomographic reconstruction and converge towards an image of a reasonable quality.

3.1.4.1 Setting Up the Reconstruction

The program starts by loading the data and parameters from a text file provided by the user. The parameters include settings for the reconstruction, and information about the machine during the measurements. The machine parameters from the input file are used as initial values for calculating the parameters' values at each machine turn. These values will be needed for the particles tracking routines.

The profile measurements are loaded from a text file, and cast to floating points. Then, the raw data goes through data treatment to be ready for the reconstruction process. First, the baseline is found based on the first 5% of the measurements of a reference profile, and subtracted from all data. After this, the arrays containing the profiles are reshaped to (N, M) . N is the number of profiles, and M is the number of bins in each profile. Then, negative values are set to zero, and the profiles are normalised. Lastly, the profiles are re-binned from the shape (N, M) to (N, X) , where X is the original number of bins in a profile, M , divided by the re-bin factor given as a parameter in the input file. When the data treatment is done, the total charge of the beam reference profile is calculated.

If the x-coordinate of the synchronous particle is not provided in the input file, it can be estimated using a linear fit based on the length of the bunch. This bunch length is estimated based on the beam reference profile. The `xat0` field in the input file (see appendix B) can be set to a negative value in order to indicate that an automatic estimation of the synchronous phase should be performed.

When everything is set up, the area of reconstruction is calculated in a couple of steps. First, the size of the bins of the phase space coordinate system is calculated (see chapter 3.1.4.2). Secondly, the upper and lower limits in phase and energy are found. It is within these borders the test particles will have their initial distribution. The borders formed by the limits in phase and energy can be seen in figure 3.1. This area will be referred to as the reconstruction area.

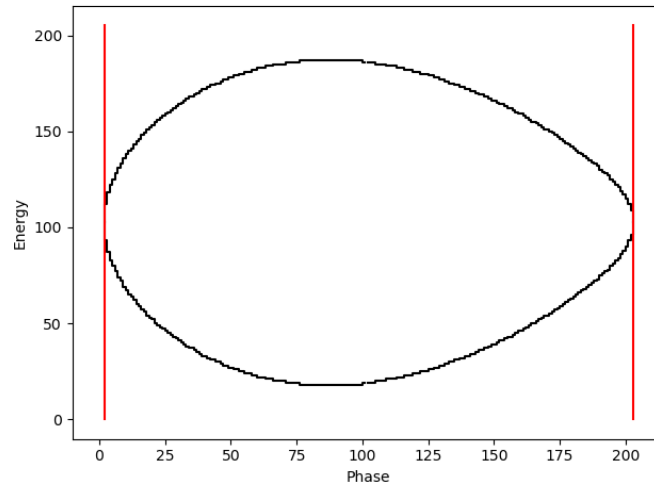


Figure 3.1: Limits of the reconstruction area in phase (red) and energy (black) given as number of bins.

3.1.4.2 Two Coordinate Systems

Two different coordinate systems are used in the program: the phase space coordinate-system and the physical coordinate-system.

The phase space coordinate-system is given in bins as integers. It has its origin at some minimum phase and consists of axes in units of phase and energy. Bins of the horizontal axis are some $\Delta\phi$ wide, and the vertical axis consists of bins which are some ΔE wide. The number of bins in the phase-axis is given by the length in bins of the profile measurements (the sampling time).

The axis of the physical-coordinate system are given in the units of phase and energy difference relative to the synchronous particle. The phase difference is given in radians, and the energy difference is given in electron volts.

3.1.4.3 Particle Tracking and Storing

The particle tracking algorithm is not straight forward, and a complex system of arrays are involved for optimising the memory usage. The following is an overall description of the tracking and sorting routine. Here, particles are distributed and tracked from one time frame (profile measurement) to the next. After this, they are sorted and stored to be used during the reconstruction process. These arrays are named maps, mapsi and mapsweight. An simple overview of the tracking algorithm can be seen in figure 3.2.

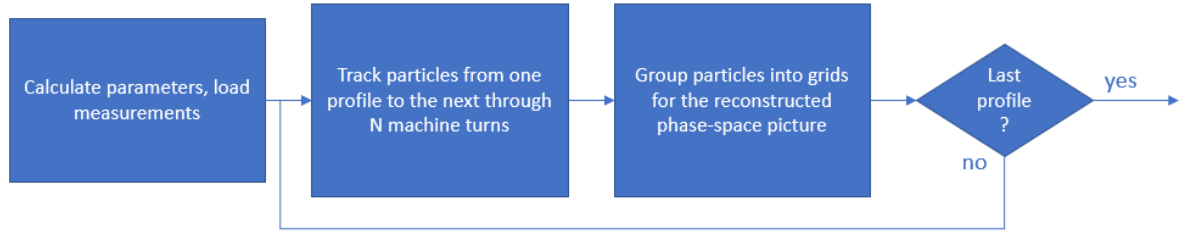


Figure 3.2: Simplified flow chart for the tracking algorithm

Maps is an array of three dimensions holding a number for each cell of the reconstruction area at every phase space. There is one phase space for each time frame. The first cell is marked as cell number one. The cell having the same coordinates in the following phase space receives the number $N + 1$, where N is the number of cells of the first phase space. In figure 3.3, a simple example of a maps array is shown. Here, two phase spaces have a reconstruction area of nine cells. The first phase space holds cells from one to nine, the second holds the cells numbered ten to 18.

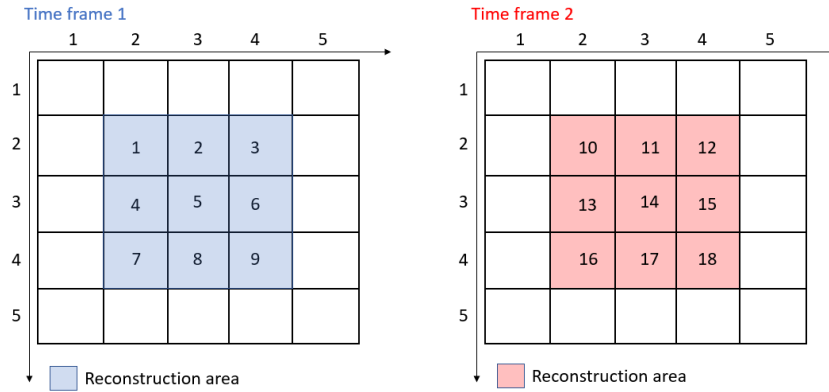


Figure 3.3: Layout and contents of the maps array. Figure shows maps array for first (blue) and second (red) time frame, out of N time frames.

Before the tracking can start, a particle distribution must be generated. This happens at the phase space to be recreated. Here, N particles are distributed homogeneously within each cell of the reconstruction area. This means that each group of particles generated in the same cell are labeled with that cell's index. The indices of the cells are stored in the maps array.

The initial particles have their coordinates as units of bins in the phase space coordinate system. Therefore, their coordinates must be mapped to physical coordinates of phase and energy (see chapter 3.1.4.2) before they can be tracked. The particles are tracked for a number of machine turns, until they reach the next time frame. Here, the particles are mapped back to the phase space coordinate system for sorting and storing in to the arrays mapsweight and maps. These arrays store information about the particles position, and the weight of the cells in the phase space coordinate-system.

Mapsweight and maps are 2D arrays, which have one vector for each index stored in the maps array. In the example of figure 3.3, the number of vectors are 18. Each vector holds information about a group of particles originating from the same cell. Mapsi records the unique bins in which the particles ends up at every time frame. Meanwhile, the mapsweight array stores how many particles ended up in these bins.

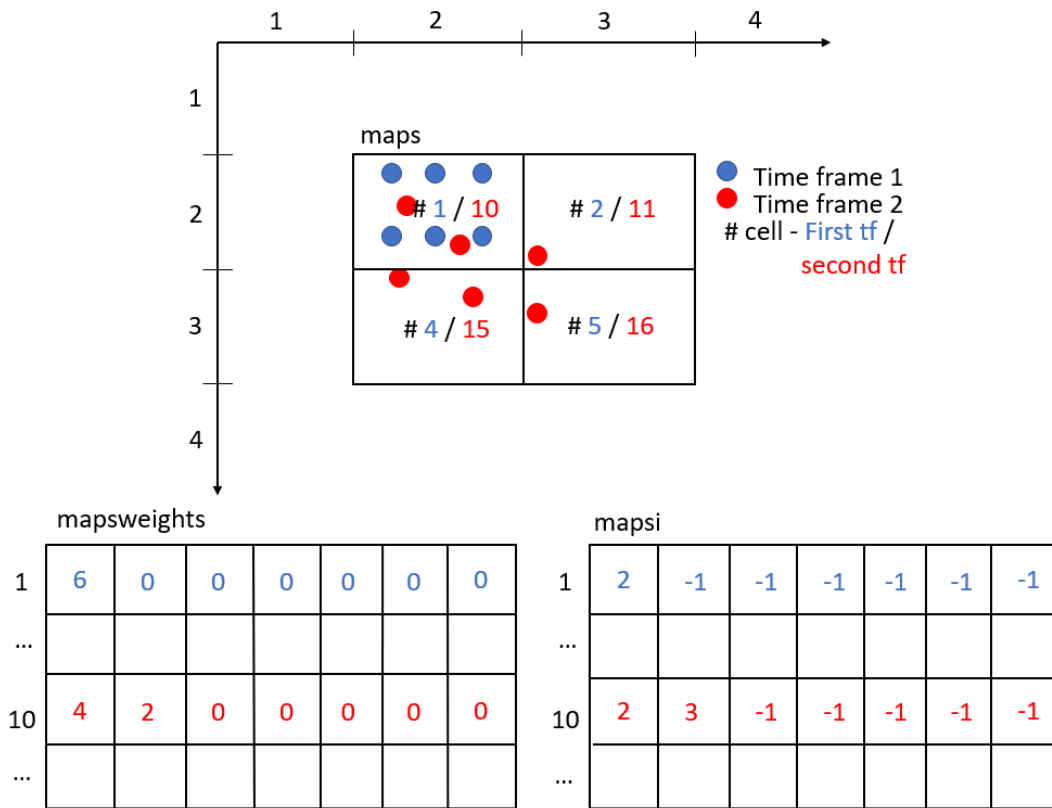


Figure 3.4: Figure showing the maps, maps and mapsweight arrays, together with particles positions at two time frames. Blue represents the initial time frame, and red represents the following time frame. The axes are from the phase space coordinate system, starting at one since Fortran indexing by default starts at one.

In figure 3.4, an example of the the grouping of the particles is shown. Here the movement of the particles from cell number one (blue) is tracked to the next time frame (red). In the corresponding maps array, it can be seen that for the particles from cell number one, all particles are in bin number 2. The mapsweight vector shows that all six of the particles are at this position. In the following time frame, the particles have been tracked through some number of machine turns, and ended up at their new coordinates. Mapsi vector

10, points to the particles from the initial distribution of cell 1, at the second time frame. In figure 3.4, their positions are marked in red. In the corresponding mapsweight and maps, it can be seen that two of the particles are no longer placed in bin 2, but have moved to bin 3. Maps then shows that the particles originating from cell 1 are at x-coordinates 2 and 3. the mapsweight vector shows that from these particles, four particles are in bin 2, and two particles are in bin 3. This sorting process happens for all particles from every cell.

Each particle is tracked from the first to the last time frame. Between the tracking and the sorting, the coordinates of every particle is mapped to and from the different coordinate systems. This process continues until the particles have been tracked through every machine turn.

When the tracking is complete, and the three arrays are filled, the program has knowledge of how many particles ended up in which bins. Also, the weight of each phase space cell at every time frame are known. Now, a fourth array holding the total weight factor of each cell, accumulated from all time frames are created. Finally, everything is set up for the tomographic reconstruction.

3.1.4.4 Tracking Using Self-Fields

In the input file, the user can enable tracking including the self-fields of the bunch. Self-fields are voltage induced by the bunch itself as it travels through the machine. If enabled, the self-fields are calculated shortly after the raw data is loaded and treated. This is done in two steps. First, the profiles are smoothed and differentiated using a Savitzky-Golay filter [18] of 4th order. This has the advantage over conventional low-pass filtering of not increasing the apparent bunch length [19]. The self-field voltages will be added to each particle based on their time and position during the tracking.

3.1.4.5 Extending Maps

During the homogeneous distribution, each cell of the phase space is populated by a number of particles. The particles coming from the same cell, will be tracked together for all the turns. Each vector of the maps and mapsweights arrays points to the particles from a cell at a given time frame. Each element of the vectors points to a bin in the x-axis of the phase space coordinate system at a given time frame.

This means that the maps and mapsweights arrays hold one vector for each cell of each phase space. Too much memory would be required in order to have the vector of the same size as the image width (the number of bins in a profile measurement). For this reason, an assumption is made that the particles starting from the same cell will move closely to one another. Thus, they will end up mostly in the same bins. For this reason, only a few elements representing individual bins per vector are needed. However, if too many particles end up in different bins, the vectors must be extended. This is done using the function "extend_maps" in the tomosubs.v2.f module of the program.

By extending the maps, additional depth is provided to the maps and mapsweights array. The excess data is stored in a supplementary array. The index to this supplementary array is stored in the last element of the vectors of the original arrays. The supplementary array is re-allocatable and this procedure can be repeated. Consequently, the actual depth of the matrix is only limited by the available memory during execution [20].

3.1.4.6 The Tomography Routine

The tomographic reconstruction routine consists of two important steps, back projection (conversion from time projections to phase space) and projection (conversion from phase space to time projections). Figure 3.5 is an overview of the algorithm.

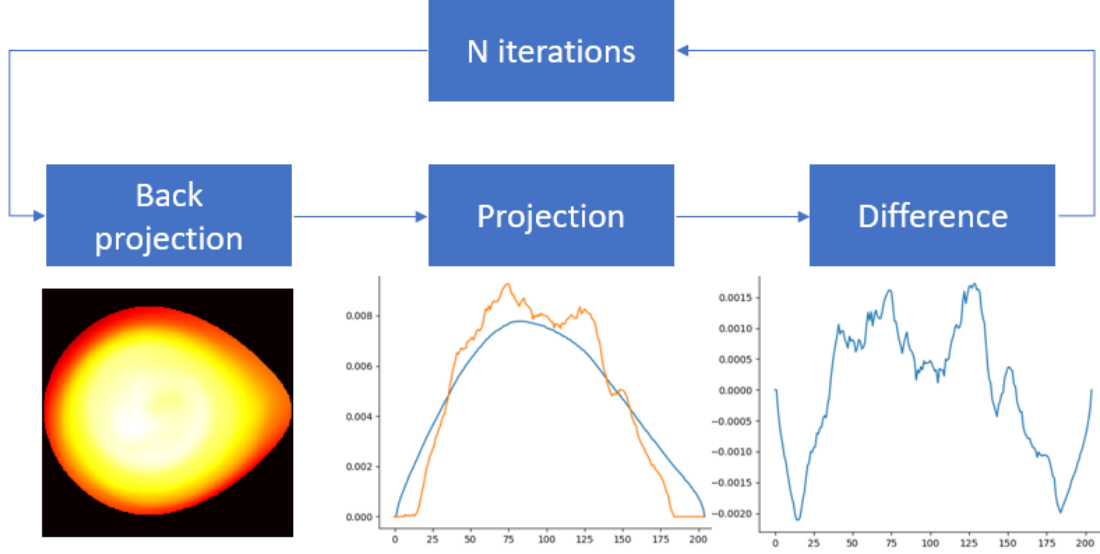


Figure 3.5: The tomography algorithm.

The algorithm starts with a back projection. Here, each cell of each phase space receives a weight. The calculation of the weights are based on the arrays created in the tracking algorithm. The maps array holds an index for each cell (pixel) of the phase space image. The time frame where the cell belongs can be known from the cells index in the maps array. The cells index also points to a vector in the mapsweights and mapspsi array. The mapspsi array stores which bins the particles ended up in, and mapsweights stores how many ended up in each of these bins. A weight for each cell can be determined by multiplying the values from mapsweight, mapspsi and the corresponding values from the profile measurements. After this, the energy distribution of the bunch is estimated.

The recreated phase space should optimally have the same qualities as the bunch. Therefore, the same profiles should be generated from the reconstructed energy distribution as the real bunch. For this reason, the recreated phase space is reduced back to one dimensional profiles using the projection function. This results in a mountain range of reconstructed time projections, which are comparable with the mountain range of measured profiles seen in figure 1.4a.

The recreated and the measured profiles are compared for two reasons. First, the discrepancy can be used as a figure of merit to evaluate the quality of the reconstruction. Secondly, the difference between the profiles can be used to adjust the weight factors. This can be done by using the difference between the profiles as arguments for the back projection routine. Then, weights with a too large value compared to the measured profiles will be negatively adjusted and vice versa. The resulting phase space with the adjusted weight factors will generate new recreated profiles. The process can be repeated as many times as needed for the phase space image to converge towards a usable quality. The reversedweights array is used to ensure that the effect of the difference adjustment are equal for all cells of the phase space regardless of the number of particles in the bins.

After N iterations, where N is specified by the user, the weights ready to be presented. At last, the final image is saved to a text file containing a one-dimensional array of numbers, resembling the weights of the pixels in the image.

3.2 Python Translation

This chapter will focus on the changes between the Python/C++ and the Fortran code. First, goals of the translation will be elaborated, then the algorithmic and structural changes are described. Finally, some opportunities and effects of the translation are investigated. Finally, results from experiments using GPU acceleration are presented.

3.2.1 Goals for the Translation

The original program is written in large blocks of heavily optimised Fortran code. The algorithm consists of many nested loops and complex array indexing which makes the program flow hard to follow. For this reason, an important goal is to make the program more understandable and user friendly. This is done by making the program as modular as possible, meaning that it is divided into atomic, independent and interchangeable modules. By distinctly separating functionality to such modules, the program flow will be easier to grasp for future developers. An implementation is made by forming a library which will make the user able to import and use only parts of the tomography program. In this way, the user can experiment with combining their own specialised modules with the original algorithm. This makes the program highly flexible, but also rises demands for a well tested and documented code.

More demands and specifications are set for the program in chapter 2.1. Among these, requirements for the run time are set. The original program is fast, and the translation should not be slower. It is therefore important that the program obtains the shortest run time possible. A short run time will be especially important if the program is to be used for online tomography, creating phase space reconstructions in real time. In this case, the run time must be below the long term goal of less than 0.5 seconds. In an attempt to reach this goal, an experimental implementation of a GPU accelerated tomography routine is set up and evaluated.

To summarise, the goal for the translation is to make a modern, fast, and flexible tomography library with the possibility for hardware acceleration being investigated.

3.2.2 First Translation - Program Flow and Structure

The program was initially translated directly from Fortran to Python. In this version, the program was a one-to-one translation of all functions. In this way, it was easy to assert that the replicated program was correct, as the values could be directly compared. However, the program was split into classes and functions in a more object-oriented manner, where the classes represented the different stages of the algorithm: loading of parameters, loading of profiles, generating reconstruction area, particle tracking and phase space reconstruction. Resulting in the classes being named Parameters, TimeSpace, MapInfo, ParticleTracker and Tomography. See figure 3.6.

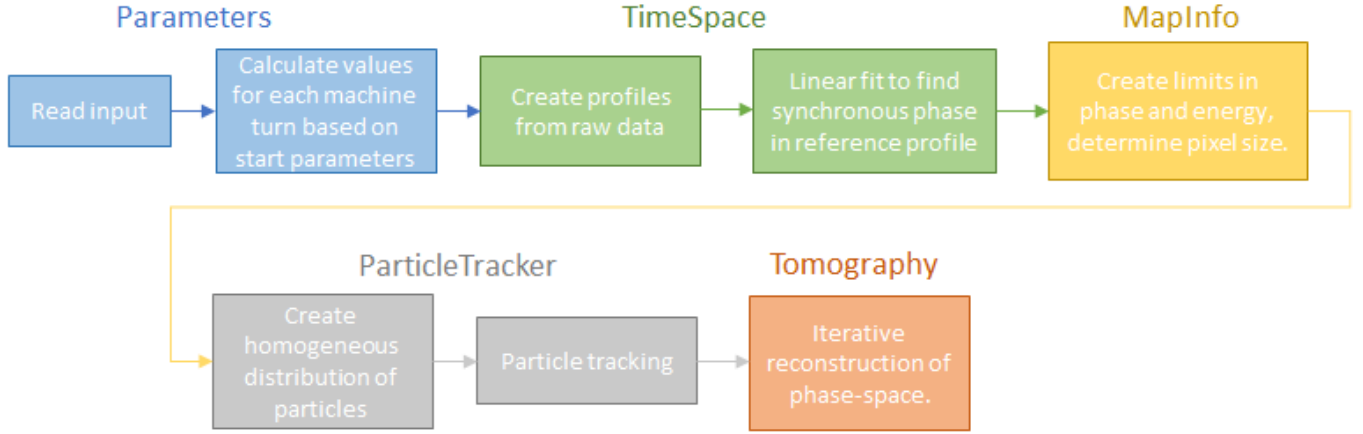


Figure 3.6: First translation - Program Flow and Classes.

The Parameters class retrieves the input parameters from an input file (see appendix B). Many of these parameters are used as initial values when the corresponding values at each machine turn is to be calculated for usage during the particle tracking.

The TimeSpace is created from one Parameter object. Profiles are created by using settings found in the Parameters object. The beam reference profile can, if desired by the operator, be used to estimate the synchronous phase. Also, this class calculates the charge of the beam reference profile and, if desired by the operator, calculates the self-fields of the bunch (see chapter 3.1.4.4).

MapInfo is a class needed for calculating and storing information about the phase space to be reconstructed. Here, the limits of the reconstruction area, and the energy size of the bins of phase space is calculated. The Mapinfo object is created with the information provided by a TimeSpace object.

A ParticleTracker object holds functions for tracking the particles. Such objects are created from a TimeSpace object holding the machine parameters for each turn, together with a MapInfo object holding the area of which the particles should be distributed. Also, the MapInfo object is needed for the conversion from physical units to phase space coordinates. The track function of the ParticleTracker class returns the arrays described in chapter 3.1.4.3.

The tomographic reconstruction happens in the Tomography class. Here, the reconstruction routine will return the reconstructed phase space image. Also, Information, like the discrepancy at each turn of the reconstruction and the reconstructed profiles can be accessed via the fields of the Reconstruction object.

3.2.3 Algorithmic Changes

After the direct translation, large changes was introduced to the reconstruction algorithm (see chapter 3.1 for an explanation of the old algorithm). These changes can be seen both in the particle tracking and the tomographic reconstruction routine, and have resulted in a more understandable and parallelizable program. This means that the program can easier profit from the attributes of modern CPUs.

3.2.3.1 Changes in the Tracking Algorithm

While the original particle tracking is performed in a track-and-sort manner, the new algorithm performs the tracking in one go. The sorting process for the earlier mentioned arrays maps, maps_i and maps_weight are not any longer performed.

As before, the tracking will start from a particle distribution initiated at the turn of the profile to be reconstructed. The distributed particles are tracked using kick and drift (see figure 1.2) routines based on

the routines developed for BLoND. A function named, creatively, `kick_and_drift` tracks the particles through all machine turns. From the initial turn, the particles' movement will be tracked in backward and forward direction until the first and last machine turn. As opposed to the original algorithm, the particle coordinates are not treated in any way after the tracking from one time frame to the next. Instead, the position of each particle at each time frame will be saved individually. The conversion from physical to phase space coordinates will not happen before the tracking is completed.

Because of the algorithmic changes, new possibilities are introduced. First, since the particles depend only on their own preceding position in phase and energy, the particle tracking can now be parallelized based on each individual particle. On the contrary, the old algorithm depended on the position of every particle in a cell to create the weights of that cell. Secondly, the earlier mentioned usage of existing BLoND routines for tracking is beneficial as this introduces a coherency in the software developed by the BE-RF-BR section.

3.2.3.2 Changes in the Reconstruction Algorithm

Based on the changes in the particle tracking, further changes were implemented in the reconstruction algorithm. Since the particles are now tracked and stored as single particles, a weight factor is given to each of them individually.

The tomographic reconstruction routine mainly consists of two stages: back projection and projection. The back projection converts from time projections to phase space distributions. The projection routine converts from phase space distributions to time projections, see figure 3.5.

In the back projection routine, the weights of the particles distributed or adjusted. All particles have a x -coordinate, which is a bin in the phase axis of the phase space coordinate system. This coordinate also corresponds to a time in a profile measurement. The weight of the particle will initially be its accumulated value from its position through the profile measurements. In this way, the particle weighting is similar to the one used in the original algorithm, but in the new algorithm each particle is weighted individually instead as cells of the grid structure in the old algorithm.

The projection routine is the reverse of the back projection. Here, the weight of each particle is added to each bin the particle visits during its movement. The result will be a histogram for each time frame of the weights in each bin. These histograms form the recreated time projections of the phase space and will be referred to as the reconstructed profiles. Examples of such profiles can be seen in figure 3.7. Here, the measured, and the reconstructed profile can be seen in the top graph. Another reconstructed profile can be seen at the right plot, projected along the other axis of the phase space.

The reconstructed profiles are normalised, and the negative parts are set to zeroes. They are now directly comparable to the measured profiles. In figure 3.8, the measured and the reconstructed profiles are presented as waterfalls. The difference between them are calculated and stored in a third array, which will be referred to as the difference profiles. These are used to calculate the discrepancy, which is a measurement of the quality of the reconstruction. The discrepancy between the reconstructed and the measured profiles is calculated using equation 3.1. This gives the bin-by-bin discrepancy for the recreated profiles. Here, Δp is the difference between a measured and a reconstructed profile in one bin, l_p is the length of the profiles in number of bins and n_p is the number of profiles. This is the same formula used to calculate the discrepancy of the original algorithm.

$$discrepancy = \sqrt{\frac{\sum \Delta p^2}{l_p n_p}} \quad (3.1)$$

When the particles have received their initial weight, the reconstructed phase space will be of a rather poor quality. For this reason, the weights are adjusted by providing the back projection routine with the difference profiles. In cases where the bins of the reconstructed profiles have a too low intensity, the difference will be positive, thereby resulting in a weight adjusted upwards for the particles in that bin. The opposite

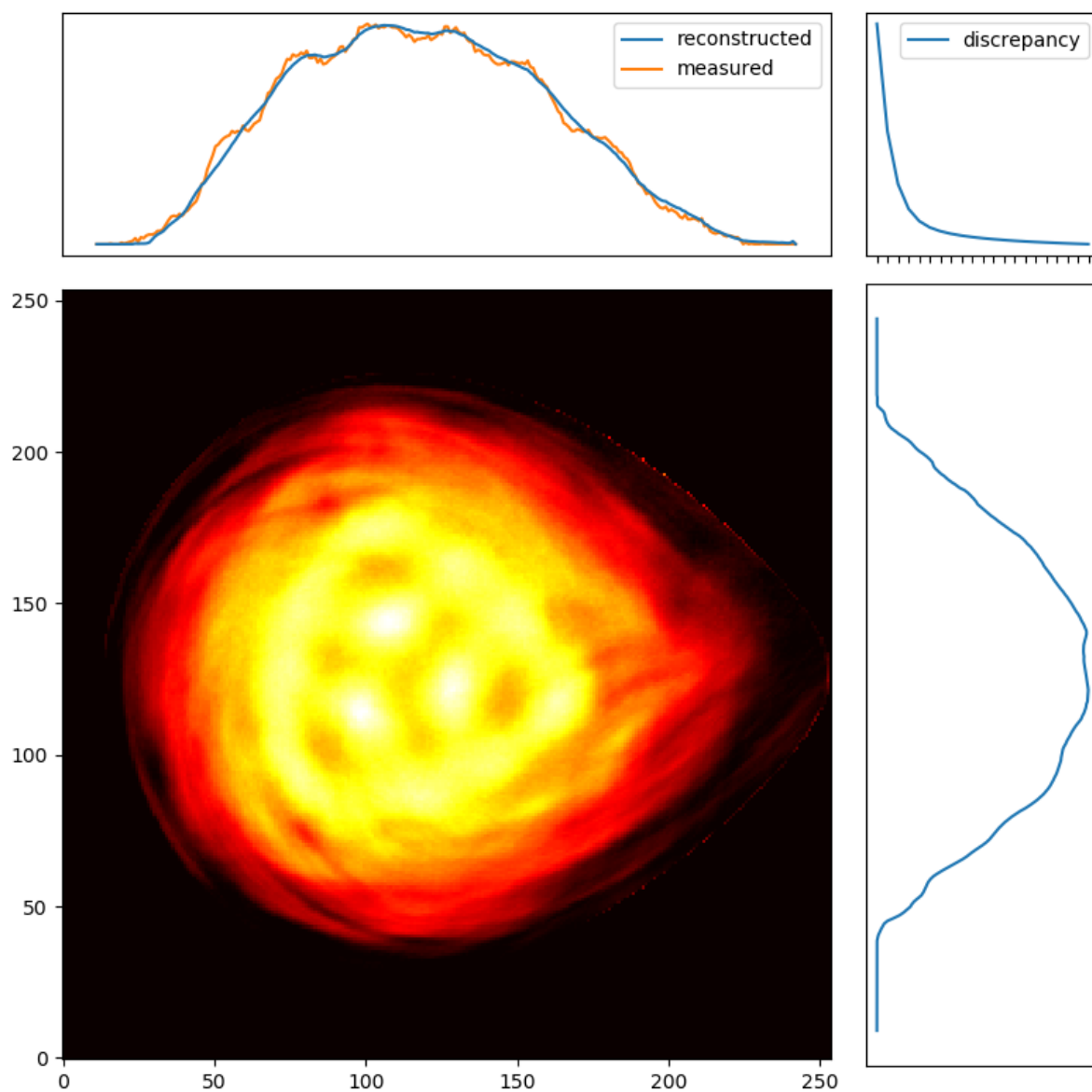


Figure 3.7: Figure showing a reconstructed phase space, together with its recreated profiles (blue) and the measured profiles (orange). The discrepancy for each iteration is shown in the top right corner.

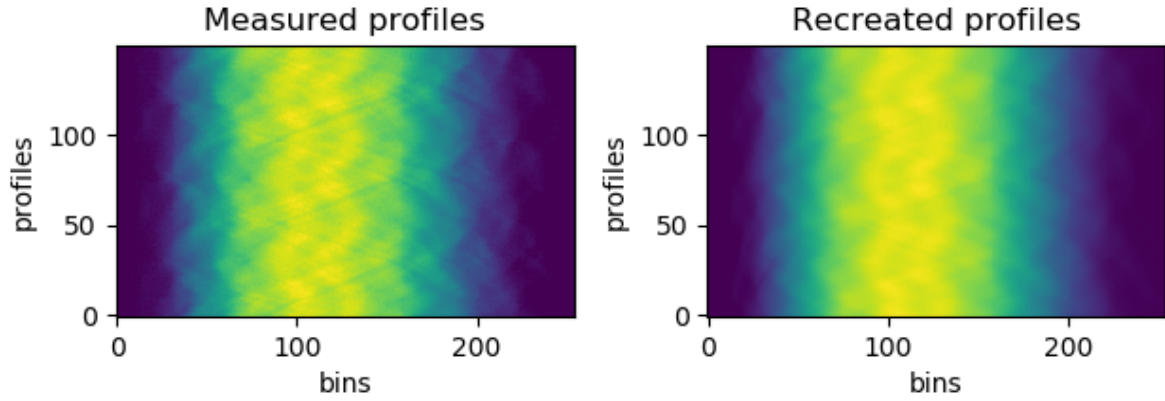


Figure 3.8: Measured and recreated profiles presented as waterfalls. All values are normalised, and the colour of the image represents the energy at each bin on a scale from 0 (dark) to 1 (bright).

happens for bins where the reconstructed profiles have a too large intensity. Here the weights will be adjusted with a negative number. The adjustment will continue for N iterations, where N is specified by the user. At each iteration, the recreated phase space has a slight smaller discrepancy than the prior. Resulting in a phase space of a reasonable quality.

In order to obtain a faster convergence, adjustments are made to the difference profiles before they are passed on to the back projection routines. The implementation of the original method has been altered to fit the new algorithm. When the particles gain their initial weight, the weight factors are given directly by the measured profiles. In the following iterations of the reconstruction, the weights are adjusted using the weights from the difference profiles. These are significantly smaller in magnitude than the measured profiles, and will contribute with a smaller impact on the weight of the particles. When the weights have been adjusted, the difference profiles will be smaller, leading to adjustment to the weight factors also being smaller. The small impact of each weight creates a problem as the number of particles in each bin is not equally distributed. A much smaller amount of particles can be found in the tail of the bunch, relative to the centre of the bunch. This means that the magnitude of the difference profile creates a larger contribution of weight adjustment in the centre than in the tails of the bunch. In order to compensate for this, each bin of the difference profiles are multiplied by the reciprocal number of particles of that bin, relative to the bin holding the most particles. The effects of the reciprocal multiplication can be seen in figure 3.9. Here, the blue line is the original difference profile, while the orange line is the same difference profile after adjustments using reciprocal multiplication. Notice that the compensation is much larger in the tails than in the centre the bunch. By using this method, the impact of the difference profile is equal for all bins independently of the number of particles in that bin. This method equalises the speed of convergence for the whole image.

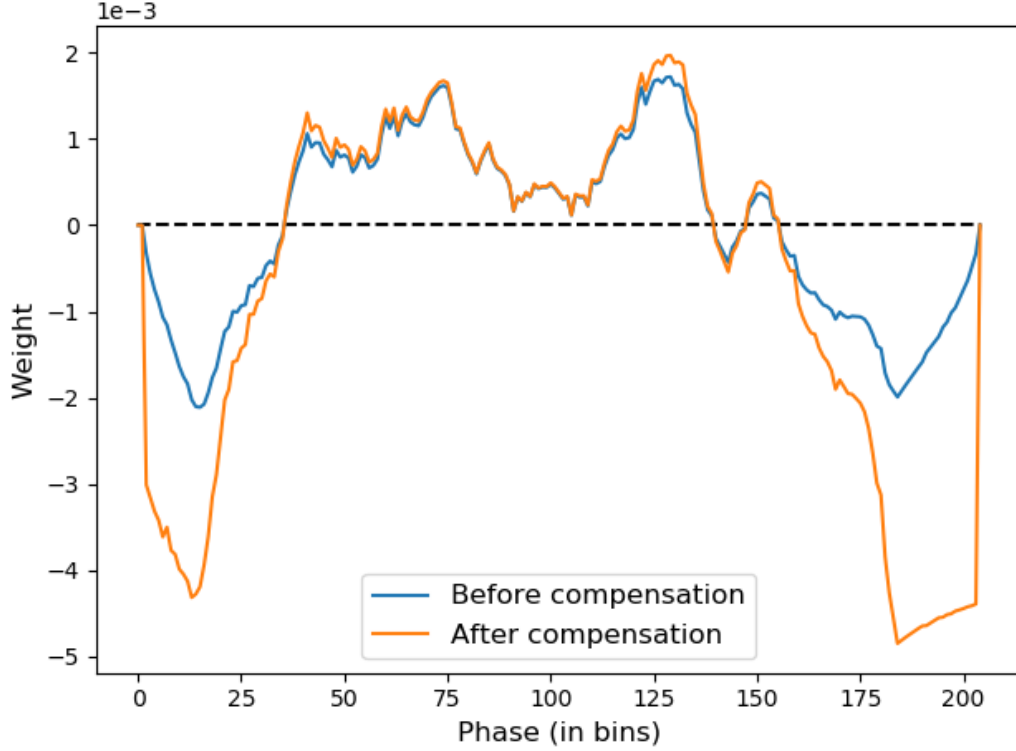


Figure 3.9: Difference profiles before (blue) and after (orange) reciprocal multiplication.

By combining the coordinates of the tracked particles with the weights found in the reconstruction routine, a phase space image of identical format as the original output can be created. This means that applications already using the original program can effortlessly change to use the new Python/C++ code. Also, the original and the new code can easily be compared one-to-one.

3.2.4 Restructuring from Program to Library

In order to obtain an optimally modular program, the fully translated program was restructured. Here, the goal was to make the stages of the tomography as independent from each other as possible. The restructuring resulted in a library for longitudinal beam tomography, rather than a monolithic architecture. This gives a drastic increase of the codes flexibility, and opens for highly customised tomography programs.

3.2.4.1 Library Structure

During the reconstruction, it was important to obtain a deep understanding of which data was needed for each part of the program. This resulted in an improved code structure which had a better compliance with the programs data flow.

An effort was made to make the modules and classes as independent from each other as possible. It was observed that the machine parameters are needed only for the tracking of the particles. meanwhile, the profile measurements are needed mostly for the tomographic reconstruction. However, Some corner cases combining the need of both the machine parameters and the measured profiles are found. These includes tracking using self-fields, and routines to estimate the position of the synchronous particle. Nevertheless, the classes and modules could be sorted into packages depending on the functionality and data required.

This led to the library being divided into five packages in order to create the most logic interface for the user. Table 3.3 shows the packages and their associated modules. The library is structured in a way such that each package represents one stage of the tomography routine. Furthermore, the layout of the classes can be seen in figure 3.10. Here, an overview can be seen of which classes are dependant on each others.

Package Name	Modules
cpp_routines	c++ souce code tomolib_wrappers.py
data	profiles.py
tomography	__tomography.py tomography.py
tracking	machine.py __tracking.py tracking.py particles.py phase_space_info.py
utils	assertions.py exceptions.py data_treatment.py physics.py tomo_input.py tomo_output.py

Table 3.3: Packages and modules of the tomography library.

The `cpp_routines` package holds all the functionality written in C++. This module should only be accessed by advanced users, and is needed internally for the other modules. The Python module `tomolib_wrappers.py` holds wrapper functions used to access the C++ routines from Python. This is done using the `ctypes` library, and loading the C++ routines from a shared object. By writing wrapper functions like these, the C++ routines can easily be tested using Python based unit tests, like `pytest`.

The package named `data` holds the `profiles` module, containing the `Profiles` class. This class is not strictly necessary for the tomography routine, but aids the user in storing all the relevant information concerning the measured data. However, the `profiles` class is needed if self fields are to be used during particle tracking.

The `tracking` package contains all needed modules for the particle tracking. Here, the modules `machine`, `particles`, `phase_space_info`, `tracking` and `__tracking` can be found. The `machine` module holds the `Machine` class. This stores the input parameters, and calculates the machine parameters for each turn. Meanwhile, the `phase_space_info` module holds the `PhaseSpaceInfo` class, which calculates parameters for the phase space to be reconstructed based on a machine object. This module is needed by the `Particles` class in the `particles` module, in order to generate the particle distribution to be tracked. The `__tracking` module holds the `ParticleTracker`, a super class for particle tracking classes. These can be found in the `tracking` module. Common for all these modules is that they are all dependant only on the input parameters stored in a machine object, and can be run without the need of measured profiles.

In the `tomography` package, the `__tomography` module holds a super class for the `Tomography` class. This holds assertions and helpful routines, common for all tomography classes. Another module named `tomography` holds the operational class for performing tomography. Here, the user can choose between running the tomography in C++, or to run the tomography orchestrated by Python. In this case, a Python function calls the projection and back projection routine from the C++ library, performing data treatment and assertions on the weights and recreated profiles in Python. This makes it easier for the users to define

their own tomography routines. Whereas the tomography class depends only on the particles coordinates and the measured profiles, the tomography package can be used totally independent from the rest of the library. On the contrary, the original tomography routine was heavily dependant on the format of the tracking routine's output. This is one of the major perks of the new algorithm, and provides great freedom for the user when it comes to particle tracking and data treatment.

The utils package holds modules needed as support by other modules. Also, the package holds modules for handling input and output of the original format. Here, functionality can be found to easily integrate the Python/C++ solution with the operational tomoscope application, the operational GUI for tomography. Modules for handling input and output is called `tomo_input` and `tomo_output`. Assertions and exceptions are modules for handling errors in the modules of the tomography library. The `data_treatment` module holds functions for processing measured data and generated data. Also, it holds functions for performing estimation of the synchronous phase. Furthermore, the physics module holds physics formulas, where most of these accept a machine object as an argument for easy usage. It is likely that this module will be extended and refined in the future.

Together, the modules of the library can aid the user in quickly creating customised tomography programs.

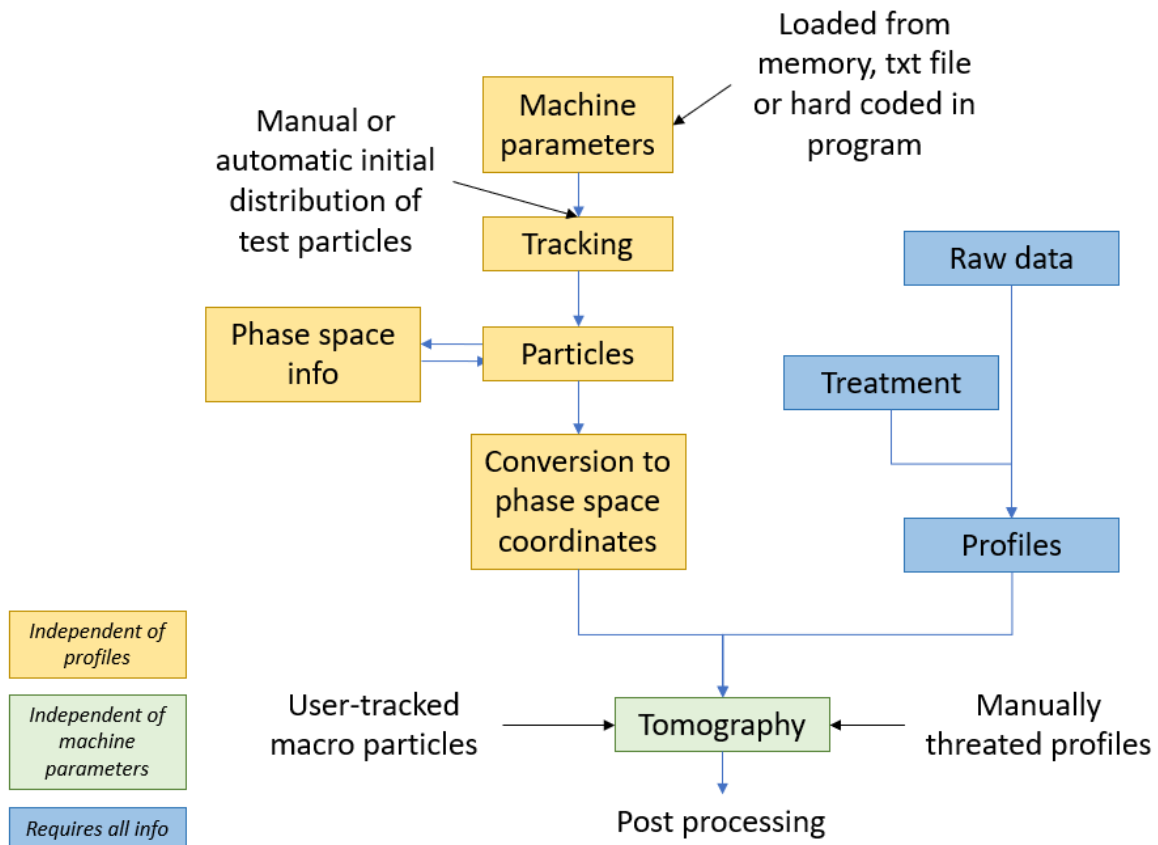


Figure 3.10: Restructured program. Black text represents optional user choices.

3.2.4.2 IO Opportunities

In appendix B an example of an input file for the Fortran version is shown. When using the original program, creating such files is the only way of changing the reconstruction parameters. However, restructuring from

program to library allows for new ways of providing input for the user. An example of this is initialising machine objects by providing a dictionary of values. Now, only the programmers imagination sets the limit for how to retrieve the needed information for the reconstruction.

The program output can also be saved in various ways. Functions of the tomography library aid the programmer to generate output of identical format as the original program. However, the programmer can also profit from the opportunities offered by the Python language and its numerous libraries.

3.2.4.3 Online and Offline Tomography

The modular library creates possibilities for an on-line tomography. This raise large demands for the run time of the program. Since the library allows for the tomography routines to be run independently from the rest of the algorithm, a minimal version of the tomography program can be made. The functions needed for such a version are all written in optimised C++, parallelized using OpenMP. These can be run very quickly utilising the parallel processing power of modern CPUs. This minimal version can, in stead of tracking the particles for each reconstruction, load them from a database of already tracked particles. This is possible since the particles trajectories are dependant only on the machine parameters, and not the profile measurements. The machine parameters are known at the operator at forehand, and the tomography program can load the particles on demand to be combined with the profile measurements for tomographic reconstruction. Such a solution is investigated and evaluated in chapter 4.3.

For off-line tomography, the run time is not as critical. Here, the accuracy of the reconstructions and the flexibility of the program is more describable. As the modules of the library works independently of each other, the programmer do not need to use the full reconstruction algorithm. Because of this, the programmer can create custom tomography programs which covers individual needs.

3.2.5 Floating Point Error

During the development of the Python/C++ program a mysterious "bug" was found. The phase space generated in the Python/C++ version resulted in a bleach phase space, with its weight being distributed incorrectly relative to the reconstruction area of the phase space. After some time, the source of the alleged bug turned out not to be an error in the code, but an effect of the different sizes of floating points used in the two versions. While the Fortran program mainly uses floats with 32 bits precision (f32), the Python/C++ implementation uses 64 bit doubles (f64). This leads to a subtle difference in the reconstructions, however in some cases the difference can have a larger magnitude than others.

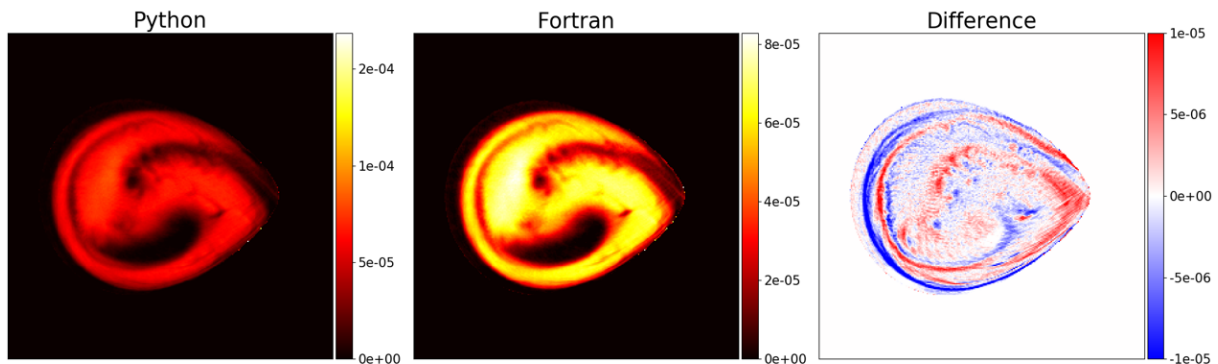


Figure 3.11: Difference in reconstructions due to floating point error running with f64.

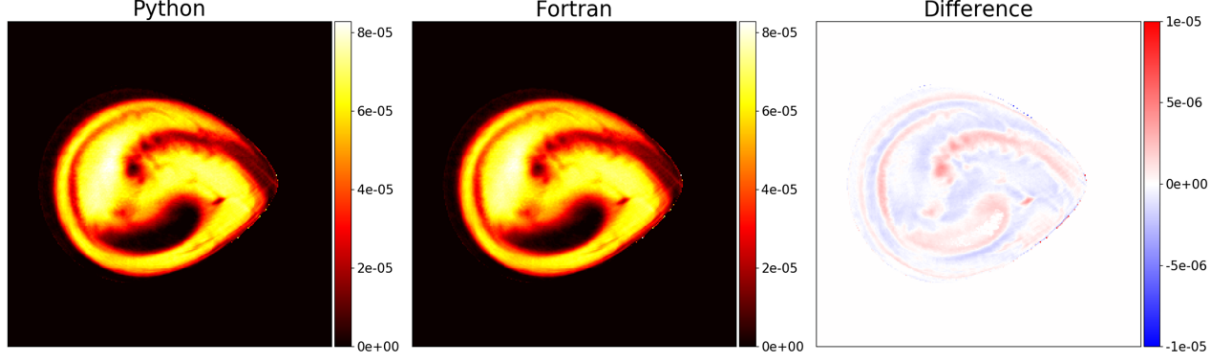


Figure 3.12: Reconstructed phase space using Fortran and Python/C++ with f32.

Such a difference can be seen in figure 3.11. The leftmost image is the reconstruction based on the Python/C++ program. The center image shows the phase space recreated by the Fortran program, and the rightmost plot shows the difference between the Fortran and Python/C++ output. In the Python/C++ image, too much weight is given to the particles in the lower right part of the phase space. This makes the rest of the phase space look bleak relative to the small intense over-weighted areas.

The difference comes from an floating point difference in an array holding the energy of the synchronous particle at the end of each turn $E0$. Its initial value is calculated in a function named B_to_E in the physics module. The energy for the remaining turns are then added to the initial energy, and stored turn by turn in the $E0$ array. From the calculations resulting in figure 3.11, the initial energy of the synchronous particle when calculated with f64 is approximately $1.036792088E9$. When calculated using f32 the same variable has the value of approximately $1.036792064E9$. This makes a difference of approximately 24 eV. The corresponding values for the first three turns can be seen in table 3.4.

Float32	Float64	F32 - F64
$1.036\,792\,064\,000\,000\,000 \times 10^9$	$1.036\,792\,088\,210\,674\,644 \times 10^9$	$-24.210\,674\,643\,516\,54$
$1.036\,793\,280\,000\,000\,000 \times 10^9$	$1.036\,793\,273\,308\,957\,219 \times 10^9$	$6.691\,042\,780\,876\,16$
$1.036\,794\,496\,000\,000\,000 \times 10^9$	$1.036\,794\,458\,407\,239\,795 \times 10^9$	$37.592\,760\,205\,268\,86$

Table 3.4: Energy [eV] of the synchronous particle for the first three turns.

Float32	Float64	F32 - F64
0.000 000 000 000 000 000	0.000 000 000 000 000 000	0.000 000 000 000 0
$1.216\,000\,000\,000\,000\,000 \times 10^3$	$1.185\,098\,282\,575\,607\,300 \times 10^3$	$30.901\,717\,424\,392\,7$
$1.216\,000\,000\,000\,000\,000 \times 10^3$	$1.185\,098\,282\,575\,607\,300 \times 10^3$	$30.901\,717\,424\,392\,7$

Table 3.5: Turn-by-turn difference in energy [eV] of the synchronous particle for the first three turns.

Float32	Float64	F32 - F64
0.450 641 900 300 979 6	0.450 641 889 543 126 3	$1.075\,785\,333\,037\,870\,8 \times 10^{-8}$

Table 3.6: Difference in synchronous phase [rad].

The turn-by-turn energy difference can be seen in table 3.6. The difference between the f32 and the f64 version differs by approximately 31 eV/turn. As this might seem like a small difference, it is enough to make an impact on the particles trajectories. Since the particles in this case are tracked through 990 turns, the accumulated difference is 30.69 KeV.

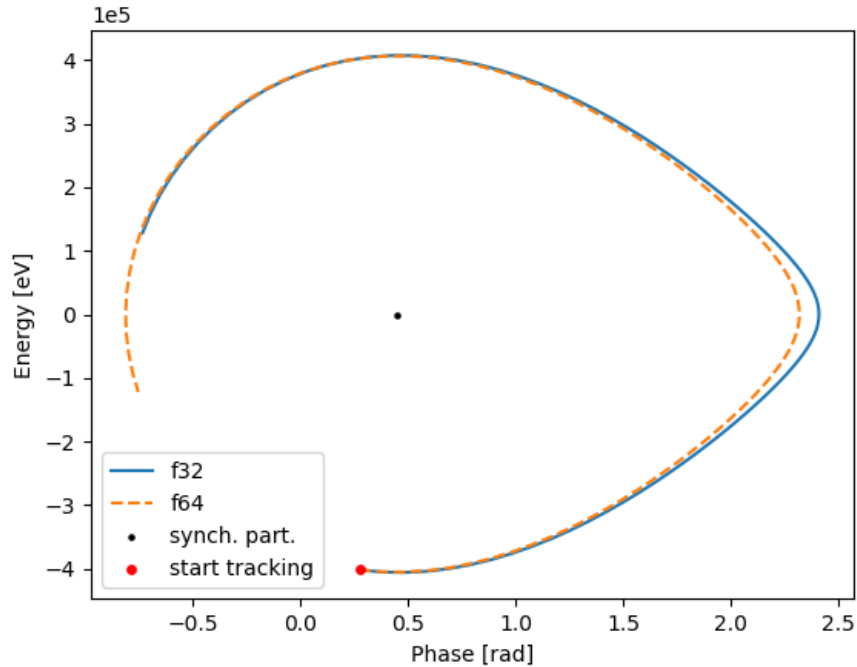


Figure 3.13: The trajectory of an particle at varying floating point precision.

In figure 3.13, the difference of a particles trajectory after running the program using f64 (orange) and f32 (blue) can be seen. Here, the program is run as a whole, varying the size of the floats describing the machine parameters with the sizes f64 and f32. By setting the variables size in the Python/C++ program to f32, the final result matches the Fortran output. This shows that some caution must be taken when comparing the Python/C++ output with the Fortran output.

3.3 Hardware Acceleration

This section describes the experimental implementation of a HW accelerated longitudinal beam tomography algorithm. Here the implementation is described, and its potential is discussed after analysing tests of run time reduction. Notice that this implementation is not meant for operational usage, but have the purpose to show a proof of principle to be investigated in the future.

3.3.1 A Quick Introduction to GPUs

In the later years, the ability to make serial programs faster has stagnated. An example of this is how the physical limits of silicon have started to meet their upper boundary on how much one can raise the CPU clock frequency. In order to make programs run faster new ways have to be found. A typical approach is to run a program using multiple cores in parallel, which, if done right, can make the program run very quickly.

Writing parallel programs for CPUs also let them be accelerated by for example the usage of Graphical Processing Units (GPUs). This can speed up the program even further than running on CPU alone.

GPUs are often used in High performance Computing (HPC) for calculating computationally heavy tasks. While a CPU can run a sequential program very effectively with low latency and a few strong cores, a GPU is optimized for parallel tasks. They consist of many cores and often have an effective internal high bandwidth memory, which can result in quick programs with high throughput.

The original purpose for GPUs were graphics processing. Here, simple mathematical operations were performed on a large amount on individual pixels. Therefore, they cannot not handle functions requiring decision handling as efficient as CPUs. For this reason, changes must be made to the algorithms in order to make it optimal for GPU acceleration.

On the other hand, GPUs also have their drawbacks. From an economic standpoint, GPUs are expensive and take a lot of energy to run. In addition to this, they introduce some technical difficulties. First, they are not straight forward to use, especially when the goal is to retrieve the maximum benefit from them. A range of tools are developed for writing parallel programs, but many of them are advanced and have a steep learning curve. Other tools which are easier to use often produces a lower gain in terms of speedup and efficiency. This comes from less ability to fine tune the GPU functions (kernels) and introduction of large overheads. Secondly, the time needed to transfer from CPU to GPU memory is notable. For this reason, an evaluation must be done of the ratio between the size of the data and the number of calculations to be performed on the data. Having a too large data set in relation to the number of calculations may result in spending more time transferring memory than actually performing calculations.

3.3.2 Tools Used

The device used in this project was the Nvidia GeForce GTX 1080. It has 2560 CUDA cores, which are the parallel processors of the GPU. It was run using CUDA 10.1. The programming tool used is OpenACC.

OpenACC is a compiler directive which is created as a collaboration between Nvidia, Cray, the Portland Group (PGI) and CAPS enterprise. It is based on the compiler directive OpenMP which is already used in the program for parallelizing on the CPU. OpenACC have the goal creating an easy to use parallel programming standard for scientists and programmers [21]. OpenACC works with compilers for C, C++ and Fortran.

OpenACC was chosen for several reasons. It has an easy-to-use programming interface, where parallelizable loops can be decorated by a pragma which tells the compiler to automatically create a parallel version of the loop. In addition to this, you have the possibility to tune your solution using a wide range of compiler instructions like flags and clauses. This gives an opportunity to easily produce a test for the performance gain when adding HW acceleration to the program.

3.3.3 Accelerating the Particle Tracking

The particle tracking consists mainly of two functions. These are called kick and drift. Kick gives a change in energy to each particle based on the voltage received by the particle as it passes the RF station (see figure 1.2). Drift is the phase drift of the particles as it is sent around the ring.

Code 1: The drift function as written in C++:

```
#pragma acc parallel loop device_type(nvidia) vector_length(32)
for (int i = 0; i < nr_particles; i++)
    dphi[i] -= dphase * denenergy[i];
```

Drift is a linear function calculating the change in phase between two machine turns. The calculation is based on the energy of a particle relative to the synchronous particle at a turn together with a coefficient

describing the phase drift. Such a calculation is done independently for every particle. In this way, the phase of every particle depends only on its own values at the preceding turn. This makes the drift function suitable for parallelizing based on particles.

Code 2: The kick function as written in C++:

```
#pragma acc parallel loop device_type(nvidia) vector_length(32)
for (int i=0; i < nr_particles; i++)
    denenergy[i] = denenergy[i] + rfv1 * sin(dphi[i] + phi0)
                + rfv2 * sin(hratio * (dphi[i] + phi0 - phi12))
                - acc_kick;
```

Kick is a nonlinear function which calculates the difference in energy for each particle between two machine turns. Since the kick routine use non linear mathematical functions, it is slower than the drift routine. As for the drift routine, the kick routine depends only on the preceding values of a particle in order to calculate the next values.

By adding the ‘parallel’ construct together with the ‘loop’ construct as seen in code 2, the compiler is instructed to create a parallel version of the loop to be run at the GPU. By combining the clauses ‘device_type’ with ‘vector_length’, the compiler is instructed to set the vector length to a given value if the connected device is of the type Nvidia.

3.3.3.1 Performance Increase for GPU Accelerated Particle Tracking

Three versions of the tracking routine were tested. One version was implemented as a hybrid between Python and C++, where the Kick and Drift implemented in C++ were called repeatedly from Python. The other versions was written fully in C++, called from Python only once. This implementation was made parallel using OpenMP for the CPU and OpenACC for the GPU.

The C++ code was compiled as a shared object and called from Python using the ctypes library. For the CPU versions, the compiler used was g++ (version 4.8.5) and the compiler flags: omp, O3, march=native and ffast-math. For the GPU version, the compiler was pgc++ (19.4-0) with compiler flags: acc and ta=nvidia. C++11 was used for all the versions.

A benchmark was created by tracking 406272 particles for 1088 turns in order to determine the speedup. The kick and the drift routines were called for every machine turn. The physical units of the particles are mapped to coordinates in the phase space reference system. This happens at every machine turn where a measurement has been taken. In this benchmark, the mapping happened at 99 of the 1088 machine turns.

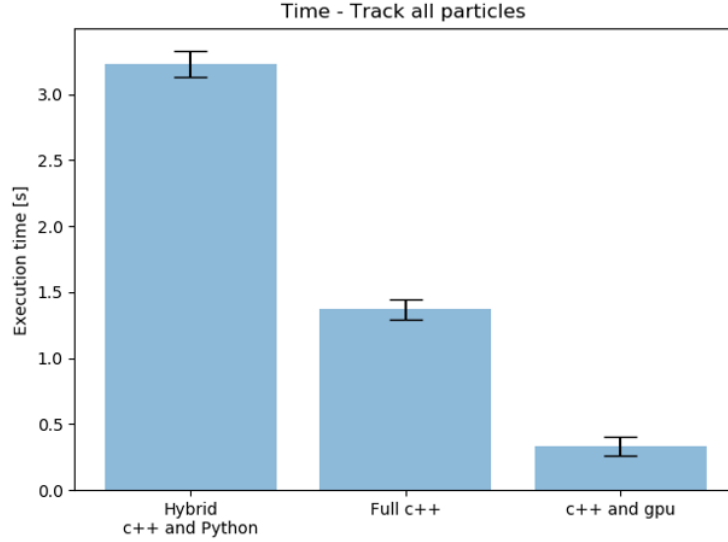


Figure 3.14: Run time for complete tracking. To the left: Python calling kick and drift as C++ routines, run on CPU. Centre: Full C++ implementation of tracking, run on CPU. Right: Full C++ implementation, accelerated with GPU. Error bars show standard deviation based on the values from 50 runs.

The test results can be seen in figure 3.14. The left and centre graph was run only using the CPU. The left graph shows the run time for the Python/C++ hybrid version. The centre graph shows the run time for the full C++ implementation running at the CPU. The rightmost graph shows the run time for full C++ implementation, accelerated using the GPU. These measurements show that there is a lot of time to be saved by running the particle tracking on a GPU. For this benchmark it was shown to be over a 5x speed-up between the Python/C++ hybrid and the GPU solution. It was also an approximate speed up of 3x when running the full C++ implementation on the GPU instead of the CPU. Further investigations have been done regarding the efficiency of CPU vs GPU by testing the run time of the routine when varying the number of particles. The following graphs are made by measuring the run time of the kick and the drift function separately.

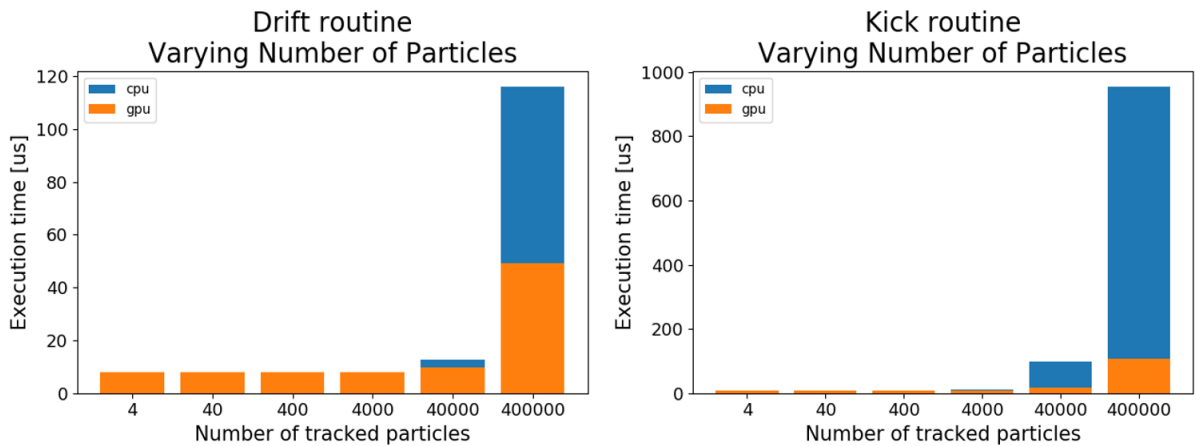


Figure 3.15: Run time for a varied number of tracked particles. Left: kick routine. Right: drift routine

Figure 3.15 shows the relationship between the number of tracked particles and the run time for the kick

and the drift routine. The left graph shows the measured times for the drift function as the number of test particles are varied. Here we can see that the CPU implementation is faster than the GPU implementation when the number of tracked particles are under 40,000. As for the kick function, the number of tracked particles needed for the GPU to be worthwhile is only approximately 4,000. Note that the run time for the drift routine is much smaller than the kick routine. This is because of a more complicated calculation which also includes nonlinear functions. Another feature seen at the plot is that the GPU implementation scales much better than the CPU version as the number of tracked particles increases.

3.3.4 Accelerating the Reconstruction Routine.

3.3.4.1 Back Projection

The two main components of the reconstruction routine are the projection and the back projection routine. The back projection routine converts from time projections to phase space. The projection routine converts in the other direction, from phase space to time projections.

Code 3: The 'flat back projection routine':

```
#pragma acc parallel loop private(sum_particle) device_type(nvidia) vector_length(32))
  for (int i = 0; i < npart; i++)
    for (int j = 0; j < nprof; j++)
      weights[i] += flat_profiles[flat_points[i][j]];
```

The CPU version of the reconstruction algorithm routine uses flattened arrays for storing the data of the measured profiles, this is shown in code 3. In the original array, the measured profile data is a two-dimensional array containing the bunch intensity in for each bin in each profile. In the flattened version, this array is converted to a one-dimensional array where each bin has its original index multiplied with the profile length. In order to make this work, the array containing the x-coordinates of the particles must also be adjusted. The x-coordinates are then added by the multiple of the profile it is situated at and the length of the profiles in bins. This has shown to be a more effective solution than to use two-dimensional arrays. However, this was not optimal for running on a GPU. To get around this problem, an implementation using two-dimensional arrays was created, the following code snippet.

Code 4: The back projection routine working on the waterfall as a 2D array:

```
for (int i = 0; i < npart; i++){
  sum_particle = 0;
  for (int j = 0; j < nprof; j++)
    sum_particle += profiles[j][xp[i][j]];
  weights[i] += sum_particle;
}
```

The two-dimensional back projection routine consists of two loops. The outer loop goes through each of the test particles. This is the loop which is made parallel. By marking the `sum_private` variable as `private`, a local copy gets created in each thread. The total weight of the particle is then calculated by summing the measured intensity at the particles x-coordinate for every profile. The sum can in the end be assigned to the particles index in the weights array. This version is possible to accelerate using a GPU and OpenACC.

3.3.4.2 Projection

Code 5: The projection routine:

```

for (int i = 0; i < npart; i++){
    for (int j = 0; j < nprof; j++){
        rec[j][xp[i][j]] += weights[i];
    }
}

```

The rec array is an array containing the reconstructed profiles. It has the same shape as the array containing the original measured profiles (N, M), where N is the number of profiles, and M is the length of a profile in bins. The xp array contains the x-coordinates of each particle at each time frame. The weight array contains the weight factor for each particle. The x-coordinates of the particles also point to a bin in the time projections. The problem for creating a parallel version is that it can be multiple particles with the same x-coordinates, thereby pointing at the same bin. Also, since the x-coordinates vary arbitrarily both for one particle between each machine turn and between the different particles, the routine is hard to vectorise.

Before projection

Reconstructed
profile

0	0	0	0
---	---	---	---

xp

1	1	2	3
---	---	---	---

Weight factor

0.1	0.2	0.3	0.4
-----	-----	-----	-----

Calculations at GPU

$rec[1] = 0 + 0.1$	$rec[2] = 0 + 0.3$
$rec[1] = 0 + 0.2$	$rec[3] = 0 + 0.4$

Returned from GPU

Reconstructed
profile

0	0.1 or 0.2	0.3	0.4
---	------------	-----	-----

Figure 3.16: Projection parallelisation problems.

As seen in figure 3.16, a problem occurs when multiple weights are added to the same bin of the rec array. When attempting to parallelise based on the particles, every bin will be added by using the initial value of the rec array and one of the weight factors. In this example, the bin with the zeroth index will stay unchanged since there are no particles in the bin. For the bins having the indices two and three, only one particle is added and the problem is avoided. At the first index are both the answer 0.1 and 0.2 calculated. Only one of these answers will be written back to the rec array. Timing will decide which value will be saved, as the last to be calculated will overwrite the preceding. An effective solution to this problem is still to be found. Algorithmic changes might be needed, but it is seen as out of scope for this project to develop such a solution.

Chapter 4

Testing

Multiple tests have been introduced in order to evaluate the performance of the Python/C++ version of the tomography code. From these tests, the output quality and the run time is compared to the original version. Both of these measures are important indices for determining the performance of the tomography library.

In the first test, the output discrepancies from the phase space reconstructions are directly compared between the old and the new code. Here, the new code is set up to be a replication of the original program. In this test, the quality of the reconstructions are compared, without measuring the run time. The second test compares the run times of original program and the Python/C++ library. Using the library, a Python program is set up to replicate the old code. In this way, the run times can be measured one-to-one. In the third test, the characteristics of the new algorithm are exploited by using the possibility of running the stages of the reconstruction process independently. Here, only the run time for the tomographic reconstruction is measured. Such a solution makes up a minimal program for the phase space reconstruction, thereby, using the shortest possible run time.

4.1 Output Quality

In this test, the output quality of the new code is measured. By utilising the Python/C++ library, a program is created to provide the same functionality as the original program. The format of the output files from the test program is identical to the output from the Fortran version. In this way, the output can easily be compared.

The Fortran and the Python program was given identical input from a database of 1219 input files of the same format as seen in appendix B. Many of these input files are not intended for valid reconstructions and results in very poor reconstructions, which are disregarded. By using a batch service, the programs were run on a CERN computer cluster. Later, the data was analysed in two steps. First, the output files were labeled as valid or invalid. Valid data is data where the input file ran successfully, with a discrepancy less than 5.0×10^{-2} . Then, the valid data from the Python/C++ program was compared to the equivalent output from the Fortran program. The result of the comparison can be seen in figure 4.1.

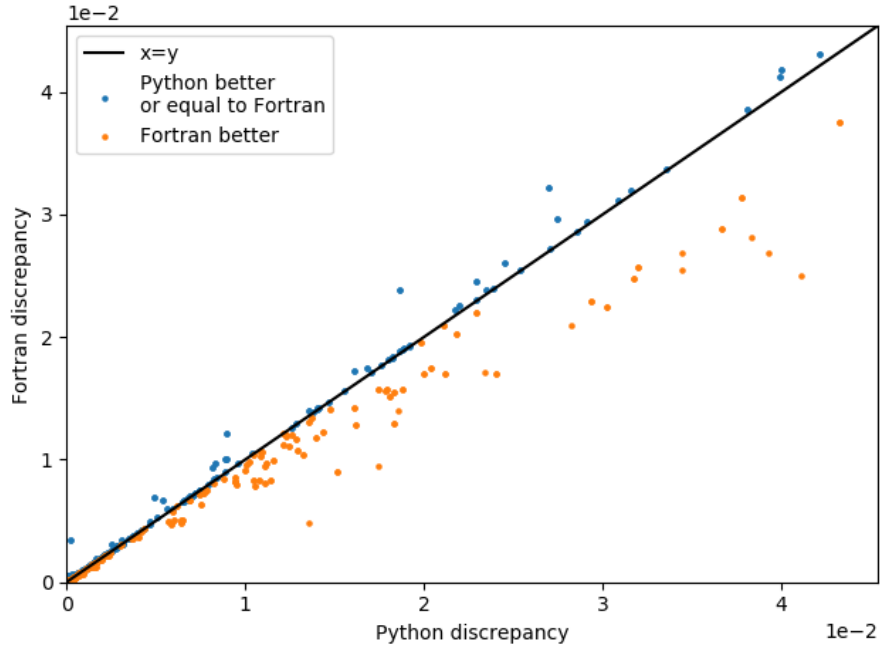


Figure 4.1: Relationship between the discrepancy of the Fortran output and the Python/C++ output.

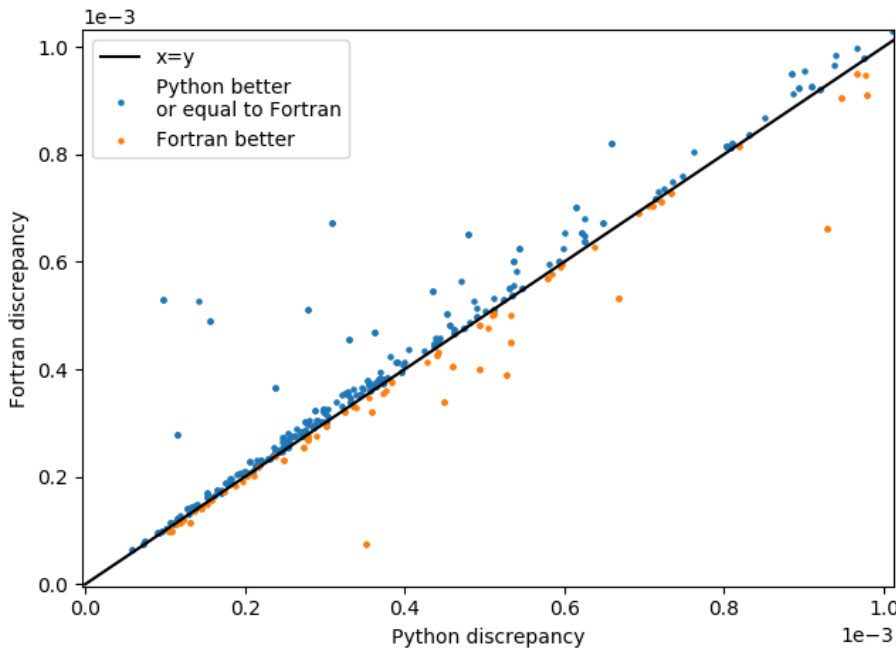


Figure 4.2: Plot showing only discrepancies between 0.0 and 1.0×10^{-3} .

Figure 4.1 shows the relationship between the discrepancy of the output from Python/C++ program (x-axis) and the Fortran program (y-axis). The black line shows the ideal line where Python/C++ is equal to Fortran. Output plotted above the line has a higher discrepancy using the Python/C++ program than the Fortran program. The opposite is true below the line. The figure might give the impression that the

most of the output has a better quality when coming from the the Fortran program. However, by taking a closer look at the lower discrepancies between 0.0 and 1.0×10^{-3} , the opposite can be seen. A plot of these limits in the x- and y-axis can be seen in figure 4.2. Here, the majority of the reconstructed output have the lowest discrepancy when coming from Python/C++. Note that this is the area where most of the discrepancies are found. From the 1219 input files, 919 (75.4%) of the reconstructions were marked as valid. From these, 586 (63.8%) files had a lower discrepancy when run with Python/C++.

Further tests were done using a smaller amount of representative input files. These files were a collection of inputs, earlier used operationally. This was done in order to obtain a overview of the output generated by a realistic input. These input files amounted to 107 files. From these, 84 (78.5%) had a lower discrepancy when reconstructed using the Python/C++ rather than the original program. The measurements indicates that the reconstructions holding the extremities in the discrepancies, also holds the largest deviations in between the two versions. Reconstructions of low discrepancy from the original program are often recreated with lower discrepancy using the Python/C++ version. The opposite happens with reconstructions of high discrepancy from the original program. However, on an overall basis the reconstructions results in a lower discrepancy using the Python/C++ code than the original program.

4.2 Run Time: Full Program

A test was set up to evaluate the run time of a Python/C++ implementation of the original program. Input from 25 different files were used, and their run time was measured. Thirteen of the files were operationally used in the machines LEIR, PSB and PS. The remaining twelve were measurement of bunches with complex structures. The average run times after ten runs of each reconstruction can be seen in figure 4.3.

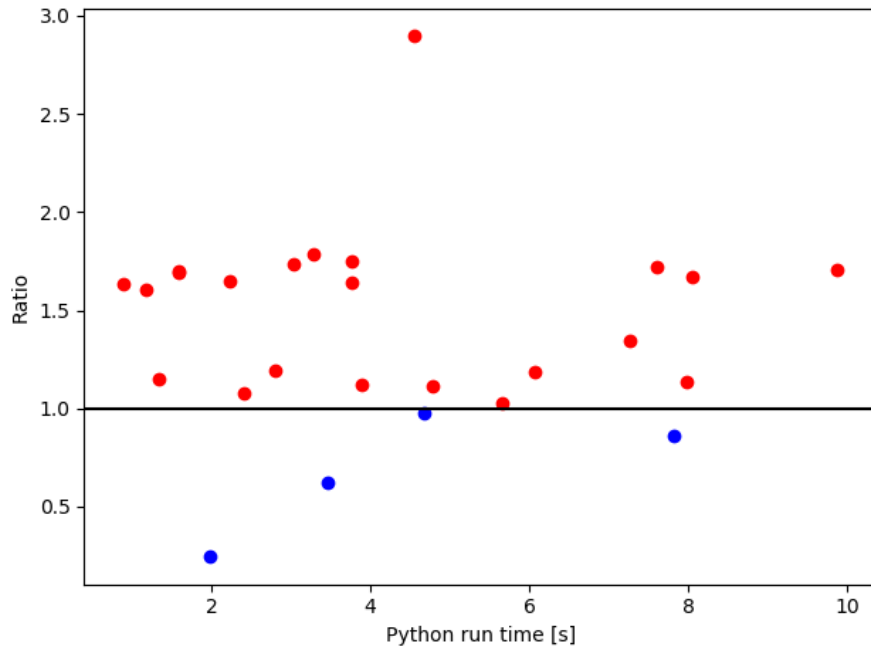


Figure 4.3: Ratio between the run time of the original program and a Python program set up as the original program, using the new library. Blue dots marks cases where Python/C++ has the lowest run time. Red dots marks the opposite.

Here, the result of the measured run times can be seen. The x-axis shows the run time of the Python/C++

reconstruction. The y-axis shows the ratio between the run time of the Python/C++ and the Fortran version. Ratios higher larger than 1, marked with red dots, are faster using the Fortran program. The contrary is shown using blue dots. From the 25 files run, only four of the input files were recreated more rapidly using the Python/C++ version than the Fortran version.

4.3 Run Time: Minimal program

The tomography library is designed in such a way that parts of the reconstruction process can be executed individually. This allows for the creation of a minimalist program, consisting only of the tomographic reconstruction without particle tracking. A database can be filled with tracked particles corresponding to frequently used machine settings can be loaded on demand, and be used as input for the tomographic reconstruction routine together with the measured profiles. The following test will investigate the potential in terms of run time.

In order to find the optimum settings for a quick reconstruction of the same or better quality as the original, a parameter scan is performed. The varied parameters are the square root of the number of particles per cell of the reconstructed phase space (snpt), and the re-binning factor (the number of bins in the profile measurements to merge into one bin). This is done for a handful of images. The result of a scan can be seen in figure 4.4. Here, the x-axis is the run time of the reconstruction, and the y-axis is the discrepancy of the reconstructed phase space. The lines represent the re-bin factor, the colour of the dots represent the snpt, and the dotted line represents the discrepancy of the original output from the Fortran program at original settings. In order to find the optimal reconstruction, results having a larger discrepancy than the original was rejected. From the remaining output, the fastest reconstruction was kept for further references. This process was repeated for all tested input files.

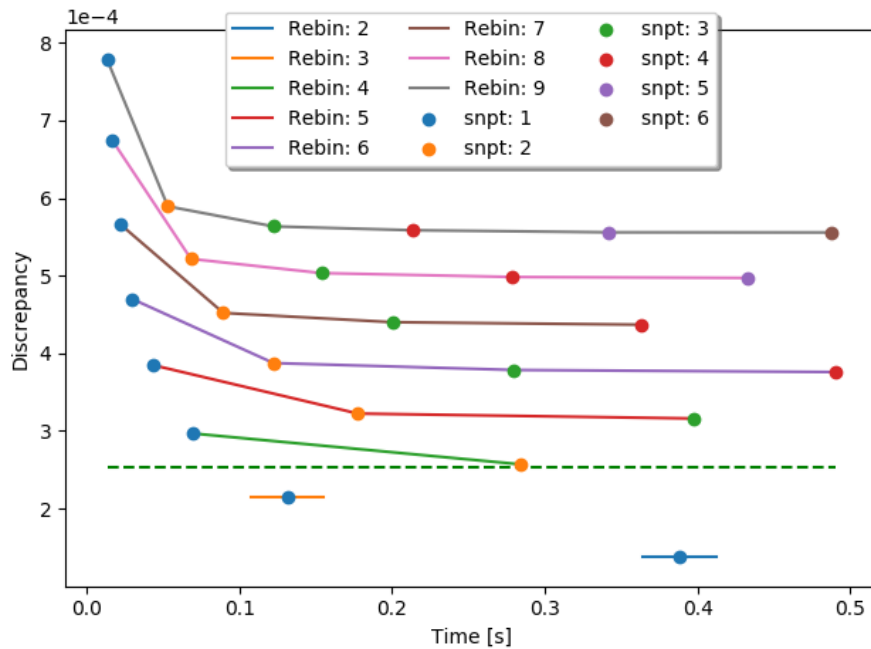


Figure 4.4: Figure showing the discrepancy and run time of the tomographic reconstruction while varying the input parameters. Lines shows the rebin parameter, dots shows the snpt parameter. The dotted green line shows the original discrepancy.

The optimal result of each parameter scan is shown in figure 4.5. Notice that all of the results have a discrepancy ratio lower than 1. The x-axis indicates the run time of the reconstruction process, and the y-axis is the ratio between the Fortran and the Python/C++ output. The vertical dotted green line marks a run time of 0.5 seconds.

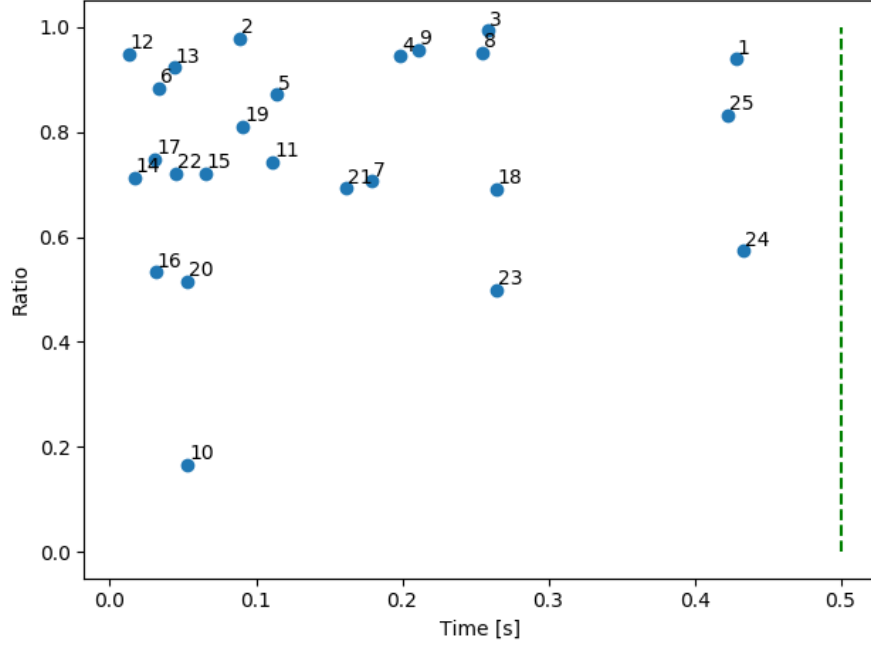


Figure 4.5: Figure showing the run time and ratio between the discrepancy of the Python/C++ version and the Fortran version. Each dot represents a file with some optimal reconstruction parameters (see table C.1). Blue dots represent Python/C++ reconstruction having lower discrepancy than Fortran. The dotted green line marks the 0.5 second limit.

This means that there is a potential for the method. By varying the reconstruction parameters, a compromise between run time and output quality can be found. The test indicates that for many input files, an optimal configuration can be set, resulting in fulfilling demands for both values.

4.4 Unit Tests and Documentation

In order to test the code, a suite of unit test was written using the Python library `pytest` [22]. The repository is set up to automatically run the tests every time code is pushed to GitLab. This is done to ensure that new code runs correctly. Furthermore, the coverage of the unit tests is inspected and visualised using the Python library `coverage` [23].

The code was toughly documented. In order to be as user friendly as possible, the documentation is presented using the `sphinx` library. This converts automatically from comments in the code to formatted documentation.

Chapter 5

Discussion

Various difficulties and problems were met during the development process. Among these are: programming languages and techniques unknown for the developer, limited earlier experience with different tools and systems used at CERN, and minor knowledge of accelerator physics and no experience with HW acceleration. In this chapter, the the development process will be evaluated.

5.1 Planning

In the projects earliest stage, an execution plan was created. The plan was important to assert that everything could be done in the time available. Here, the project was decided to be split into two phases, the first phase lasting from February to August and the second lasting from August to February.

The goal of the first phase was to have a working version of the program as a direct translation from the original Fortran version. The translated program should also be sped up using C++ or Python libraries. Meanwhile, the targets of the second part of the project could not be decided in the earliest stages of the project. For this reason, the planning of the second phase had to happen at the end of the first. The goals for the second phase was to have a modular version of the program and to test out the possibilities of HW acceleration.

5.2 Translation Process

Translating the program was done according to the plan, but some initial problems were met. Most of these were introduced due to lack of experience with programming languages and tools.

The first problem encountered was the lack of Fortran knowledge. Since the original program was written in Fortran, the syntax and commands of the language had to be learned in order to understand how the program worked. Learning Fortran happened by working with the program, but the lack of prior experience with the language made the translation a time consuming process.

A sparse documentation combined with no knowledge of accelerator physics made the understanding of the program harder. Some time had to be spent studying the theory behind the program.

After the Fortran program was investigated, the process of translating it to Python could start. Due to a very limited knowledge of Python at this time, the initial translation happened at a relatively slow pace. However, thanks to the large online community of Python users and earlier experience with languages like C++, C# and JavaScript, the learning curve was manageable. The initial iteration of the translation was written rather poorly, but as the the experience using Python grew, the program was rewritten and improved.

At the same time as learning Fortran and Python, the usage of Anaconda, git, and tools like PyCharm and Eclipse had to be investigated. An extra difficulty was to use this in a Linux environment. Having no prior experience with any of these tools, some time was spent on getting everything to run smoothly and to learn how to use the functionality of the tools in the best way for the project.

During the translation process, a "bug" was found resulting in a bleached phase space compared to the original output. After investigating the problem for some time, a solution was found. The reason was due to a floating point mismatch between the Fortran and the Python Program. The full explanation can be found in chapter 3.2.5. Between one and two weeks were lost in the examination of this problem.

When the first version of the translation was finished, a new repository was created on GitLab. Here it was desired to set up continuous integration / continuous delivery (CI/CD) for the solution. This ended up taking longer time than expected as the documentation was quite sparse. Also, the integration of coveralls, a plugin to show which parts of the code are covered by unit tests, was not straight forward.

5.3 Speeding Up the Code

The process of speeding up the code was relatively straight forward. The first speed-up was done by using the Python library numba. this could easily be done by adding decorators to functions to be compiled at run time. However, this also came with some obstacles. Functions to be accelerated using numba could only contain native Python code and basic ndarray functionality from the numpy library. Furthermore, the functions could not be part of objects. This meant that many functions had to be adjusted or re-written. Also, code sped up using numba would not be seen by the coveralls plug-in, resulting in a lower coverage reported than in reality.

Later, the functions sped up using numba was substituted by C++ routines. By using the ctypes library, an interface could be made connecting Python and the C++ routines. This was done by using the 'external "C" keyword'. This means that the routines can be called from both C and C++. For this reason Python, based on C, can call the routines using scalars and numpy arrays as arguments.

The interface between the Python and the C++ code came with some problems. Python gets access to the C++ routines by importing them from a shared library. The shared library can be compiled correctly using the provided compile script. However, this approach led to some unforeseen problems. First, the Python data types must be explicitly stated. For example, when a C++ function takes an array of doubles as the argument, the Python array must be of the type float64. If another datatype is provided, subtle errors may arise. Secondly, the Python arrays are not guaranteed to be contiguous in memory. This means that the elements of the array are not guaranteed to be saved neatly on a "line", but rather are saved at random locations of the memory. For this reason, bugs can arise when iterating through an array in C++, given as a pointer for a numpy array. An example of such an error is the following. Imagine that a two dimensional array is created and filled. Later, this array is transposed using a library function, not actually changing the the array in the memory. What is then assumed by the programmer is an transposed matrix, get passed on to a C++ routine which iterates through the array. Here, the C++ loop will read the non-transposed matrix from memory. Situations like these can also lead to subtle errors. Luckily, errors like these can be prevented by using functions like `numpy.ascontiguousarray` [24] before passing data to C++ from Python. For these reasons, the interfaces between Python and the C++ functions must be protected carefully. Assertions and exception handling must be introduced in order to catch errors before they reach the C++ routines.

During the HW acceleration, some problems were introduced while installing CUDA10 and and setting up PyCUDA. However, PyCUDA was early found to give small return for the time spent on developing it, and was abandoned in favour of OpenACC. Some time was needed for learning how to use OpenACC, but it led to some promising results regarding the particle tracking. For the acceleration of the tomography routine there is still some work to be done. An experienced GPU programmer should be able to improve the current solution, and perform the changes necessary for an effective HW accelerated tomography routine.

Experimenting using HW acceleration (see chapter 3.3) has shown that the usage of GPUs can lead to a large increase in performance. For the particle tracking, one can easily implement an efficient parallel version to be run at a GPU. The naïve GPU implementation used in the experiments can doubtless be improved by fine tuning the implementation. Also, it would be beneficial to create an implementation and by writing a specific parallel version using native CUDA, optimised for the given hardware. For the tomography part of the reconstruction algorithm, more investigation is necessary in order to implement an efficient parallel version of the reconstruction routine. It appears that changes in the algorithm will be needed, but it is out of the scope of this project to find or create this algorithm.

5.4 Run Time and Output Quality

The longitudinal beam tomography code was changed from being a program to a modular library. A choice was made to use Python as the main language, and C++ for the computationally demanding tasks. This was done in order for the library to be used as a tool in the easiest way possible.

Changes in the algorithm was needed in order to make the program modular and easier to understand. The new algorithm has changed the way the program scales in run time by the amount of tracked particles. The original program tracks the particles between two time frames, and sorts the info about the particles in a collection of arrays before tracking to the next time frame. This process is described in greater depth in chapter 3.1.4.3. Meanwhile, in the new version is the position of each individual particle saved at each time frame. In the next step of the process, the tomographic reconstruction, the Fortran version will mostly depend on the number of bins in the phase space, rather than the number of points. The Python version, working directly on each individual particle, is directly dependant on the number of particles also at this stage of the reconstruction. Read more about the new algorithm in chapter 3.2.3.

Tests of reconstruction using a Python/C++ implementation of the original program is described in chapter 4.2. Here, it is shown that the new program is overall somewhat slower than the earlier version. However, with the new algorithm and modular structure of the longitudinal tomography code, it is possible to reduce the run time considerably. This can be done by creating a data base of tracked particles, corresponding to a set of machine settings. These particles can the be loaded on demand for rapid phase space reconstructions using only the tomography routines. Such a solution has been tested by timing only the tomographic reconstruction (see chapter 4.3). The potential was shown for the usage of this method as the reconstructions could happen at run times often under 0.5 seconds. This time limit has been a goal for the project (see chapter 2.1). These result show the potential of setting up automatic on-line tomography for injections and extractions at the PSB.

Chapter 6

Conclusion

A program for longitudinal beam tomography written in Fortran95 has been used for many many years, and is an important part of the beam diagnostics during operation of the CERN synchrotrons. A modern version of the program is desired, and should replace the original program after the Long Shutdown 2. The new program should be fast and flexible, and able to meet future requirements such as the possibility to be used for automatic online tomography. Also, it should contain the same functionality and quality as the original program, and be able to run at the systems currently using the tomography program.

It was decided that the new tomography code should be a Python library, where the computationally demanding parts are sped up using optimised C++. In addition to this, the possibility to speed up the reconstruction process using hardware acceleration was decided to be investigated. It was finally decided to set up an experimental implementation using GPUs for the evaluation.

The analysis of the original program pointed out some challenges with the reconstruction algorithm. First, the original program was written in a monolithic structure, consisting of highly optimised Fortran code. Combined with a complex array structure and a lack of documentation, the original program was somewhat hard to work with and develop further as is.

Translating to Python was done in several iterations. The first of which, was a direct translation from the original program. Next, a more easy-to-understand algorithm was developed and implemented. This algorithm had a series of benefits. First, it was more parallelizable, leading to a greater chance to successfully implement a GPU version. Secondly, the different parts of the algorithm can be run independently. This was a benefit for making the library as modular as possible. When the algorithm was changed, the program was sped up using C++ for the most computationally demanding parts of the algorithm. The C++ code was optimised, and sped up using the compiler directive OpenMP for automatic parallelism on the CPU. Wrapper functions were made creating an easy and safe interface between Python and the C++ functions. Finally the Python program was restructured into a modular library. The library was organised with relation to functionality and data required. It created an easy interface for users who wish either to use the full reconstruction algorithm, or to customise it for their own specific needs. In addition, the code was covered by unit tests, and was documented thoroughly for the aid of users and future developers.

A GPU version was implemented. This lead to substantial speed-ups for large parts of the algorithm. However, for one part of the algorithm, more time is needed to implement an effective parallel version than available for this project.

The output of the new version has, in most cases, proven to be of better quality than the original. Also, it has been observed that some cases not run by the original program, can be run using the new code. On the other hand, a discrepancy between the output of the original and the new version has been seen in some cases due to difference in the size of the floating points between the versions. Tests of the run time has shown that the Python/C++ library set up as the original program leads to a slightly longer run time. This is due to the changes in the algorithm, and the fact that Python is a interpreted language, introducing

overhead and loss of time compared to Fortran and C++.

A workaround for a shorter run time has been developed. Because of the new modular version it is now possible to calculate needed data for the reconstruction and store it in a database. This data can then be loaded on demand and be used in a minimal program, cutting the run time to a fraction of the original. Soon, this will be used for an experimental set up for online tomography in the PS Booster [25].

During this project, the student has learned a lot about Python, GIT, Accelerator physics, GPU programming, Linux, C++, Fortran, Computer Science, L^AT_EX, academic writing, and the process of process of creating a program from scratch.

6.1 Further Improvements

A GPU implementation for longitudinal beam tomography should be investigated more closely. More functionality could be added to the `data_treatment` module, like options for the estimation of the synchronous phase, noise reduction, and baseline cuts. The algorithm for particle tracking using self fields is somewhat outdated, and would benefit from some further improvements.

Bibliography

- [1] CERN, “Our mission.” <https://home.cern/about/who-we-are/our-mission>, 2019. [Online; accessed 02-may-2019].
- [2] CERN, “Cern beam longitudinal dynamics code blond.” <http://blond.web.cern.ch>, 2019. [Online; accessed 19-January-2020].
- [3] CERN, “Beams & rf studies (br).” <https://be-dep-rf.web.cern.ch/content/br-beams-rf-studies>, 2016. [Online; accessed 03-may-2019].
- [4] S. Hancock, “A simple algorithm for longitudinal phase space tomography,” tech. rep., 1997.
- [5] CERN, “Wall current monitors.” <http://psring.web.cern.ch/psring/psring/misc/wcm.html>, 2014. [Online; accessed 19-January-2020].
- [6] P. Kozłowski and A. Lasheen, “Ideas and proposal for improved longitudinal beam observation in the ps,” tech. rep., 2019.
- [7] R. Bradshaw, C. Citro, and D. S. Seljebotn, “Cython: the best of both worlds,” CiSE 2011 Special Python Issue, p. 25, 2010.
- [8] JetBrains, “Features.” <https://www.jetbrains.com/pycharm/features/>, 2019. [Online; accessed 19-January-2020].
- [9] I. Eclipse Foundation, “Eclipse ide.” <https://www.eclipse.org/eclipseide/>, 2019. [Online; accessed 19-January-2020].
- [10] Microsoft, “Visual studio code.” <https://code.visualstudio.com/>, 2019. [Online; accessed 19-January-2020].
- [11] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation,” Parallel Computing, vol. 38, no. 3, pp. 157–174, 2012.
- [12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” Queue, vol. 6, no. 2, pp. 40–53, 2008.
- [13] J. E. Stone, D. Gohara, and G. Shi, “Opencl: A parallel programming standard for heterogeneous computing systems,” Computing in science & engineering, vol. 12, no. 3, pp. 66–73, 2010.
- [14] N. Corp, “Pgi documentation.” <https://www.pgroup.com/resources/docs/19.10/x86/index.htm>, 2019. [Online; accessed 19-January-2020].
- [15] S. Hancock and S. A. J-L, “A pedestrian guide to online phase space tomography in the CERN complex,” Tech. Rep. CERN-PS-RF-NOTE-2001-010, CERN, Geneva, Aug 2001.

- [16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in Fortran 90 (2nd Ed.): The Art of Parallel Scientific Computing. USA: Cambridge University Press, 1996.
- [17] R. Gordon, “A tutorial on art (algebraic reconstruction techniques),” IEEE Transactions on Nuclear Science, vol. 21, no. 3, pp. 78–93, 1974.
- [18] A. Savitzky and M. J. Golay, “Smoothing and differentiation of data by simplified least squares procedures,” Analytical chemistry, vol. 36, no. 8, pp. 1627–1639, 1964.
- [19] S. Hancock, M. Lindroos, and S. Koscielniak, “Longitudinal phase space tomography with space charge,” Physical Review Special Topics-Accelerators and Beams, vol. 3, no. 12, p. 124202, 2000.
- [20] S. Hancock, M. Lindroos, E. McIntosh, and M. Metcalf, “Tomographic measurements of longitudinal phase space density,” Computer Physics Communications, vol. 118, no. 1, pp. 61–70, 1999.
- [21] R. Farber, Parallel programming with OpenACC. Newnes, 2016.
- [22] B. Oliveira, pytest Quick Start Guide: Write better Python code with simple and maintainable tests. Packt Publishing Ltd, 2018.
- [23] N. Batchelder, “coverage.py documentation.” <https://coverage.readthedocs.io/en/v4.5.x/index.html#>, 2019. [Online; accessed 13-January-2019].
- [24] T. S. community, “numpy.ascontiguousarray.” <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ascontiguousarray.html>, 2019. [Online; accessed 26-January-2019].
- [25] S. Albright, “A Proposal for an Automatic Longitudinal Tomography Demonstration Project in the Proton Synchrotron Booster,” Tech. Rep. CERN-ACC-NOTE in preparation, CERN, Geneva, 2020.

Appendices

Appendix A

Abbreviations and Dictionary Explanations

The following list contains an overview of abbreviations and technical terms used in the paper.

- CERN - The European Organization for Nuclear Research
- BE-RF-BR - Beams department - Radio Frequency group - Beam and RF studies section
- RF - Radio Frequency
- BLoND - "Beam Longitudinal Dynamics" simulation library developed at CERN
- PSB - Proton Synchrotron Booster
- PS - Proton Synchrotron
- SPS - Super Proton Synchrotron
- LHC - Large Hadron Collider
- LS2 - Long Shutdown 2
- CCC - CERN Control Center
- HW - hardware
- SW - software
- GPU - Graphics Processing Unit
- FPGA - Field-Programmable Gate Array
- OS - Operating System
- IDE - Integrated Development Environment
- HPF - High Performance Fortran
- PGI - The Portland Group, Inc
- pghpf - Portland Group High Performance Fortran compiler

- GUI - Graphical User Interface
- ART - Algebraic reconstruction technique
- CI/CD - Continuous Integration and Continuous Delivery
- Beam - Stream of charged particles
- Bunch - Group of charged particles
- Synchronous particle - Theoretical particle with an ideal phase and energy. Used as a reference for other particles in the same bunch
- Particle accelerator - Machine used to change the energies of a (group of) particle(s)
- separatrix - Border between stable and unstable synchrotron motion.
- emittance - Area within a bunch populated by stable particles.
- f32 - 32 bit floating point precision.
- f64 - 64 bit floating point precision.
- LEIR - Low Energy Ion Ring

Appendix B

Example Fortran Input File

Input file: MidIDIVNoiseC350-2.dat

```
MD2=MD3088\_LHCINDIV\_PN
1524128345
350
1212.998
3
1253.0000000000002
0.01
-2.182389937106919
! Version 2
!
! Data Format:
! Input data file =
pipe
! Directory in which to write all output =
<Your favourite directory!>
! Number of frames in input data =
100
! Number of frames to ignore =
0
! Number of bins in each frame =
1000
! Width (in s) of each frame bin =
9.999999999999999E-10
! Number of machine turns between frames =
10
! Number of frame bins before the lower profile bound to ignore =
130
! Number of frame bins after the upper profile bound to ignore =
40
! Number of frame bins after the lower profile bound to treat as empty
! at the reconstructed time =
0
! Number of frame bins before the upper profile bound to treat as empty
! at the reconstructed time =
0
! Number of frame bins to rebin into one profile bin =
3
```

```

! Time (in frame bins) from the lower profile bound to the synchronous phase
! (if <0, a fit is performed) in the bunch reference frame =
326.00000000000006
! Max energy (in eV) of reconstructed phase space (if >0) =
-1.E6
! Number of the first profile at which to reconstruct =
21
! Number of the last profile at which to reconstruct =
21
! Step between reconstructions =
1
! Number of iterations for each reconstruction =
20
! Square root of the number of test particles to track per cell =
4
! Flag to extend the region in phase space of map elements (if =1) =
0
! Reference frame for bunch parameters (synchronous phase, baseline, integral) =
21
! Reference frame for machine parameters (RF voltages, B-field) =
1
!
! Machine and Particle Parameters:
! Peak RF voltage (in V) of principal RF system =
2720.964655042207
! and its time derivative (in V/s) =
0.0
! Peak RF voltage (in V) of higher-harmonic RF system =
0
! and its time derivative (in V/s) =
0.0
! Harmonic number of principal RF system =
1
! Ratio of harmonics between RF systems =
2.0
! Phase difference (in radians of the principal harmonic) between RF systems =
0.4506418895431263
! Dipole magnetic field (in T) =
0.17859
! and its time derivative (in T/s) =
0.9157142857142578
! Machine radius (in m) =
25.0
! Bending radius (in m) =
8.239
! Gamma transition =
4.1
! Rest mass (in eV/c**2) of accelerated particle =
0.93827231E9
! Charge state of accelerated particle =
1
!
! Space Charge Parameters:
! Flag to include self-fields in the tracking (if =1) =

```

```

0
! Geometrical coupling coefficient =
0.0
! Reactive impedance (in Ohms per mode number) over a machine turn =
0.0
! Effective pick-up sensitivity (in digitizer units per instantaneous Amp) =
0.36
-0.03828125*
-0.03828125
-0.0390625
-0.03828125
-0.03828125
-0.037890625000000004
-0.037890625000000004
-0.0390625
-0.037890625000000004
...

```

** Start of measured data*

Appendix C

Table of Optimal Reconstruction Settings for Reference Files

#input	Machine	Cycle	Run Time	Ratio	Rebin	Snpt
1	LEIR	1840	0.428	0.939	2	1
2	LEIR	1675	0.089	0.979	2	2
3	PSB	796	0.259	0.993	2	2
4	PSB	798.	0.199	0.945	2	2
5	PSB	798	0.114	0.872	2	2
6	PSB	798	0.033	0.883	2	1
7	PSB	778	0.179	0.706	2	2
8	PSB	798	0.255	0.952	1	4
9	PSB	798	0.211	0.957	1	4
10	PS	1060	0.053	0.165	1	1
11	PS	200	0.111	0.742	1	1
12	PS	1130	0.013	0.948	2	1
13	PS	1050	0.044	0.923	2	2
14	PS	237	0.017	0.714	8	2
15	PS	344	0.065	0.722	2	1
16	PS	1940	0.031	0.533	2	1
17	PS	1395	0.031	0.748	2	1
18	PS	200	0.264	0.690	1	1
19	PS	740	0.091	0.811	2	1
20	PS	1130	0.053	0.515	1	1
21	PS	237	0.162	0.693	4	2
22	PS	500	0.045	0.721	2	1
23	PS	690	0.264	0.498	1	1
24	PS	200	0.433	0.573	2	1
25	LEIR	1840	0.423	0.831	2	2

Table C.1: Measurements for scatter plot in figure 4.5.

Appendix D

Program Documentation

tomo

Release 1.0

Christoffer Hjertø Grindheim

Feb 19, 2020

CONTENTS:

1	Contents	1
2	Modules	3
2.1	data Package	3
2.1.1	profiles Module	3
2.2	tracking Package	5
2.2.1	machine Module	5
2.2.2	__tracking Module	10
2.2.3	particles Module	12
2.2.4	phase_space_info Module	15
2.2.5	tracking Module	16
2.3	tomography Package	19
2.3.1	__tomography Module	19
2.3.2	tomography Module	21
2.4	utils Package	23
2.4.1	physics Module	23
2.4.2	data_treatment Module	26
2.4.3	assertions Module	27
2.4.4	exceptions Module	32
2.4.5	tomo_input Module	33
2.4.6	tomo_output Module	36
2.4.7	tomo_run Module	38
2.5	cpp_routines Package	38
2.5.1	tomolib_wrappers Module	38
2.6	compile Module	42
2.6.1	compile Module	42
3	Indices and tables	43
	Python Module Index	45
	Index	47

CONTENTS

The Longitudinal Beam Tomography code tomo is a CERN software package for the reconstruction of the phase space density of a particle bunch.

2.1 data Package

2.1.1 profiles Module

Module containing the Profiles class for storing measurements.

Author(s) Christoffer Hjertø Grindheim

class tomo.data.profiles.Profiles (*machine, sampling_time, waterfall, profile_charge=None*)
 Bases: object

Class holding measured data.

This class holds the measured data (waterfall) and their properties.

The waterfall is a nprofiles x nbins shaped array, waterfall[0] is the first measurement. Each bin is *Machine.dtbins* long, and holds the intensity of the beam at this time of the measurement.

The profile charge can be provided, or calculated using the *Profiles.calc_profilecharge* function which is based on the machines pickup sensitivity.

Also, the class can be used to calculate the self-fields of the bunch by calling the *Profiles.calc_self_fields* function.

Parameters

- **machine** (*Machine*) – Holds information about the measurements and the machine.
- **sampling_time** (*float*) – Original measurement sampling time.
- **waterfall** (*ndarray*) – 2D-array containing all profile measurements.
- **profile_charge** (*float, optional, default=None*) – Total charge of a reference profile.

machine

The machine and its settings when measurements was taken. This object is needed for assertions, to check that the provided waterfall is correct compared to the machine parameters. It is also needed for calculating the self fields and profile charge of the bunch.

Type *Machine*

sampling_time

Original sampling time of measurements. Needed for calculation of profile charge.

Type float

waterfall

2D array containing all profile measurements. Array should have the shape: (N, M), where N is the number of profiles and M is the number of bins of each profile.

Type ndarray

profile_charge

The total charge of a reference profile.

Type float

vself

2D array of self-fields at each bin of each profile. The shape of the array is (N, W), N is the number of profiles, and W is the *Profiles.wrap_length*.

Type ndarray, float

dsprofiles

2D array containing Filtered profiles. Shape should be (N, M), where N is the number of profiles and M is the number of bins of each profile.

Type ndarray, float

phiwrap

The phase [rad] covering an integer of rf periods.

Type float

wrap_length

Maximum number of bins to cover an integer number of rf periods.

Type float

calc_profilecharge ()

Calculate the total charge of profile. Uses the beam reference profile to decide which profile should be used for the calculation. Sets the *Profiles.profile_charge* field.

NB: The function requires that the **original sampling time** of the measurements is provided.

calc_self_fields (*filtered_profiles=None, in_filter=None*)

Calculate self-fields based on filtered profiles. If filtered profiles are not provided by the user, standard filter (savitzky-golay smoothing filter) is used.

The function sets the following fields:

- **vself** - 2D array of self-fields at each bin of each profile.
- **dsprofiles** - Filtered profiles.
- **phiwrap** - Phase covering an integer of rf periods.
- **wrap_length** - Maximum number of bins to cover an integer number of rf periods.

A description of these attributes can be found in the documentation for the *Profiles* class.

Parameters

- **filtered_profiles** (*ndarray (optional, default=None)*) – If the filtered profiles are provided, they will be used in the calculation of the self fields. If not, the original profiles will be filtered using a standard or user specified filter.
- **in_filter** (*function (optional, default=None)*) – If provided, the measured profiles will be filtered using the provided filter in stead of the savitzky-golay smoothing filter.

Raises

- ***ProfileChargeNotCalculated*** – Exception: No profile charge has been provided or calculated.
- ***FilteredProfilesError*** – Exception: Filtered profiles has the wrong shape or are not iterable.

property waterfall

Waterfall defined as @property. The property does the following when accessed:

- Asserts for correctness of waterfall, here the number of profiles should be as stated by the *Machine* object.
- Negative values of waterfall are set to zero.
- *Machine.nbins* is updated with the number of bins in the waterfall.

Parameters waterfall (*ndarray*) – Measured profiles as a 2D array. Shape: (N, M), N is the number of profiles and M is the number of bins of each profile.

Returns waterfall – Measured profiles as a 2D array. Shape: (N, M), N is the number of profiles and M is the number of bins of each profile.

Return type ndarray

Raises

- ***WaterfallError*** – Exception: Waterfall not iterable or wrong has the amount of profiles.
- ***WaterfallReducedToZero*** – Exception: All of waterfall is reduced to zero after removal of negative values.

2.2 tracking Package

2.2.1 machine Module

Module containing Machine class for storing machine and reconstruction parameters

Author(s) Christoffer Hjertø Grindheim

```
class tomo.tracking.machine.Machine(dturns, vrf1, mean_orbit_rad, bending_rad, b0,
                                     bdot, trans_gamma, rest_energy, nprofiles, nbins,
                                     synch_part_x, dtbin, **kwargs)
```

Bases: object

Class holding machine and reconstruction parameters.

This class holds machine parameters and information about the measurements. Also, it holds settings for the reconstruction process.

The Machine class and its values are needed for the original particle tracking routine. Its values are used for calculation of reconstruction area and info concerning the phase space, the distribution of particles, and the tracking itself. In addition to this, the machine object is needed for the generation of *Profiles* objects.

To summarize, the Machine class must be used if a program resembling the original Fortran version is to be created.

Parameters

- **dturns** (*int*) – Number of machine turns between each measurement.
- **vrf1** (*float*) – Peak voltage of the first RF system at the machine reference frame.

- **mean_orbit_rad** (*float*) – Mean orbit radius of machine [m].
- **bending_rad** (*float*) – Machine bending radius [m].
- **b0** (*float*) – B-field at machine reference frame [T].
- **bdot** (*float*) – Time derivative of B-field (considered constant) [T/s].
- **trans_gamma** (*float*) – Transitional gamma.
- **rest_energy** (*float*) – Rest energy of accelerated particle [eV/C²], saved as `e_rest`.
- **nprofiles** (*int*) – Number of measured profiles.
- **nbins** (*int*) – Number of bins in a profile.
- **synch_part_x** (*float*) – Synchronous phase given in number of bins, counting from the lower profile bound to the synchronous phase.
- **dtbin** (*float*) – Size of profile bins [s].
- **kwargs** – All scalar attributes can be set via the kwargs.

demax

Maximum energy of reconstructed phase space.

Type float, default=-1.E6

dturns

Number of machine turns between each measurement.

Type int

vrfl

Peak voltage of the first RF system at the machine reference frame.

Type float

vrff2

Peak voltage of the second RF system at the machine reference frame.

Type float, default=0.0

vrff1dot

Time derivatives of the voltages of the first RF system (considered constant).

Type float, default=0.0

vrff2dot

Time derivatives of the voltages of the second RF system (considered constant).

Type float, default=0.0

mean_orbit_rad

Mean orbit radius of machine [m].

Type float

bending_rad

Machine bending radius [m].

Type float

b0

B-field at machine reference frame [T].

Type float

bdot
Time derivative of B-field (considered constant) [T/s].
Type float

phi12
Phase difference between the two RF systems (considered constant).
Type float, default=0.0

h_ratio
Ratio of harmonics between the two RF systems.
Type float, default=1.0

h_num
Principle harmonic number.
Type int, default=1

trans_gamma
Transitional gamma.
Type float

e_rest
Rest energy of accelerated particle [eV/C²].
Type float

q
Charge state of accelerated particle.
Type int, default=1

g_coupling
Space charge coupling coefficient (geometrical coupling coefficient).
Type float, default=None

zwall_over_n
Magnitude of Zwall/n, reactive impedance.
Type float, default=None

min_dt
Minimum phase of reconstruction area measured in seconds.
Type float, default=None

max_dt
Maximum phase of reconstruction area measured in seconds.
Type float, default=None

nprofiles
Number of measured profiles.
Type int

pickup_sensitivity
Effective pick-up sensitivity (in digitizer units per instantaneous Amp).
Type float, default=None

nbins
Number of bins in a profile.

Type int

synch_part_x

Synchronous phase given in number of bins, counting from the lower profile bound to the synchronous phase.

Type float

synch_part_y

Energy coordinate of synchronous particle in phase space coordinates of bins. The coordinate is set to be one half of the image width (nbins).

Type float

dtbin

Size of profile bins [s].

Type float

self_field_flag

Flag to include self-fields in the tracking.

Type boolean, default=False

full_pp_flag

If set, all pixels in reconstructed phase space will be tracked.

Type boolean, default=False

machine_ref_frame

Frame to which machine parameters are referenced.

Type int, default=0

beam_ref_frame

Frame to which beam parameters are referenced.

Type int, default=0

snpt

Square root of particles pr. cell of phase space.

Type int, default=4

niter

Number of iterations in tomographic reconstruction.

Type int, default=20

filmstart

First profile to reconstruct.

Type int, default=0

filmstop

Last profile to reconstruct.

Type int, default=1

filmstep

Step between profiles to reconstruct.

Type int, default=1

output_dir

Directory to save output.

Type string, default=None

time_at_turn

1D array holding the time at each turn. Turn zero = 0 [s].

Type ndarray

phi0

1D array holding the synchronous phase angle at the end of each turn.

Type ndarray

e0

1D array holding the total energy of synchronous particle at the end of each turn.

Type ndarray

beta0

1D array holding the Lorentz beta factor (v/c) at the end of each turn.

Type ndarray

deltaE0

1D array holding the difference between $e0(n)$ and $e0(n-1)$ for each turn.

Type ndarray

eta0

1D array holding the phase slip factor at each turn.

Type ndarray

drift_coef

1D array holding coefficient used for calculating difference, from phase n to phase $n + 1$. Needed by trajectory height calculator, and tracking.

Type ndarray

omega_rev0

1D array holding the revolution frequency at each turn.

Type ndarray

vrf1_at_turn

1D array holding the peak voltage at each turn for the first RF station.

Type ndarray

vrf2_at_turn

1D array holding the peak voltage at each turn for the second RF station.

Type ndarray

load_fitted_synch_part_x_ftn (*fit_info*)

Function for setting the `synch_part_x` if a fit has been performed. Saves parameters retrieved from the fitting routine needed by the `write_plotinfo_ftn()` function in the `tomo.utils.tomo_output`. All needed info will be returned from the `fit_synch_part_x()` function.

Sets the following fields:

- **fitted_synch_part_x** The new x-coordinate of the synchronous particle (needed for `write_plotinfo_ftn()`).
- **bunchlimit_low** Lower phase of bunch (needed for `write_plotinfo_ftn()`).
- **bunchlimit_up** Upper phase of bunch (needed for `write_plotinfo_ftn()`).

- **synch_part_x** The x-coordinate of the synchronous particle.

Parameters **fit_info** (*tuple*) – Tuple should hold the following info in the following format: (F, L, U), where F is the fitted value of the synchronous particle, L is the lower bunch limit, and U is the upper bunch limit. All of the values should be given in bins. The info needed by the `write_plotinfo_ftn()` function if a fit has been performed, and the a Fortran style output is to be given during the particle tracking.

property nbins

nbins defined as @property. Updates the position of the y-coordinate of the synchronous particle in the phase space coordinate system when set.

Parameters **nbins** (*int*) – Number of bins in a profile.

Returns **nbins** – Number of bins in a profile.

Return type *int*

values_at_turns()

Calculating machine values for each turn.

The following values are calculated in this function. All are ndarrays of the data type float.

- **time_at_turn** Time at each turn. Turn zero = 0 [s].
- **phi0** Synchronous phase angle at the end of each turn.
- **e0** Total energy of synchronous particle at the end of each turn.
- **beta0** Lorenz beta factor (v/c) at the end of each turn.
- **deltaE0** Difference between $e0(n)$ and $e0(n-1)$ for each turn.
- **eta0** Phase slip factor at each turn.
- **drift_coef** Coefficient used for calculating difference, from phase n to phase $n + 1$. Needed in trajectory height calculator and tracking.
- **omega_rev0** Revolution frequency at each turn.
- **vrf1_at_turn** Peak voltage at each turn for the first RF station.
- **vrf2_at_turn** Peak voltage at each turn for the second RF station.

The values are saved as fields of the Machine object.

2.2.2 `__tracking` Module

Module containing ParticleTracker class, a super class for particle trackers.

Author(s) Christoffer Hjertø Grindheim

class `tomo.tracking.__tracking.ParticleTracker` (*machine*)

Bases: `object`

Super class for classes meant tracking particles.

This class holds some general utilities for tracking particles. These utilities includes assertions and flags.

Parameters **machine** (*Machine*) – Holds all information needed for particle tracking and generating the particle distribution.

machine

Holds all information needed for particle tracking and generation of the particle distribution.

Type *Machine*

particles

Creates and/or stores the initial distribution of particles.

Type *Particles*

nturns

Number of machine turns of which the particles should be tracked through.

Type `int`

self_field_flag

Flag to indicate that self-fields should be included during the tracking.

Type `boolean`

fortran_flag

Flag to indicate that the particle tracking should print Fortran-style output strings to stdout during tracking.

Type `boolean`

Raises *MachineParameterError* – Exception: Input argument is not *Machine*, or the Machine object provided is missing needed fields.

enable_fortran_output (*profile_charge*)

Function for enabling of Fortran-styled output.

Call this function in order to print a Fortran-styled output to stdout during the particle tracking.

The output will initially be a print of the plot info needed for the tomoscope application. Then, During the tracking, the output will print which profiles the particles are currently being tracked between.

The number of 'lost particles' will be also be printed. This is however **not a real measurement**, but a static string needed for the interface to the tomoscope application. The lost particles is not found during the tracking due to changes in the algorithm.

Parameters *profile_charge* (*float*) – Total charge of a reference profile.

Raises *ProfileChargeNotCalculated* – Exception: Needed field for enabling Fortran output, *profile_charge*, is missing from the Machine object.

enable_self_fields (*profiles*)

Function for enabling particle tracking using self-fields.

Call this function to track the particles using self-fields. Note that the self-field tracking is **much slower** than tracking without self-fields.

Parameters *profiles* (*Profiles*) – Self-fields must be calculated in the the Profiles object prior to calling this function. See *tomo.data.profiles.Profiles.calc_self_fields()*.

Raises *SelfFieldTrackingError* – Exception: Needed fields for tracking using self-fields missing from Machine object.

property fortran_flag

self_field_flag defined as @property

Flag can be set to true by calling *enable_fortran_output()*.

Returns *fortran_flag* – Flag to indicate if Fortran styled output is enabled.

Return type `boolean`

property self_field_flag

self_field_flag defined as @property

Flag can be set to true by calling `enable_self_fields()`.**Returns self_field_flag** – Flag to indicate if particle tracking using self-fields is enabeled.**Return type** boolean

2.2.3 particles Module

Module containing the Particles class for creating and storing and initial particle distribution.

Moule also contains utility functions for particle distribution like assertions, conversions and filtering of lost particles.

Author(s) Christoffer Hjertø Grindheim

class tomo.tracking.particles.Particles

Bases: object

Class holding the initial particle distribution to be tracked.

An autmatic homogeneous particle based on the algorithm from the original tomography program can be created, by calling `homogeneous_distribution()`.

dEbin

Energy size of bins in phase space coordinate system.

Type float, default=None**xorigin**

The absolute difference (in bins) between phase=0 and the origin of the reconstructed phase space coordinate system.

Type float, default=None**imin**

Minimum phase of reconstruction area. Given in bins phase space.

Type int, default=None**imax**

Maximum phase of reconstruction area. Given in bins phase space.

Type int, default=None**jmin**

Minimum energy for each phase of the phase space coordinate system.

Type ndarray, default=None**jmax**

Maximum energy for each phase of the phase space coordinate system.

Type ndarray, default=None**coordinates_dphi_denergy**

Tuple containing coordinates of initial particle distribution as 1D ndarrays (dphi, denergy). These are the differences in phase [rad] energy [eV] relative to the synchronous particle.

Type tuple, default=(None, None)

property coordinates_dphi_denergy

Particle coordinates defined as @property. Use this property to provide and retrieve coordinates of the initial particle distribution in units of phase [rad] and energy [eV] relative to the synchronous particle.

This property contains a built-in assertion of the provided particles.

Parameters **coordinates** (*tuple*) – Tuple holding (dphi, denergy)

- **dphi** ndarray containing the phase [rad] of each particle reative to the synchronous particle.
- **denergy** ndarray containing the energy [eV] of each particle reative to the synchronous particle.

Returns

coordinates – Returns a tuple holding (dphi, denergy)

- **dphi** ndarray containing the phase [rad] of each particle reative to the synchronous particle.
- **denergy** ndarray containing the energy [eV] of each particle reative to the synchronous particle.

Return type tuple

homogeneous_distribution (*machine, recprof*)

Function for automatic generation of particle distribution.

The distributions created are identical to the distributions created in the Fortran tomography. The reconstruction area is found by calling `find_binned_phase_energy_limits()`.

Parameters

- **machine** (*Machine*) – Object containing the machine settings during the measurements. Needed for the calculation of the reconstruction area, and for setting the the number of particles to be created.
- **recprof** (*int*) – The index of the profile (time frame) to be reconstructed. This will be the profile where the distribution is generated.

Raises *MachineParameterError* – Exception: Raised if fields are missing from provided machine object.

`tomo.tracking.particles.filter_lost` (*xp, yp, img_width*)

Remove lost particles (particles that leaves the image width).

Parameters

- **xp** (*ndarray*) – 2D array containing x-coordinates of all particles at each time frame. Particle coordinates must be given in phase space coordinates as integers. Shape: (nprofiles, nparts).
- **yp** (*ndarray, int*) – 2D array containing y-coordinates of all particles at each time frame. Particle coordinates must be given in phase space coordinates as integers. Shape: (nprofiles, nparts).
- **img_width** (*int*) – Number of bins in measured profiles.

Returns

- **xp** (*ndarray*) – 2D array containing x-coordinates of all particles at each time frame. Now containing only particles inside of the image width. Shape: (nprofiles, nparts).

- **yp** (*ndarray*) – 2D array containing y-coordinates of all particles at each time frame. Now containing only particles inside of the image width. Shape: (nprofiles, nparts).
- **nr_lost_points** (*int*) – Number of particles removed.

`tomo.tracking.particles.physical_to_coords` (*tracked_dphi*, *tracked_denergy*, *machine*, *xorigin*, *dEbin*)

Function to convert from physical units ([rad], [eV]) to reconstructed phase space coordinates (bin numbers).

The phase space coordinates are needed for the tomographic reconstruction routines.

Parameters

- **tracked_dphi** (*ndarray*) – 2D array containing the phase difference [rad] relative to the synchronous particle for every particle at every time frame. Array shape: (nprofiles, nparticles)
- **tracked_denergy** (*ndarray*) – 2D array containing the energy difference [eV] relative to the synchronous particle for every particle at every time frame. Array shape: (nprofiles, nparticles)
- **machine** (*Machine*) – machine object holding machine parameters, and settings for the reconstruction.
- **xorigin** (*float*) – The absolute difference (in bins) between phase=0 and the origin of the reconstructed phase space coordinate system.
- **dEbin** (*float*) – Energy size of bins in phase space coordinate system.

Returns

- **xp** (*ndarray*) – 2D array containing the x-coordinate for every particle at every time frame, given fractions of bins of the phase space coordinate system. Array shape: (nprofiles, nparticles)
- **yp** (*ndarray*) – 2D array containing the y-coordinate for every particle at every time frame, given fractions of bins of the phase space coordinate system. Array shape: (nprofiles, nparticles)

`tomo.tracking.particles.ready_for_tomography` (*xp*, *yp*, *nbins*)

Function to prepare tracked particles tomography routine.

Handy if particles are tracked using the functions of the `tomo.tracking.tracking` module.

Function does the following actions:

- Removes particles leaving the image width.
- Transposes the coordinate arrays from (nprofiles, nparts) to (nparts, nprofiles).
- Casts coordinates to integers.

The returned coordinates are ready to be used in the tomographic reconstruction routines.

Parameters

- **xp** (*ndarray*) – 2D array containing the x-coordinate for every particle at every time frame, given fractions of bins of the phase space coordinate system. Array shape: (N, M), where N is the number of profiles, and M is the number of particles.
- **yp** (*ndarray*, *float*) – 2D array containing the y-coordinate for every particle at every time frame, given fractions of bins of the phase space coordinate system. Array shape: (N, M), where N is the number of profiles, and M is the number of particles.

Returns

- **xp** (*ndarray*) – 2D array containing the x-coordinate for every particle at every time frame, given in phase space coordinates. Array shape: **(M, N)**, where N is the number of profiles, and M is the number of particles.
- **yp** (*ndarray*) – 2D array containing the y-coordinate for every particle at every time frame, given in phase space coordinates. Array shape: **(M, N)**, where N is the number of profiles, and M is the number of particles.

2.2.4 phase_space_info Module

Module containing the PhaseSpaceInfo class

Author(s) Christoffer Hjertø Grindheim

class tomo.tracking.phase_space_info.PhaseSpaceInfo(*machine*)

Bases: object

Class for calculating and storing information about the phase space reconstruction area.

Parameters **machine** (*Machine*) – Object containing machine parameters and settings.

machine

Object containing machine parameters and settings.

Type *Machine*

jmin

1D array of integers holding the minimum energy for each phase of the phase space coordinate system.

Type ndarray

jmax

1D array of integers holding the maximum energy for each phase of the phase space coordinate system.

Type ndarray

imin

Minimum phase of reconstruction area, given in phase space coordinates.

Type int

imax

Maximum phase of reconstruction area, given in phase space coordinates.

Type int

dEbin

Energy size of bins in phase space coordinate system.

Type float

xorigin

The absolute difference (in bins) between phase=0 and the origin of the reconstructed phase space coordinate system.

Type float

calc_xorigin()

Function for calculating xorigin.

Needed by *find_binned_phase_energy_limits()* in order to calculate the reconstruction area.

Returns **xorigin** – The absolute difference (in bins) between phase=0 and the origin of the reconstructed phase space coordinate system.

Return type float

find_binned_phase_energy_limits()

This function finds the limits in the i (phase) and j (energy) axes of the reconstructed phase space coordinate system.

The area within the limits of i and j will be the phase space reconstruction area. This is where the particles will be populated. By setting the *full_pp_flag* in the provided *Machine* object to True, the reconstruction area will be set to all of the phase space image.

This function gives a value to all attributes of the class.

Raises

- *EnergyBinningError* – Exception: Size of each phase space bin in the enegy axis is less than zero.
- *PhaseLimitsError* – Exception: Error in the limits of the i (phase) axis.
- *EnergyLimitsError* – Exception: Error in the limits of the j (energy) axis.
- *ArrayLengthError* – Exception: Difference in length of jmin and jmax.

find_dEbin()

Function to calculate the size of a energy bin in the reconstructed phase space coordinate system.

Needed by *find_binned_phase_energy_limits()* in order to calculate the reconstruction area.

Returns Energy size of bins in phase space coordinate system.

Return type dEbin

Raises *EnergyBinningError* – Exception: The provided value of machine.demax is invalid, or xorigin is not calculated before the function is called.

2.2.5 tracking Module

Module containing the Tracking class.

Author(s) Christoffer Hjertø Grindheim

class tomo.tracking.tracking.Tracking(*machine*)

Bases: *tomo.tracking.__tracking.ParticleTracker*

Class for particle tracking.

This class perform the particle tracking based on the algorithm from the original tomography program. Here, an initial distribution of test particles are homogeneously distributed across the reconstruction area of the phase space image. Later, the particles will be tracked trough all machine turns and saved for every time frame.

Parameters *machine* (*Machine*) – Holds all information needed for the particle tracking and the generation of initial the particle distribution.

machine

Holds all information needed for the particle tracking and the generation of initial the particle distribution.

Type *Machine*

particles

Creates and/or stores the initial particle distribution.

Type *Particles*

nturns

Number of machine turns particles should be tracked trough.

Type int

self_field_flag

Flag to indicate if self-fields should be included during tracking.

Type boolean

fortran_flag

Flag to indicate if a Fortran-style output should be printed to stdout during particle tracking.

Type boolean

kick_and_drift (*denergy, dphi, rflv, rf2v, rec_prof*)

Routine for tracking a distribution of particles for N turns. N is given by *tracking.nturns*

A full C++ implementation is used in *track()*. This function is implemented as hybrid between Python and C++, and kept for reference.

Parameters

- **denergy** (*ndarray*) – particle energy relative to synchronous particle [eV]
- **dphi** (*ndarray*) – particle phase relative to synchronous particle [rad]
- **rflv** (*ndarray*) – Radio frequency voltages (RF station 1), multiplied by particle charge
- **rf2v** (*ndarray*) – Radio frequency voltages (RF station 2), multiplied by particle charge
- **rec_prof** (*int*) – Time slice to initiate tracking.

Returns

- *dphi* – Phase [rad] relative to the synchronous particle, for each particle at each measured time frame. Shape: (nprofiles, nparts).
- **denergy** (*ndarray*) – Energy [eV] relative to the synchronous particle, for each particle at each measured time slice [eV]. Shape: (nprofiles, nparts).

kick_and_drift_self (*denergy, dphi, rflv, rf2v, rec_prof*)

Routine for tracking a given distribution of particles, including self-fields. Implemented as hybrid between Python and C++.

Routine for tracking, with self-fields, a distribution of particles for N turns. N is given by *tracking.nturns*.

Used by the function *track()* to track using self-fields.

Returns the coordinates of the particles as phase space coordinates for efficiency reasons due to tracking algorithm based on the Fortran tomography. large room for improvement.

Fortran output not yet supported.

Parameters

- **denergy** (*ndarray*) – particle energy relative to synchronous particle [eV]
- **dphi** (*ndarray*) – particle phase relative to synchronous particle [rad]
- **rflv** (*ndarray*) – Radio frequency voltages (RF station 1), multiplied by particle charge
- **rf2v** (*ndarray*) – Radio frequency voltages (RF station 2), multiplied by particle charge
- **rec_prof** (*int*) – Time slice to initiate tracking.

Returns

- **xp** (*ndarray, float*) – 2D array holding the x-coordinates of each particles at each time frame (nprofiles, nparts). Coordinates given in bins of phase space coordinate system.
- **yp** (*ndarray, float*) – 2D array holding the y-coordinates of each particles at each time frame (nprofiles, nparts). Coordinates given in bins of phase space coordinate system.

track (*recprof, init_distr=None*)

Primary function for tracking particles.

The tracking routine starts at a given time frame, with an initial distribution of particles. From here, the particles are tracked ‘forward’ towards the last time frame and ‘backwards’ towards the first time frame.

By default, an distribution of particles is spread out homogeneously over the area to be reconstructed. This area is found using the *PhaseSpaceInfo* class. The homogeneous distribution is placed on the time frame intended to be reconstructed for optimum quality. This is based on the original tomography algorithm.

An user spesified distribution can be given and override the default, automatic generation of particles.

By calling *enable_self_fields()*, a flag indicating that self-fields should be included is set. In this case, *kick_and_drift_self()* will be used. Note that tracking including self fields are much slower than without.

By calling *enable_fortran_output()*, an output resembling the original is written to stdout. Note that the values for the number of lost particles is **not valid**. Note also, that if the full Fortran output is to be printed, the automatic generation of particles must be performed.

Parameters

- **recprof** (*int*) – The profile (time frame) to be reconstructed. Here, the particles will have its initial distribution. By giving negative values as arguments, the index will count from the last time frame.
- **init_distr** (*tuple, optional, default=None*) – An user generated initial distribution. Must be given as a tuple of coordinates (dphi, denenergy). dphi is the phase difference from the synchronous particle [rad]. denenergy is the difference in energy from the synchronous particle [eV].

Returns

- **xp** (*ndarray*) – 2D array containing each particles x-coordinate at each time frame. Shape: (nprofiles, nparts).
 - If self-fields are enabeled, the coordinates will be given as phase-space coordinates.
 - If self-fields are disbeled, the returned x-coordinates will be given as phase [rad] relative to the synchronous particle.
- **yp** (*ndarray, float*) – 2D array containing each particles y-coordinate at each time frame. Shape: (nprofiles, nparts).
 - If self-fields are enabeled, the coordinates will be given as phase-space coordinates.
 - If self-fields are disbeled, the returned y-coordinates will be given as energy [eV] relative to the synchronous particle.

2.3 tomography Package

2.3.1 __tomography Module

Module containing the Tomography super class

Author(s) Christoffer Hjertø Grindheim

```
class tomo.tomography.__tomography.Tomography (waterfall, x_coords=None,
                                              y_coords=None)
```

Bases: object

Base class for tomography classes.

This class holds tomography utilities and assertions.

Parameters

- **waterfall** (*ndarray*) – 2D array of measured profiles, shaped: (nprofiles, nbins).
- **x_coords** (*ndarray: int*) – x-coordinates of particles, given as coordinates of the reconstructed phase space coordinate system. Shape: (nparts, nprofiles).

nparts

Number of test particles.

Type int

nprofs

Number of profiles (time frames).

Type int

nbins

Number of bins in each profile.

Type int

waterfall

2D array of measured profiles, shaped: (nprofiles, nbins). Negative values of waterfall is set to zero, and the waterfall is normalized.

Type ndarray

xp

x-coordinates of particles, given as coordinates of the reconstructed phase space coordinate system. Shape: (nparts, nprofiles).

Type ndarray

recreated

Recreated waterfall. Directly comparable with *Tomogaphy.waterfall*. Shape: (nprofiles, nbins).

Type ndarray

diff

Discrepancy for phase space reconstruction at each iteration of the reconstruction process.

Type ndarray

property nbins

Number of profile bins defined as @property.

Returns nbins – Number of bins in each profile.

Return type int

property nparts

Number of particles defined as @property.

Returns nparts – Number of test particles.

Return type int

property nprofs

Number of profiles defined as @property.

nprofs: int Number of profiles (time frames).

property waterfall

Waterfall defined as @property.

Returns waterfall – Measured profiles (nprofiles, nbins). waterfall is normalized and its negative values are set to zero.

Return type ndarray

property xp

X-coordinates defined as @property.

Automatically updates *nparts*.

Parameters value (*ndarray*, *None*) – 2D array of tracked particles phase coordinates, given in phase space coordinates as integers. Shape: (nparts, nprofiles).

By setting tomography.xp = None, the saved x-coordinates are deleted.

Returns xp – particles x-coordinates, given as coordinates of the reconstructed phase space coordinate system in integers. shape: (nparts, nprofiles).

Return type ndarray

Raises

- *CoordinateImportError* – Exception: Provided coordinates are invalid.
- *XPOutOfImageWidthError* – Exception: Particle(s) out of image width.

property yp

Y-coordinates defined as @property.

Parameters value (*ndarray*, *None*) – 2D array of tracked particles energy coordinates, given in phase space coordinates as integers. Shape: (nparts, nprofiles).

By setting tomography.yp = None, the saved y-coordinates are deleted.

Returns xp – particles y-coordinates, given as coordinates of the reconstructed phase space coordinate system in integers. shape: (nparts, nprofiles).

Return type ndarray

Raises *CoordinateImportError* – Exception: Provided coordinates are invalid.

2.3.2 tomography Module

Module containing the TomographyCpp class

Author(s) Christoffer Hjertø Grindheim

class `tomo.tomography.tomography.TomographyCpp` (*waterfall*, *x_coords=None*,
y_coords=None)

Bases: `tomo.tomography.__tomography.Tomography`

Class for performing tomographic reconstruction of phase space.

The tomographic routine largely consists of two parts. Projection and back projection. The **back projection** creates a phase space reconstruction based on the measured profiles. The **projection** routine converts from back projection to reconstructed profiles.

By comparing the reconstructed profiles to the measured profiles, adjustments of the weights can be made in order to create a better reconstruction. The number of iterations in this process can be specified by the user.

Parameters

- **waterfall** (*ndarray*) – 2D array of measured profiles, shaped: (nprofiles, nbins).
- **x_coords** (*ndarray*) – x-coordinates of particles, given as coordinates of the reconstructed phase space coordinate system. Shape: (nparts, nprofiles).

nparts

Number of test particles.

Type int

nprofs

Number of profiles (time frames).

Type int

nbins

Number of bins in each profile.

Type int

waterfall

2D array of measured profiles, shaped: (nprofiles, nbins). Negative values of waterfall is set to zero, and the waterfall is normalized.

Type ndarray

xp

x-coordinates of particles, given as coordinates of the reconstructed phase space coordinate system. Shape: (nparts, nprofiles).

Type ndarray

recreated

Recreated waterfall. Directly comparable with *Tomography.waterfall*. Shape: (nprofiles, nbins).

Type ndarray

diff

Discrepancy for phase space reconstruction at each iteration of the reconstruction process.

Type ndarray

run (*niter=20*, *verbose=False*)

Function to perform tomographic reconstruction.

Performs the full reconstruction using C++.

- The discrepancy of each iteration is saved in the objects **diff** variable.
- The particles weights are saved in the objects **weight** variable.
- The reconstructed profiles are saved to the objects **recreated** variable.

Parameters

- **niter** (*int*) – Number of iterations in reconstruction.
- **verbose** (*boolean*) – Flag to indicate that the status of the tomography should be written to stdout. The output is identical to output generated in the original Fortran tomography.

Returns **weight** – 1D array containing the weight of each particle.

Return type ndarray

Raises **CoordinateError** – Exception: X-coordinates is None

run_hybrid (*niter=20, verbose=False*)

Function to perform tomographic reconstruction, implemented as a hybrid between C++ and Python.

Projection and back projection routines are called from C++, the rest is written in python. Kept for reference.

The discrepancy of each iteration is saved in the *diff* array of the object.

The recreated waterfall can be found calling the *recreated* field of the object.

Parameters

- **niter** (*int*) – Number of iterations in reconstruction.
- **verbose** (*boolean*) – Flag to indicate that the status of the tomography should be written to stdout. The output is identical to output generated in the original Fortran tomography.

Returns **weight** – 1D array containing the weight of each particle.

Return type ndarray

Raises

- **CoordinateError** – Exception: X-coordinates is None
- **WaterfallReducedToZero** – Exception: All of reconstructed waterfall reduced to zero.

show_convergence (*counter*)

show_difference (*image, rec_idx, out_dir, counter*)

show_image (*image, out_dir, counter*)

show_iter (*weight, counter, rec_idx=0*)

show_profile (*image, rec_idx, out_dir, counter*)

show_waterfall (*counter*)

2.4 utils Package

2.4.1 physics Module

Module containing fundamental physics formulas.

Author(s) Christoffer Hjertø Grindheim

`tomo.utils.physics.b_to_e(machine)`

Calculates the energy for a particle in a circular machine at dipole field B.

Parameters `machine` (`Machine`) – Stores machine parameters and reconstruction settings.

Returns `energy` – Energy calculated from B-field.

Return type float

`tomo.utils.physics.calc_self_field_coeffs(machine)`

Calculates self-field coefficient for each profile. Needed for calculation of self-fields.

Parameters `machine` (`Machine`) – Stores machine parameters and reconstruction settings.

Returns `Self-field coefficients` – 1D array holding a self-field coefficient for each time frame.

Return type ndarray

`tomo.utils.physics.dphase_low(phase, machine, bunch_phaselength, *args)`

Calculates derivative of the `phase_low` function. The function is needed for estimation of x-coordinate of synchronous particle.

Parameters

- `phase` (`float`) – Phase [rad] where potential energy is calculated.
- `machine` (`Machine`) – Stores machine parameters and reconstruction settings.
- `bunch_phaselength` (`float`) – Length of bunch, given as phase [rad]
- `args` (`None`) – Not used, arg needed for implementation of Newton-Raphson root finder.

`tomo.utils.physics.drf_voltage(phi, machine, rf_turn)`

Calculates derivative of RF voltage from both RF systems seen by particle. Needed by the Newton root finder to calculate `phi0`.

Parameters

- `phi` (`float`) – Phase where voltage is to be calculated.
- `machine` (`Machine`) – Stores machine parameters and reconstruction settings.
- `rf_turn` (`int`) – At which RF turn the calculation should happen.

Returns `Derivative of voltage` – Derivative of voltage from both RF systems, as seen by particle.

Return type float

`tomo.utils.physics.drfvolt_rf1(phi, machine, rf_turn)`

Calculates derivative of RF voltage1 seen by particle. Needed by the Newton root finder to calculate `phi0`.

Parameters

- `phi` (`float`) – Phase where voltage is to be calculated.
- `machine` (`Machine`) – Stores machine parameters and reconstruction settings.
- `rf_turn` (`int`) – At which RF turn the calculation should happen.

Returns Derivative of voltage – Time-derivative of voltage from RF system 1, as seen by particle.

Return type float

`tomo.utils.physics.find_dphase(machine)`

Calculates coefficient needed for drift calculation during tracking. The drift coefficient is calculated for each machine turn.

Parameters machine (*Machine*) – Stores machine parameters and reconstruction settings.

Returns drift factor – 1D array holding drift coefficient for each machine turn.

Return type ndarray

`tomo.utils.physics.find_phi_lower_upper(machine, rf_turn)`

Calculates lower and upper phase of RF voltage for use in estimation of ϕ_0 .

Parameters

- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **rf_turn** (*int*) – At which RF turn the calculation should happen.

Returns

- **phi lower** (*float*) – Lower phase boundary [rad].
- **phi upper** (*float*) – Upper phase boundary [rad].

`tomo.utils.physics.find_synch_phase(machine, rf_turn, phi_lower, phi_upper)`

Uses the Newton-Raphson root finder to estimate the synchronous phase for a particle on the normal orbit.

Parameters

- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **rf_turn** (*int*) – At which RF turn the calculation should happen.
- **phi_lower** – Lower phase boundary
- **phi_upper** – Upper phase boundary

Returns synchronous phase – Synchronous phase given in radians.

Return type float

`tomo.utils.physics.lorenz_beta(machine, rf_turn)`

Calculates Lorentz beta factor (v/c) at a turn.

Parameters

- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **rf_turn** (*int*) – At which RF turn the calculation should happen.

Returns Lorentz beta – The Lorentz beta factor (v/c) at a given turn.

Return type float

`tomo.utils.physics.phase_low(phase, machine, bunch_phaselength, rf_turn)`

Calculates potential energy at phase. Needed for estimation of x-coordinate of synchronous particle.

Parameters

- **phase** (*float*) – Phase [rad] where potential energy is calculated.
- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **bunch_phaselength** (*float*) – Length of bunch, given as phase [rad]

- **rf_turn** (*int*) – At which RF turn the calculation should happen.

`tomo.utils.physics.phase_slip_factor(machine)`

Calculates phase slip factor at each turn.

Parameters `machine` (*Machine*) – Stores machine parameters and reconstruction settings.

Returns `phase slip factor` – 1D array of phase slip factors for each turn.

Return type `ndarray`

`tomo.utils.physics.revolution_freq(machine)`

Calculate revolution frequency for each turn.

Parameters `machine` (*Machine*) – Stores machine parameters and reconstruction settings.

Returns `revolution frequency` – 1D array holding revolution frequency at each machine turn.

Return type `ndarray`

`tomo.utils.physics.rf_voltage(phi, machine, rf_turn)`

Calculates RF voltage from both RF systems seen by particle. Needed by the Newton root finder to calculate `phi0`.

Parameters

- **phi** (*float*) – Phase where voltage is to be calculated.
- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **rf_turn** (*int*) – At which RF turn the calculation should happen.

Returns `voltage` – Voltage from both RF systems, as seen by particle.

Return type `float`

`tomo.utils.physics.rf_voltage_at_phase(phi, vrf1, vrf1dot, vrf2, vrf2dot, h_ratio, phi12, time_at_turn, rf_turn)`

RF voltage formula without calculating the difference in `E0`.

`tomo.utils.physics.rfvolt_rfl(phi, machine, rf_turn)`

Calculates RF voltage1 seen by particle. Needed by the Newton root finder to calculate `phi0`.

Parameters

- **phi** (*float*) – Phase where voltage is to be calculated.
- **machine** (*Machine*) – Stores machine parameters and reconstruction settings.
- **rf_turn** (*int*) – At which RF turn the calculation should happen.

Returns `voltage` – Voltage from RF system 1, as seen by particle.

Return type `float`

`tomo.utils.physics.vrft(vrf, vrf_dot, turn_time)`

Calculates the RF peak voltage at turn `rf_turn` assuming a linear voltage function. Time=0 at machine reference frame.

Parameters

- **vrf** (*float*) – Peak voltage.
- **vrf_dot** (*float*) – Time derivative of RF voltage.
- **time_at_turn** (*float*) – time at turn for which the RF voltage should be calculated.

Returns `RF voltage` – RF voltage at turn.

Return type float

2.4.2 data_treatment Module

Module containing functions for threatement of data.

Author(s) Christoffer Hjertø Grindheim

`tomo.utils.data_treatment.calc_baseline_ftn(waterfall, ref_prof, percent=0.05)`

Function for finding baseline of raw data.

The function is based on the original Fortran program, and uses a percentage of a reference profile in order to find the baseline of the measurements.

Parameters

- **waterfall** (*ndarray*) – Raw-data shaped as waterfall: (nprofiles, nbins).
- **ref_prof** (*int*) – Index of reference profile.
- **percent** (*float, optional, default=0.05*) – A number between 0 and 1 describing the percentage of the reference profile used to find the baseline.

Returns **baseline** – Baseline of reference profile.

Return type float

Raises ***InputError*** – Exception: Raised if percentage is not given as a float between 0 and 1.

`tomo.utils.data_treatment.fit_synch_part_x(profiles)`

Linear fit to estimate the phase coordinate of the synchronous particle. The found phase is returned as a x-coordinate of the phase space coordinate systems in fractions of bins. The estimation is done at the beam reference profile, which is set in the *Machine* object.

Parameters **profiles** (*Profiles*) – Profiles object containing waterfall and information about the measurement.

Returns

- **fitted_synch_part_x** – X coordinate in the phase space coordinate system of the synchronous particle given in bin numbers.
- **lower_bunch_limit** – Estimation of the lower bunch limit in bin numbers. Needed for `write_plotinfo_ftn()` function in order to write the original output format.
- **upper_bunch_limit** – Estimation of the upper bunch limit in bin numbers. Needed for `write_plotinfo_ftn()` function in order to write the original output format.

`tomo.utils.data_treatment.phase_space(tomo, machine, profile=0)`

returns time, energy and phase space density arrays from a reconstruction, requires the homogenous distribution to have been generated by the particles class.

Parameters

- **tomo** (*Tomography*) – Object holding the information about a tomographic reconstruction.
- **machine** (*Machine*) – Object holding information about machine and reconstruction parameters.
- **profile** (*int*) – Index of profile to be reconstructed.

Returns

- **t_range** (*ndarray*) – 1D array containing time axis of reconstructed phase space image.

- **E_range** (*ndarray*) – 1D array containing energy axis of reconstructed phase space image.
- **density** (*ndarray*) – 2D array containing the reconstructed phase space image.

Raises *InputError* – Exception: phase_space function requires automatic phase space generation to have been used.

`tomo.utils.data_treatment.rebin(waterfall, rbn, dtbin=None, synch_part_x=None)`

Rebin waterfall from shape (P, X) to (P, Y). P is the number of profiles, X is the original number of bins, and Y is the number of bins after the re-binning.

The algorithm is based on the rebinning function from the original tomography program.

An array of length N, rebinned with a rebin factor of R will result in a array of length N / R. If N is not dividable on R, the resulting array will have the length (N / R) + 1.

Parameters

- **waterfall** (*ndarray*) – 2D array of raw-data shaped as waterfall. Shape: (nprofiles, nbins).
- **rbn** (*int*) – Rebinning factor.
- **dtbin** (*float*) – Size of profile bins [s]. If provided, the function will return the new size of the bins after rebinning.
- **synch_part_x** (*float*) – x-coordinate of synchronous particle, measured in bins. If provided, the function will return its updated coordinate.

Returns

- **rebinned** (*ndarray*) – Rebinned waterfall
- **dtbin** (*float, optional, default=None*) – If a dtbin has been provided in the arguments, the new size of the profile bins will be returned. Otherwise, None will be returned.
- **synch_part_x** (*float, optional, default=None*) – If a synch_part_x has been provided in the arguments, the new x-coordinate of the synchronous particle in bins will be returned. Otherwise, None will be returned.

2.4.3 assertions Module

Module containing assertion functions with standardised output for tomography.

Author(s) Christoffer Hjertø Grindheim

`tomo.utils.assertions.assert_array_greater(array, limit, error_class, msg="", index_offset=0)`

Assert all array elements are greater than x.

Parameters

- **array** (*ndarray*) – Array to be asserted.
- **array_name** (*string*) – Name of array, for user reference.
- **limit** (*int, float*) – Scalar value array elements should be greater than.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_array_greater_eq(array, limit, error_class, msg="", index_offset=0)`

Assert all array elements are greater than or equal to x.

Parameters

- **array** (*ndarray*) – Array to be asserted.
- **array_name** (*string*) – Name of array, for user reference.
- **limit** (*int, float*) – Scalar value array elements should be greater than or equal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_array_in_range(array, low_lim, up_lim, error_class, msg="",  
                                             index_offset=0)
```

Assert all array elements are within a range [x, y].

Parameters

- **array** (*ndarray*) – Array to be asserted.
- **array_name** (*string*) – Name of array, for user reference.
- **low_lim** (*int, float*) – Lower limit of array elements.
- **up_lim** (*int, float*) – Upper limit of array elements.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_array_less(array, limit, error_class, msg=" ", index_offset=0)
```

Assert all array elements are less than x.

Parameters

- **array** (*ndarray*) – Array to be asserted.
- **array_name** (*string*) – Name of array, for user reference.
- **limit** (*int, float*) – Scalar value array elements should be less than.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_array_less_eq(array, limit, error_class, msg=" ", in-  
                                           dex_offset=0)
```

Assert all array elements are less than or equal to x.

Parameters

- **array** (*ndarray*) – Array to be asserted.
- **array_name** (*string*) – Name of array, for user reference.
- **limit** (*int, float*) – Scalar value array elements should be less than or equal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_array_not_equal(array, array_name, limit, error_class, ex-  
                                             tra_text="")
```

Assert array not equal to X.

This function asserts that not all elements of a value has a scalar value X.

Parameters

- **array** (*ndarray*) – Array to be asserted.

- **array_name** (*string*) – Name of array, for user reference.
- **limit** (*int*, *float*) – Scalar value which the array should be unequal of.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_array_shape_equal (arrays, array_names, demanded_shape,
                                                extra_text="")
```

Assert that two arrays have a given shape.

Parameters

- **arrays** (*tuple*) – tuple should contain (ndarrayX, ndarrayY). ndarrays are the arrays to be asserted.
- **array_names** (*tuple*) – tuple of strings, being names of the arrays to be tested for the users reference.
- **demanded_shape** (*tuple*) – Demanded shape of arrays to be asserted. Tuple: (int, int).
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_equal (var, var_name, limit, error_class, extra_text="")
```

Assert scalar equal to X.

Parameters

- **var** (*int*, *float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int*, *float*) – Limit of which variable should be equal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

```
tomo.utils.assertions.assert_fields (obj, obj_name, needed_fields, error_class, msg="")
```

Assert that object contains all necessary fields.

Parameters

- **obj** (*Object*) – Object to be asserted.
- **obj_name** (*string*) – Name of object, for user reference.
- **needed_fields** (*array like*) – Array containing attributes which object should contain.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **msg** (*string*) – Extra text for error message.

```
tomo.utils.assertions.assert_frame_inputs (frame)
```

Assert that frame parameters are valid, and that raw data will be correctly shaped to waterfall.

Parameters **frame** (*Frame*) – Frame object to be asserted.

Raises [InputError](#) – Exception: An invalid frame parameter has been found.

```
tomo.utils.assertions.assert_greater (var, var_name, limit, error_class, extra_text="")
```

Assert scalar greater than X.

Parameters

- **var** (*int, float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int, float*) – Limit of which variable should be greater than.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_greater_or_equal` (*var, var_name, limit, error_class, extra_text=""*)

Assert scalar greater than or equal to X.

Parameters

- **var** (*int, float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int, float*) – Limit of which variable should be greater than or equal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_index_ok` (*index, index_limit, wrap_around=False*)

Assert index is valid.

Assert that index is within bounds [0, index limit). Wrap_around for negative indices can be enabled. If this is the case, index -1 is points at the last index.

Parameters

- **index** (*int*) – Index to be tested.
- **limit** (*index*) – Maximum index + 1
- **wrap_around** (*bool, optional, default=False*) – Set to true to enable wrap around for negative indices.

Returns **index** – Asserted index from 0 to index_limit-1.

Return type `int`

Raises

- **IndexError** – Exception: Index out of bounds.
- **NegativeIndexError** – Exception: Negative index given without setting `wrap_around=True`

`tomo.utils.assertions.assert_inrange` (*var, var_name, low_lim, up_lim, error_class, extra_text=""*)

Assert scalar is in range of [x, y].

Parameters

- **var** (*int, float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **low_limit** (*int, float*) – Lower limit of variable.
- **up_limit** (*int, float*) – Upper limit of variable.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_less` (*var*, *var_name*, *limit*, *error_class*, *extra_text*="")
 Assert scalar less than X.

Parameters

- **var** (*int*, *float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int*, *float*) – Limit of which variable should be less than.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_less_or_equal` (*var*, *var_name*, *limit*, *error_class*, *extra_text*="")
 Assert scalar less than or equal to X.

Parameters

- **var** (*int*, *float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int*, *float*) – Limit of which variable should be less than or equal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_machine_input` (*machine*)
 Assert that input parameters for a machine object is valid.

Parameters *machine* (*Machine*) – Machine object to be asserted.

Raises

- **MachineParameterError** – Exception: Invalid value found for machine parameter.
- **SpaceChargeParameterError** – Exception: Invalid value found for machine parameter regarding self-field measurements.

`tomo.utils.assertions.assert_not_equal` (*var*, *var_name*, *limit*, *error_class*, *extra_text*="")
 Assert scalar unequal to X.

Parameters

- **var** (*int*, *float*) – Variable to be asserted.
- **var_name** (*string*) – Name of variable, for users reference.
- **limit** (*int*, *float*) – Limit of which variable should be unequal to.
- **error_class** (*Exception*) – Error class to be raised if assertion fails.
- **extra_text** (*string*) – Extra text to be written after standard error message.

`tomo.utils.assertions.assert_only_valid_particles` (*xp*, *n_bins*, *msg*="")
 Assert all particles are within the image width.

An `InvalidParticleError` is raised if the trajectory of one or more particles goes outside of the image width.

Parameters

- **xp** (*ndarray*) – Array of particles.
- **n_bins** (*int*) – Number of bins in a profile measurement (image width).
- **msg** (*string*) – Extra text for error message.

Raises **`InvalidParticleError`** – Exception: One or mor particles has left the image.

`tomo.utils.assertions.assert_var_not_none` (*var*, *var_name*, *error_class*)

2.4.4 exceptions Module

exception `tomo.utils.exceptions.ArgumentError`

Bases: `Exception`

exception `tomo.utils.exceptions.ArrayElementsNotEqualError`

Bases: `Exception`

exception `tomo.utils.exceptions.ArrayLengthError`

Bases: `Exception`

exception `tomo.utils.exceptions.CoordinateError`

Bases: `Exception`

exception `tomo.utils.exceptions.CoordinateImportError`

Bases: `Exception`

exception `tomo.utils.exceptions.EnergyBinningError`

Bases: `Exception`

exception `tomo.utils.exceptions.EnergyLimitsError`

Bases: `Exception`

exception `tomo.utils.exceptions.FilteredProfilesError`

Bases: `Exception`

exception `tomo.utils.exceptions.InputError`

Bases: `Exception`

exception `tomo.utils.exceptions.InvalidParticleError`

Bases: `Exception`

exception `tomo.utils.exceptions.LibraryNotFound`

Bases: `Exception`

exception `tomo.utils.exceptions.MachineParameterError`

Bases: `Exception`

exception `tomo.utils.exceptions.MapCreationError`

Bases: `Exception`

exception `tomo.utils.exceptions.NegativeIndexError`

Bases: `IndexError`

exception `tomo.utils.exceptions.OverMaxIterationsError`

Bases: `Exception`

exception `tomo.utils.exceptions.PhaseLimitsError`

Bases: `Exception`

exception `tomo.utils.exceptions.PhaseSpaceReducedToZeroes`

Bases: `Exception`

exception `tomo.utils.exceptions.ProfileChargeNotCalculated`

Bases: `Exception`

exception `tomo.utils.exceptions.RawDataImportError`

Bases: `Exception`

```

exception tomo.utils.exceptions.RebinningError
    Bases: Exception

exception tomo.utils.exceptions.SelfFieldTrackingError
    Bases: Exception

exception tomo.utils.exceptions.SpaceChargeParameterError
    Bases: Exception

exception tomo.utils.exceptions.TrackingError
    Bases: Exception

exception tomo.utils.exceptions.UnequalArrayShapes
    Bases: Exception

exception tomo.utils.exceptions.ValuesOutOfBrackets
    Bases: Exception

exception tomo.utils.exceptions.WaterfallError
    Bases: Exception

exception tomo.utils.exceptions.WaterfallReducedToZero
    Bases: Exception

exception tomo.utils.exceptions.XPOutOfImageWidthError
    Bases: Exception

```

2.4.5 tomo_input Module

Module containing functions for handling input from Fortran style text files.

Author(s) Christoffer Hjertø Grindheim

```

class tomo.utils.tomo_input.Frames(framecount, framelength, skip_frames, skip_bins_start,
                                   skip_bins_end, rebin, dtbin, raw_data_path=)
    Bases: object

```

Class for storing raw data and information on how to treat them, based on a Fortran style input file.

Parameters

- **framecount** (*int*) – Number of time frames.
- **framelength** (*int*) – Number of bins in a time frame.
- **skip_frames** (*int*) – Number of time frames to ignore from the beginning of the raw input file.
- **skip_bins_start** (*int*) – Subtract this number of bins from start of the raw input.
- **skip_bins_end** (*int*) – Subtract this number of bins from end of the raw input.
- **rebin** (*int*) – Rebinning factor - Number of frame bins to rebin into one profile bin.
- **dtbin** (*float*) – Size of profile bins (sampling size) [s].
- **raw_data_path** (*string, optional, default=''*) – Path to file holding raw data for programmers reference.

raw_data

Measured raw data as read from text file.

nframes

Number of measured time frames.

Type int

nbins_frame

Number of bins in a measured time frame.

Type int

skip_frames

Number of time frames to ignore from the beginning of the raw input file.

Type int

skip_bins_start

Subtract this number of bins from start of the raw input.

Type int

skip_bins_end

Subtract this number of bins from end of the raw input.

Type int

rebin

Rebinning factor, the number of frame bins to rebin into one profile bin.

Type int

sampling_time

Size of profile bins [s].

Type float

raw_data_path

Path to file holding raw data for programmers reference.

Type string

nbins ()

Function to calculate number of bins in profiles. This number should be used when creating a machine object.

Returns nbins – Number of bins in profiles.

Return type int

nprofs ()

Function for calculating the number of profiles. This number should be used when creating a machine object.

Returns nprofiles – Number of profiles.

Return type int

property raw_data

Raw data defined as a @property.

Holds assertions for validity of raw data.

Parameters in_raw_data (*ndarray*) – 1D array holding raw data as a direct read from text file.

Returns raw_data – Copy of array holding raw data as a direct read from text file. None is returned if object holds no raw data.

Return type ndarray, none

Raises `RawDataImportError` – Exception: Raised if raw data is not iterable or has an unexpected length.

to_waterfall (*raw_data*)

Function to convert from raw data to waterfall for use in reconstruction. The waterfall will be shaped based on the settings found in the `Frames` object.

Shape of waterfall is generated based on parameters of the `Frames` object.

Parameters `raw_data` (*ndarray*) – 1D array holding all measured raw data. The function works on a copy of the given `raw_data`.

Returns `waterfall` – Raw-data shaped as waterfall (nprofiles, nbins).

Return type `ndarray`

Raises `RawDataImportError` – Exception: Raised if something is wrong with the raw data.

`tomo.utils.tomo_input.get_user_input()`

Function to let user provide input as a text file either as file path or as stdin.

If a filepath is given through the **system arguments**, the path to the output directory can be given as the second argument. The specified output path will substitute the output path read from the input file. A failed path to the measured data, or the measured data itself can be given in the input file.

If the input is provided via **stdin**, the measured data must be stored in the same input file.

Returns `tomo_input` – Tuple holding machine and reconstruction parameters as a list of strings directly read from text file and `ndarray` containing raw data (parameters, raw data).

Return type tuple (list, `ndarray`)

`tomo.utils.tomo_input.raw_data_to_profiles(waterfall, machine, rbn, sampling_time, synch_part_x=None)`

Converts from waterfall of untreated data, to waterfall ready for tomography. The input waterfall is copied, and the treated waterfall is saved to a profiles object.

The data treatment in this function is the same as in the original original tomography program:

- Subtracts baseline from measurements
- Re-bins profiles
- Creates Profiles object to store waterfall
- Negative values of waterfall is set to zero.

NB: dtbin and synch_part_x of the provided machine object will be updated after re-binning.

Parameters

- **waterfall** (*ndarray*) – Raw-data shaped as waterfall (nprofiles, nbins).
- **machine** (`Machine`) – Machine object holding machine and reconstruction parameters.
- **rbn** (*int*) – Rebinning factor, the number of frame bins to rebin into one profile bin.
- **sampling_time** (*float*) – Size of profile bins [s].
- **synch_part_x** (*float, optional, default=None*) – X-coordinate of synchronous particle. If not given as argument, the `machine.synch_part_x` will be used.

Returns `profile` – Profiles object holding the waterfall and information about the measurements.

Return type `Profiles`

`tomo.utils.tomo_input.txt_input_to_machine(input_array)`

Function converts the content of an input file and uses this to generate a machine object. The input file is given as a list holding one line of the file in each element. The list should contain a direct read from a Fortran styled input file, where all lines should be included.

Parameters `input_array` (*list*) – Array containing each line of an input file as an element of the array. This should be a direct read from a tomography text file.

Returns `machine` – Machine object containing parameters for the machine and the tomographic reconstruction.

Return type *Machine*

Raises *InputError* – Exception: Error in input array

2.4.6 tomo_output Module

Module containing functions for handling output from tomography programs.

Every function ending on 'ftn' creates an output equal original Fortran program.

Author(s) Christoffer Hjertø Grindheim

`tomo.utils.tomo_output.create_phase_space_image(xp, yp, weight, n_bins, recprof)`

Convert from weighted particles to phase-space image.

The output is equal to the phase space image created in the original version.

Parameters

- `xp` (*ndarray*) – 2D array containing the x coordinates of every particle at every time frame. Must be given in coordinates of the phase space coordinate system as integers. Shape: (N, M), where N is the number of particles and M is the number of profiles.
- `yp` (*ndarray*) – 2D array containing the y coordinates of every particle at every time frame. Must be given in coordinates of the phase space coordinate system as integers. Shape: (N, M), where N is the number of particles and M is the number of profiles.
- `weight` (*ndarray*) – 1D array containing the weight of each particle.
- `n_bins` (*int*) – Number of bins in a profile measurement.
- `recprof` (*int*) – Index of reconstructed profile.

Returns `phase_space` – Phase space presented as 2D array with shape (N, N), where N is the number of bins in a profile. This phase space image has the same format as from the original program.

Return type *ndarray*

`tomo.utils.tomo_output.print_tracking_status_ftn(ref_prof, to_profile)`

Write output for particle tracking in the original format. Since the original algorithm is somewhat different, the **output concerning lost particles is not valid**. Meanwhile, the format it is needed by the tomoscope application. Profile numbers are added by one in order to compensate for differences in Python and Fortran indexing. Fortran counts from one, Python counts from 0. This function is used in the tracking algorithm.

Parameters

- `recprof` (*int*) – Index of profile to be reconstructed.
- `to_profile` (*int*) – Profile to which the algorithm is currently tracking towards.

`tomo.utils.tomo_output.save_difference_ftn(diff, output_path, recprof)`

Write reconstruction discrepancy to text file with original format.

Parameters

- **diff** (*ndarray*) – 1D array containing the discrepancy for the phase space at each iteration of the reconstruction.
- **output_dir** (*string*) – Path to output directory.
- **recprof** (*int*) – Index of profile to be saved.

`tomo.utils.tomo_output.save_phase_space_ftn(image, recprof, output_path)`

Save phase-space image in a tomoscope format.

Parameters

- **image** (*ndarray*) – 2D array holding the weight of each cell of the recreated phase-space.
- **recprof** (*int*) – Index of reconstructed profile.
- **output_dir** (*string*) – Path to the output directory.

`tomo.utils.tomo_output.save_profile_ftn(profiles, recprof, output_dir)`

Write phase-space image to text-file in the original format. The name of the file will be profileXXX.data, where XXX is the index of the time frame to be reconstructed counting from one.

Parameters

- **profiles** (*ndarray*) – Profile measurements as a 2D array with the shape: (nprofiles, nbins).
- **recprof** (*int*) – Index of profile to be saved.
- **output_dir** (*string*) – Path to output directory.

`tomo.utils.tomo_output.save_self_volt_profile_ftn(self_fields, output_dir)`

Write self volts to text file in tomoscope format.

Parameters

- **self_fields** (*ndarray*) – Calculated self-field voltages.
- **output_dir** (*string*) – Path to output directory.

`tomo.utils.tomo_output.show(image, diff, recprof)`

Nice presentation of reconstruction.

Parameters

- **Image** (*ndarray*) – Recreated phase-space image. Shape: (N, N), where N is the number of profile bins.
- **Diff** (*ndarray*) – 1D array containing discrepancies for each iteration of reconstruction.
- **recprof** (*ndarray*) – 1D array containing the measured profile to be reconstructed.

`tomo.utils.tomo_output.write_plotinfo_ftn(machine, particles, profile_charge)`

Creates string of plot info needed for the original output for the tomography program.

Parameters

- **machine** (*Machine*) – Object containing machine parameters.
- **particles** (*Particles*) – Object containing particle distribution and information about the phase space reconstruction.
- **profile_charge** (*float*) – Total charge of a reference profile.

Returns **plot_info** – String containing information needed by the tomoscope application. The returned string has the same format as in the original version.

Return type string

2.4.7 tomo_run Module

Module containing function for full Fortran style tomographic reconstruction.

Author(s) Christoffer Hjertø Grindheim

`tomo.utils.tomo_run.run_file` (*file*, *reconstruct_profile=None*, *verbose=False*)

Function to perform full reconstruction based on the original algorithm.

Parameters

- **file** (*string*) – Path to input file.
- **reconstruct_profile** (*int, optional, default=None*) – Profile to be reconstructed. If not provided, machine.filmstart will be reconstructed.
- **verbose** (*boolean, optional, default=False*) – If set to True, the output similar to the one generated by the original program will be printed to stdout.

Returns

- **Phase axis** (*ndarray*) – 1D array containing time axis of reconstructed phase space image.
- **Energy axis** (*ndarray*) – 1D array containing energy axis of reconstructed phase space image.
- **density** (*ndarray*) – 2D array containing the reconstructed phase space image.

Example

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import tomo.utils.run_tomo as tomorun
>>>
>>> filepath = '../my/favourite/input.dat'
>>> tRange, ERange, density = tomorun.run_file(filepath)
>>>
>>> vmin = np.min(density[density>0])
>>> vmax = np.max(density)
>>> plt.contourf(tRange*1E9, ERange/1E6, density.T,
>>>              levels=np.linspace(vmin, vmax, 50), cmap='Oranges')
>>> plt.xlabel('dt (ns)')
>>> plt.ylabel('dE (MeV)')
>>> plt.show()
```

2.5 cpp_routines Package

2.5.1 tomlib_wrappers Module

Module containing Python wrappers for C++ functions.

Should only be used by advanced users.

Author(s) Christoffer Hjertø Grindheim

`tomo.cpp_routines.tomolib_wrappers.back_project` (*weights*, *flat_points*, *flat_profiles*,
nparts, *nprofs*)

Wrapper for back projection routine written in C++. Used in the *tomography* module.

Parameters

- **weights** (*ndarray*) – 1D array containing the weight of each particle.
- **flat_points** (*ndarray*) – 2D array containing particle coordinates as integers, pointing at flattened versions of the waterfall. Shape: (nparts, nprofiles)
- **flat_profiles** (*ndarray*) – 1D array containing a flattened waterfall.
- **nparts** (*int*) – Number of tracked particles.
- **nprofs** (*int*) – Number of profiles.

Returns **weights** – 1D array containing the **new weight** of each particle.

Return type *ndarray*

`tomo.cpp_routines.tomolib_wrappers.drift` (*denergy*, *dphi*, *drift_coef*, *npart*, *turn*, *up=True*)

Wrapper for C++ drift function.

Particle drift for **one** machine turn

Used in the *tracking* module.

Parameters

- **denergy** (*ndarray*) – 1D array holding the energy difference relative to the synchronous particle for each particle at a turn.
- **dphi** (*ndarray*) – 1D array holding the phase difference relative to the synchronous particle for each particle at a turn.
- **drift_coef** (*ndarray*) – 1D array of drift coefficient at each machine turn.
- **npart** (*int*) – The number of tracked particles.
- **turn** (*int*) – The current machine turn.
- **up** (*boolean, optional, default=True*) – Direction of tracking. Up=True tracks towards last machine turn, up=False tracks toward first machine turn.

Returns **dphi** – 1D array containing the new phase for each particle after drifting for a machine turn.

Return type *ndarray*

`tomo.cpp_routines.tomolib_wrappers.kick` (*machine*, *denergy*, *dphi*, *rfv1*, *rfv2*, *npart*, *turn*,
up=True)

Wrapper for C++ kick function.

Particle kick for **one** machine turn.

Used in the *tracking* module.

Parameters

- **machine** (*Machine*) – Object holding machine parameters.
- **denergy** (*ndarray*) – 1D array holding the energy difference relative to the synchronous particle for each particle at a turn.
- **dphi** (*ndarray*) – 1D array holding the phase difference relative to the synchronous particle for each particle at a turn.

- **rfv1** (*ndarray*) – 1D array holding the radio frequency voltage at RF station 1 for each turn, multiplied with the charge state of the particles.
- **rfv2** (*ndarray*) – 1D array holding the radio frequency voltage at RF station 2 for each turn, multiplied with the charge state of the particles.
- **npart** (*int*) – The number of tracked particles.
- **turn** (*int*) – The current machine turn.
- **up** (*boolean, optional, default=True*) – Direction of tracking. Up=True tracks towards last machine turn, up=False tracks toward first machine turn.

Returns **denergy** – 1D array containing the new energy of each particle after voltage kick.

Return type *ndarray*

```
tomo.cpp_routines.tomolib_wrappers.kick_and_drift(xp, yp, denergy, dphi, rfv1, rfv2,  
                                                    rec_prof, nturns, nparts, *args, ma-  
                                                    chine=None, fn_out=False)
```

Wrapper for full kick and drift algorithm written in C++.

Tracks all particles from the time frame to be recreated, trough all machine turns.

Used in the *tracking* module.

Parameters

- **xp** (*ndarray*) – 2D array large enough to hold the phase of each particle at every time frame. Shape: (nprofiles, nparts)
- **yp** (*ndarray*) – 2D array large enough to hold the energy of each particle at every time frame. Shape: (nprofiles, nparts)
- **denergy** (*ndarray*) – 1D array holding the energy difference relative to the synchronous particle for each particle the initial turn.
- **dphi** (*ndarray*) – 1D array holding the phase difference relative to the synchronous particle for each particle the initial turn.
- **rfv1** (*ndarray*) – Array holding the radio frequency voltage at RF station 1 for each turn, multiplied with the charge state of the particles.
- **rfv2** (*ndarray*) – Array holding the radio frequency voltage at RF station 2 for each turn, multiplied with the charge state of the particles.
- **rec_prof** (*int*) – Index of profile to be reconstructed.
- **nturns** (*int*) – Total number of machine turns.
- **nparts** (*int*) – The number of particles.
- **args** (*tuple*) – Arguments can be provided via the args if a machine object is not to be used. In this case, the args should be:
 - phi0
 - deltaE0
 - omega_rev0
 - drift_coef
 - phi12
 - h_ratio
 - dturns

The args will not be used if a Machine object is provided.

- **machine** (*Machine*, *optional*, *default=False*) – Object containing machine parameters.
- **ftn_out** (*boolean*, *optional*, *default=False*) – Flag to enable printing of status of tracking to stdout. The format will be similar to the Fortran version. Note that the **information regarding lost particles are not valid**.

Returns

- **xp** (*ndarray*) – 2D array holding every particles coordinates in phase [rad] at every time frame. Shape: (nprofiles, nparts)
- **yp** (*ndarray*) – 2D array holding every particles coordinates in energy [eV] at every time frame. Shape: (nprofiles, nparts)

`tomo.cpp_routines.tomolib_wrappers.project` (*recreated*, *flat_points*, *weights*, *nparts*, *nprofs*, *nbins*)

Wrapper projection routine written in C++. Used in the *tomography* module.

Parameters

- **recreated** (*ndarray*) – 2D array with the shape of the waterfall to be recreated, initiated as zero. Shape: (nprofiles, nbins)
- **flat_points** (*ndarray*) – 2D array containing particle coordinates as integers, pointing at flattened versions of the waterfall. Shape: (nparts, nprofiles)
- **weights** (*ndarray*) – 1D array containing the weight of each particle.
- **nparts** (*int*) – Number of tracked particles.
- **nprofs** (*int*) – Number of profiles.
- **nbins** (*int*) – Number of bins in profiles.

Returns **recreated** – 2D array containing the projected profiles as waterfall. Shape: (nprofiles, nbins)

Return type *ndarray*

`tomo.cpp_routines.tomolib_wrappers.reconstruct` (*xp*, *waterfall*, *niter*, *nbins*, *npart*, *nprof*, *verbose*)

Wrapper for full reconstruction in C++. Used in the *tomography* module.

Parameters

- **xp** (*ndarray*) – 2D array containing the coordinate of each particle at each time frame. Coordinates should given be as integers of the phase space coordinate system. shape: (nparts, nprofiles).
- **waterfall** (*ndarray*) – 2D array containing measured profiles as waterfall. Shape: (nprofiles, nbins).
- **niter** (*int*) – Number of iterations in the reconstruction process.
- **nbins** (*int*) – Number of bins in a profile.
- **npart** (*int*) – Number of tracked particles.
- **nprof** (*int*) – Number of profiles.
- **verbose** (*boolean*) – Flag to indicate that the tomography routine should broadcast its status to stdout. The output is identical to the output from the Fortran version.

Returns

- **weights** (*ndarray*) – 1D array containing the weight of each particle.
- **discr** (*ndarray*) – 1D array containing discrepancy at each iteration of the reconstruction.
- **recreated** (*ndarray*) – 2D array containing the projected profiles as waterfall. Shape: (nprofiles, nbins)

2.6 compile Module

2.6.1 compile Module

Module containing the Tracking class.

Author(s) Christoffer Hjertø Grindheim

`tomo.compile.main()`

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

t

- `tomo.compile`, 42
- `tomo.cpp_routines.tomolib_wrappers`, 38
- `tomo.data.profiles`, 3
- `tomo.tomography.__tomography`, 19
- `tomo.tomography.tomography`, 21
- `tomo.tracking.__tracking`, 10
- `tomo.tracking.machine`, 5
- `tomo.tracking.particles`, 12
- `tomo.tracking.phase_space_info`, 15
- `tomo.tracking.tracking`, 16
- `tomo.utils.assertions`, 27
- `tomo.utils.data_treatment`, 26
- `tomo.utils.exceptions`, 32
- `tomo.utils.physics`, 23
- `tomo.utils.tomo_input`, 33
- `tomo.utils.tomo_output`, 36
- `tomo.utils.tomo_run`, 38

A

ArgumentError, 32
 ArrayElementsNotEqualError, 32
 ArrayLengthError, 32
 assert_array_greater() (in module *tomo.utils.assertions*), 27
 assert_array_greater_eq() (in module *tomo.utils.assertions*), 27
 assert_array_in_range() (in module *tomo.utils.assertions*), 28
 assert_array_less() (in module *tomo.utils.assertions*), 28
 assert_array_less_eq() (in module *tomo.utils.assertions*), 28
 assert_array_not_equal() (in module *tomo.utils.assertions*), 28
 assert_array_shape_equal() (in module *tomo.utils.assertions*), 29
 assert_equal() (in module *tomo.utils.assertions*), 29
 assert_fields() (in module *tomo.utils.assertions*), 29
 assert_frame_inputs() (in module *tomo.utils.assertions*), 29
 assert_greater() (in module *tomo.utils.assertions*), 29
 assert_greater_or_equal() (in module *tomo.utils.assertions*), 30
 assert_index_ok() (in module *tomo.utils.assertions*), 30
 assert_inrange() (in module *tomo.utils.assertions*), 30
 assert_less() (in module *tomo.utils.assertions*), 30
 assert_less_or_equal() (in module *tomo.utils.assertions*), 31
 assert_machine_input() (in module *tomo.utils.assertions*), 31
 assert_not_equal() (in module *tomo.utils.assertions*), 31
 assert_only_valid_particles() (in module *tomo.utils.assertions*), 31
 assert_var_not_none() (in module

tomo.utils.assertions), 32

B

b0 (*tomo.tracking.machine.Machine* attribute), 6
 b_to_e() (in module *tomo.utils.physics*), 23
 back_project() (in module *tomo.cpp_routines.tomolib_wrappers*), 38
 bdot (*tomo.tracking.machine.Machine* attribute), 6
 beam_ref_frame (*tomo.tracking.machine.Machine* attribute), 8
 bending_rad (*tomo.tracking.machine.Machine* attribute), 6
 beta0 (*tomo.tracking.machine.Machine* attribute), 9

C

calc_baseline_ftn() (in module *tomo.utils.data_treatment*), 26
 calc_profilecharge() (*tomo.data.profiles.Profiles* method), 4
 calc_self_field_coeffs() (in module *tomo.utils.physics*), 23
 calc_self_fields() (*tomo.data.profiles.Profiles* method), 4
 calc_xorigin() (*tomo.tracking.phase_space_info.PhaseSpaceInfo* method), 15
 CoordinateError, 32
 CoordinateImportError, 32
 coordinates_dphi_denergy (*tomo.tracking.particles.Particles* attribute), 12
 coordinates_dphi_denergy (*tomo.tracking.particles.Particles* property), 12
 create_phase_space_image() (in module *tomo.utils.tomo_output*), 36

D

dEbin (*tomo.tracking.particles.Particles* attribute), 12
 dEbin (*tomo.tracking.phase_space_info.PhaseSpaceInfo* attribute), 15
 deltaE0 (*tomo.tracking.machine.Machine* attribute), 9
 demax (*tomo.tracking.machine.Machine* attribute), 6
 diff (*tomo.tomography.__tomography.Tomography* attribute), 19

- `diff` (*tomo.tomography.tomography.TomographyCpp attribute*), 21
- `dphase_low()` (*in module tomo.utils.physics*), 23
- `drf_voltage()` (*in module tomo.utils.physics*), 23
- `drfvolt_rfl()` (*in module tomo.utils.physics*), 23
- `drift()` (*in module tomo.cpp_routines.tomolib_wrappers*), 39
- `drift_coef` (*tomo.tracking.machine.Machine attribute*), 9
- `dsprofiles` (*tomo.data.profiles.Profiles attribute*), 4
- `dtbin` (*tomo.tracking.machine.Machine attribute*), 8
- `dturns` (*tomo.tracking.machine.Machine attribute*), 6
- ## E
- `e0` (*tomo.tracking.machine.Machine attribute*), 9
- `e_rest` (*tomo.tracking.machine.Machine attribute*), 7
- `enable_fortran_output()` (*tomo.tracking.__tracking.ParticleTracker method*), 11
- `enable_self_fields()` (*tomo.tracking.__tracking.ParticleTracker method*), 11
- `EnergyBinningError`, 32
- `EnergyLimitsError`, 32
- `eta0` (*tomo.tracking.machine.Machine attribute*), 9
- ## F
- `filmstart` (*tomo.tracking.machine.Machine attribute*), 8
- `filmstep` (*tomo.tracking.machine.Machine attribute*), 8
- `filmstop` (*tomo.tracking.machine.Machine attribute*), 8
- `filter_lost()` (*in module tomo.tracking.particles*), 13
- `FilteredProfilesError`, 32
- `find_binned_phase_energy_limits()` (*tomo.tracking.phase_space_info.PhaseSpaceInfo method*), 16
- `find_dEbin()` (*tomo.tracking.phase_space_info.PhaseSpaceInfo method*), 16
- `find_dphase()` (*in module tomo.utils.physics*), 24
- `find_phi_lower_upper()` (*in module tomo.utils.physics*), 24
- `find_synch_phase()` (*in module tomo.utils.physics*), 24
- `fit_synch_part_x()` (*in module tomo.utils.data_treatment*), 26
- `fortran_flag` (*tomo.tracking.__tracking.ParticleTracker attribute*), 11
- `fortran_flag` (*tomo.tracking.tracking.Tracking attribute*), 17
- `fortran_flag()` (*tomo.tracking.__tracking.ParticleTracker property*), 11
- `Frames` (*class in tomo.utils.tomo_input*), 33
- `full_pp_flag` (*tomo.tracking.machine.Machine attribute*), 8
- ## G
- `g_coupling` (*tomo.tracking.machine.Machine attribute*), 7
- `get_user_input()` (*in module tomo.utils.tomo_input*), 35
- ## H
- `h_num` (*tomo.tracking.machine.Machine attribute*), 7
- `h_ratio` (*tomo.tracking.machine.Machine attribute*), 7
- `homogeneous_distribution()` (*tomo.tracking.particles.Particles method*), 13
- ## I
- `imax` (*tomo.tracking.particles.Particles attribute*), 12
- `imax` (*tomo.tracking.phase_space_info.PhaseSpaceInfo attribute*), 15
- `imin` (*tomo.tracking.particles.Particles attribute*), 12
- `imin` (*tomo.tracking.phase_space_info.PhaseSpaceInfo attribute*), 15
- `InputError`, 32
- `InvalidParticleError`, 32
- ## J
- `jmax` (*tomo.tracking.particles.Particles attribute*), 12
- `jmax` (*tomo.tracking.phase_space_info.PhaseSpaceInfo attribute*), 15
- `jmin` (*tomo.tracking.particles.Particles attribute*), 12
- `jmin` (*tomo.tracking.phase_space_info.PhaseSpaceInfo attribute*), 15
- ## K
- `kick()` (*in module tomo.cpp_routines.tomolib_wrappers*), 39
- `kick_and_drift()` (*in module tomo.cpp_routines.tomolib_wrappers*), 40
- `kick_and_drift()` (*tomo.tracking.tracking.Tracking method*), 17
- `kick_and_drift_self()` (*tomo.tracking.tracking.Tracking method*), 17
- ## L
- `LibraryNotFound`, 32
- `load_fitted_synch_part_x_ftn()` (*tomo.tracking.machine.Machine method*), 9
- `lorenz_beta()` (*in module tomo.utils.physics*), 24

M

Machine (class in *tomo.tracking.machine*), 5
 machine (*tomo.data.profiles.Profiles* attribute), 3
 machine (*tomo.tracking.__tracking.ParticleTracker* attribute), 10
 machine (*tomo.tracking.phase_space_info.PhaseSpaceInfo* attribute), 15
 machine (*tomo.tracking.tracking.Tracking* attribute), 16
 machine_ref_frame (*tomo.tracking.machine.Machine* attribute), 8
 MachineParameterError, 32
 main() (in module *tomo.compile*), 42
 MapCreationError, 32
 max_dt (*tomo.tracking.machine.Machine* attribute), 7
 mean_orbit_rad (*tomo.tracking.machine.Machine* attribute), 6
 min_dt (*tomo.tracking.machine.Machine* attribute), 7

N

nbins (*tomo.tomography.__tomography.Tomography* attribute), 19
 nbins (*tomo.tomography.tomography.TomographyCpp* attribute), 21
 nbins (*tomo.tracking.machine.Machine* attribute), 7
 nbins() (*tomo.tomography.__tomography.Tomography* property), 19
 nbins() (*tomo.tracking.machine.Machine* property), 10
 nbins() (*tomo.utils.tomo_input.Frames* method), 34
 nbins_frame (*tomo.utils.tomo_input.Frames* attribute), 34
 NegativeIndexError, 32
 nframes (*tomo.utils.tomo_input.Frames* attribute), 33
 niter (*tomo.tracking.machine.Machine* attribute), 8
 nparts (*tomo.tomography.__tomography.Tomography* attribute), 19
 nparts (*tomo.tomography.tomography.TomographyCpp* attribute), 21
 nparts() (*tomo.tomography.__tomography.Tomography* property), 20
 nprofiles (*tomo.tracking.machine.Machine* attribute), 7
 nprofs (*tomo.tomography.__tomography.Tomography* attribute), 19
 nprofs (*tomo.tomography.tomography.TomographyCpp* attribute), 21
 nprofs() (*tomo.tomography.__tomography.Tomography* property), 20
 nprofs() (*tomo.utils.tomo_input.Frames* method), 34
 nturns (*tomo.tracking.__tracking.ParticleTracker* attribute), 11
 nturns (*tomo.tracking.tracking.Tracking* attribute), 16

O

omega_rev0 (*tomo.tracking.machine.Machine* at-

tribute), 9

output_dir (*tomo.tracking.machine.Machine* attribute), 8

OverMaxIterationsError, 32

P

Particles (class in *tomo.tracking.particles*), 12
 particles (*tomo.tracking.__tracking.ParticleTracker* attribute), 11
 particles (*tomo.tracking.tracking.Tracking* attribute), 16
 ParticleTracker (class in *tomo.tracking.__tracking*), 10
 phase_low() (in module *tomo.utils.physics*), 24
 phase_slip_factor() (in module *tomo.utils.physics*), 25
 phase_space() (in module *tomo.utils.data_treatment*), 26
 PhaseLimitsError, 32
 PhaseSpaceInfo (class in *tomo.tracking.phase_space_info*), 15
 PhaseSpaceReducedToZeroes, 32
 phi0 (*tomo.tracking.machine.Machine* attribute), 9
 phi12 (*tomo.tracking.machine.Machine* attribute), 7
 phiwrap (*tomo.data.profiles.Profiles* attribute), 4
 physical_to_coords() (in module *tomo.tracking.particles*), 14
 pickup_sensitivity (*tomo.tracking.machine.Machine* attribute), 7
 print_tracking_status_ftn() (in module *tomo.utils.tomo_output*), 36
 profile_charge (*tomo.data.profiles.Profiles* attribute), 4
 ProfileChargeNotCalculated, 32
 Profiles (class in *tomo.data.profiles*), 3
 project() (in module *tomo.cpp_routines.tomolib_wrappers*), 41

Q

q (*tomo.tracking.machine.Machine* attribute), 7

R

raw_data (*tomo.utils.tomo_input.Frames* attribute), 33
 raw_data() (*tomo.utils.tomo_input.Frames* property), 34
 raw_data_path (*tomo.utils.tomo_input.Frames* attribute), 34
 raw_data_to_profiles() (in module *tomo.utils.tomo_input*), 35
 RawDataImportError, 32
 ready_for_tomography() (in module *tomo.tracking.particles*), 14
 rebin (*tomo.utils.tomo_input.Frames* attribute), 34
 rebin() (in module *tomo.utils.data_treatment*), 27

RebinningError, 32
 reconstruct() (in module *tomo.cpp_routines.tomolib_wrappers*), 41
 recreated(*tomo.tomography.__tomography.Tomography* attribute), 19
 recreated(*tomo.tomography.tomography.TomographyCpp* attribute), 21
 revolution_freq() (in module *tomo.utils.physics*), 25
 rf_voltage() (in module *tomo.utils.physics*), 25
 rf_voltage_at_phase() (in module *tomo.utils.physics*), 25
 rfvolt_rfl() (in module *tomo.utils.physics*), 25
 run() (*tomo.tomography.tomography.TomographyCpp* method), 21
 run_file() (in module *tomo.utils.tomo_run*), 38
 run_hybrid() (*tomo.tomography.tomography.TomographyCpp* method), 22

S

sampling_time (*tomo.data.profiles.Profiles* attribute), 3
 sampling_time (*tomo.utils.tomo_input.Frames* attribute), 34
 save_difference_ftn() (in module *tomo.utils.tomo_output*), 36
 save_phase_space_ftn() (in module *tomo.utils.tomo_output*), 37
 save_profile_ftn() (in module *tomo.utils.tomo_output*), 37
 save_self_volt_profile_ftn() (in module *tomo.utils.tomo_output*), 37
 self_field_flag (*tomo.tracking.__tracking.ParticleTracker* attribute), 11
 self_field_flag (*tomo.tracking.machine.Machine* attribute), 8
 self_field_flag (*tomo.tracking.tracking.Tracking* attribute), 17
 self_field_flag() (*tomo.tracking.__tracking.ParticleTracker* property), 11
 SelfFieldTrackingError, 33
 show() (in module *tomo.utils.tomo_output*), 37
 show_convergence() (*tomo.tomography.tomography.TomographyCpp* method), 22
 show_difference() (*tomo.tomography.tomography.TomographyCpp* method), 22
 show_image() (*tomo.tomography.tomography.TomographyCpp* method), 22
 show_iter() (*tomo.tomography.tomography.TomographyCpp* method), 22
 show_profile() (*tomo.tomography.tomography.TomographyCpp* method), 22
 show_waterfall() (*tomo.tomography.tomography.TomographyCpp* method), 22
 skip_bins_end (*tomo.utils.tomo_input.Frames* attribute), 34
 skip_bins_start (*tomo.utils.tomo_input.Frames* attribute), 34
 skip_frames (*tomo.utils.tomo_input.Frames* attribute), 34
 snpt (*tomo.tracking.machine.Machine* attribute), 8
 SpaceChargeParameterError, 33
 synch_part_x (*tomo.tracking.machine.Machine* attribute), 8
 synch_part_y (*tomo.tracking.machine.Machine* attribute), 8
 time_at_turn (*tomo.tracking.machine.Machine* attribute), 9
 to_waterfall() (*tomo.utils.tomo_input.Frames* method), 35
 tomo.compile (module), 42
 tomo.cpp_routines.tomolib_wrappers (module), 38
 tomo.data.profiles (module), 3
 tomo.tomography.__tomography (module), 19
 tomo.tomography.tomography (module), 21
 tomo.tracking.__tracking (module), 10
 tomo.tracking.machine (module), 5
 tomo.tracking.particles (module), 12
 tomo.tracking.phase_space_info (module), 15
 tomo.tracking.tracking (module), 16
 tomo.utils.assertions (module), 27
 tomo.utils.data_treatment (module), 26
 tomo.utils.exceptions (module), 32
 tomo.utils.physics (module), 23
 tomo.utils.tomo_input (module), 33
 tomo.utils.tomo_output (module), 36
 tomo.utils.tomo_run (module), 38
 Tomography (class in *tomo.tomography.__tomography*), 19
 TomographyCpp (class in *tomo.tomography.tomography*), 21
 track() (*tomo.tracking.tracking.Tracking* method), 18
 Tracking (class in *tomo.tracking.tracking*), 16
 TrackingError, 33
 trans_gamma (*tomo.tracking.machine.Machine* attribute), 7
 txt_input_to_machine() (in module *tomo.utils.tomo_input*), 35

U

UnequalArrayShapes, 33

V

values_at_turns()
(*tomo.tracking.machine.Machine* method),
10

ValuesOutOfBrackets, 33

vrfl (*tomo.tracking.machine.Machine* attribute), 6

vrfl_at_turn (*tomo.tracking.machine.Machine* at-
tribute), 9

vrfl1dot (*tomo.tracking.machine.Machine* attribute), 6

vrfl2 (*tomo.tracking.machine.Machine* attribute), 6

vrfl2_at_turn (*tomo.tracking.machine.Machine* at-
tribute), 9

vrfl2dot (*tomo.tracking.machine.Machine* attribute), 6

vrft () (in module *tomo.utils.physics*), 25

vself (*tomo.data.profiles.Profiles* attribute), 4

W

waterfall (*tomo.data.profiles.Profiles* attribute), 3

waterfall (*tomo.tomography.__tomography.Tomography*
attribute), 19

waterfall (*tomo.tomography.tomography.TomographyCpp*
attribute), 21

waterfall () (*tomo.data.profiles.Profiles* property), 5

waterfall () (*tomo.tomography.__tomography.Tomography*
property), 20

WaterfallError, 33

WaterfallReducedToZero, 33

wrap_length (*tomo.data.profiles.Profiles* attribute), 4

write_plotinfo_ftn () (in module
tomo.utils.tomo_output), 37

X

xorigin (*tomo.tracking.particles.Particles* attribute),
12

xorigin (*tomo.tracking.phase_space_info.PhaseSpaceInfo*
attribute), 15

xp (*tomo.tomography.__tomography.Tomography* at-
tribute), 19

xp (*tomo.tomography.tomography.TomographyCpp* at-
tribute), 21

xp () (*tomo.tomography.__tomography.Tomography*
property), 20

XPOutOfImageWidthError, 33

Y

yp () (*tomo.tomography.__tomography.Tomography*
property), 20

Z

zwall_over_n (*tomo.tracking.machine.Machine* at-
tribute), 7