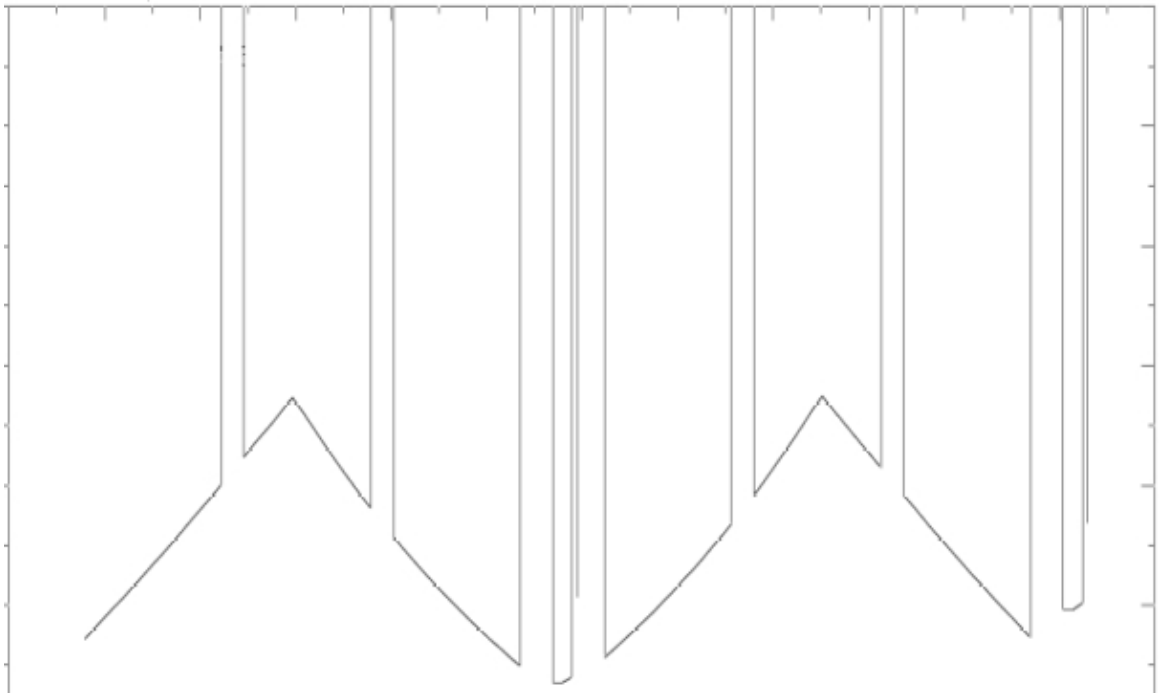
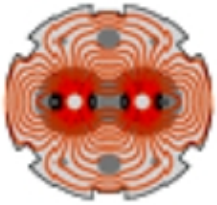


Computation of accelerator aperture and its application to LHC



Ivar-Kristian Waarum
Sør-Trøndelag University College

Bachelor project, 2004

SØR-TRØNDELAG UNIVERSITY COLLEGE
Faculty of Technology
Department of Electrical Engineering
7004 Trondheim

Project Report

Project Title: Computation of accelerator aperture and its application to LHC	**Date:** 25.11.2004
Oppgavetittel: Datastyrt beregning av strålerom i partikkelakselleratorer, med utgangspunkt i LHC.	**Pages/Appendix:** 64/38 + CD
Participants: Ivar-Kristian Waarum ivar_waarum@hotmail.com +47 97739444	**STUC Supervisor** Morten Christian Svensson morten@iet.hist.no +47 73559609
Institute/Branch of study: Program for elektro- og datateknikk / Automatiseringsteknikk	**Project number:** 77
Employer: CERN Accelerators and Beams department Accelerators and Beam Physics group	**CERN Supervisor:** J. Bernard Jeanneret jbj@cern.ch +41 22763190

Completely available ☒

Available after agreement with employer ☐

The report is available as from 01.12.2004

Preface

This report is the written result of the subject "P0500E - Hovedprosjekt i automatisering". Normally, this is a 18 study point subject carried out in the spring semester of the year of graduation as an engineer from Sør-Trøndelag University College. The author enjoyed the priviliege of being offered a contract as a technical student at CERN, Geneva, in the context of the graduation project. Working one year in the Accelerators and Beam Physics group was an extraordinary experience, giving a glimpse into new physics subjects and a working environment applying high levels of innovativity and thoroughness while handling advanced engineering issues of particle experiment machinery. This is the type of experience that motivates a student for new educational goals.

A major challenge was the introduction to a subject completely new to me, that of accelerator physics. Programming I knew to an extent, but for the outcome to be as good as possible it is important to understand the why's and the total picture. A programmer must know the needs of the end-users. Feeling I succeeded in this, the report was made accordingly, with a focus on background information as well as programming technicalities.

The report describes a finished product. It is fully capable of aperture analysis of the LHC and other machines. However, since I was offered to extend my contract, we will continue working on the project in 2005 to add the possibility to treat special-purpose magnets in the experiment regions. The updated report will be released this spring as a CERN document on accelerator aperture computation.

Well-deserved thanks goes to my supervisor Dr. Jean-Bernard Jeanneret, for providing his enormous knowledge of professional skills and method of working "slowly and step-by-step". To Dr's. Werner Herr and Eric D'Amico for help with MAD and C programming in general. To my STUC supervisor Dr. M. C. Svensson for his encouragements directing me towards CERN in the first place. And last but not least to office mate and soon-to-be Dr. Jaroslaw Pasternak for answering twenty thousand questions about accelerators and particle stuff.

Meyrin, Geneva 17.11.2004

Ivar-Kristian Waarum

Summary

Design and construction of a particle accelerator machine of todays measures is a tremendous engineering task, for which calculation and analysis would be an overwhelming job without specialised tools. This report describes a module in a computer program used for simulating and designing such particle accelerator machines. The task of the module is to calculate the aperture around the beam at every point in the machine. Initially, a short introduction to accelerator theory is given, and important terms explained. The computer program itself is presented along with an example of usage in Chapters 3 and 4. In order to understand more of the background for the project, chapters 5 and 6 present the particle beam and the pipe containing it, as well as an introduction to collimation and the importance of protecting the machinery against heat transfer from the beam. One issue of the project was how to model the beam and the pipe in the computer program. The solutions for this are also presented, both how they were programmed and the geometrical considerations behind. Chapter 7 elaborates on movement and oscillations of the beam and error tolerances of the pipe, and gives an understanding of the accuracy and precision needed in the construction of a machine of this kind. Solutions of the main problem, how to calculate the aperture, is given in Chapter 8. The interesting part for the end-user is Chapter 9, which shows the module output and gives ideas on how the module can be used for analysis. Throughout the report are references to functions in the source code, all of which can be found in Appendix D.3. Documentation on the source code can be found in Appendix A as explanations of functions and parameters, pseudo code and a data flowchart. A user's guide for MAD-X users is included as Appendix B.

For background information on the project itself, see the report from the preproject, included on the project CD. On this CD one also finds the total program source code and some appendixes which was impractical to include in paper format.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why accelerators?

Particle physics as a field of research started properly at the end of the 19th century, with the discovery of the electron (Thomson, 1898). Further elementary particles, the proton and the neutron, were discovered in respectively 1919 (Rutherford) and 1931 (Chadwick). The motivation of peering even further down into matter by splitting the atom nucleus led to the construction of the first particle accelerators and colliders in the 1920's. Boosting the energy of particles with electric charge, these machines have turned out to play an important role in sub-atomic particle studies. Einstein's equation $E = m \cdot c^2$ states the relationship between energy E and mass m. As more energy is applied to a particle, more mass may be created in a crash. Over time these machines have grown in size, energy level and complexity, and have opened doors into new areas of physics. The most recent accomplishments includes the discoveries of quarks, and as higher collision energies will be reached in the future, we will learn even more about the fundamental building blocks of nature. Now a common tool, particle accelerators are used for education and research at universities and laboratories around the world.

1.2 What is CERN?

The most complex network structure of accelerators in the world is situated at the European Organization for Nuclear Research (CERN), Geneva [1], see Figure 1.1. Meant to be Europe's answer to the huge progress in American and Japanese science and technology in the first half of the 20th century, CERN was founded in Geneva in 1954. This would initiate a prosperous era for European scientific and elementary particle research activity. Since then, extensive experiment research, scientific discoveries and three Nobel Prize winners have made CERN an important, if not the most important, nuclear research organization on world scale. Today, CERN employs 3000 workers on different levels, in addition to almost 6500 visiting scientists and students from all over the world.

Figure 1.2 is an overview of the main accelerator structure. The different machines provide particles to a number of different experiments, and allow treatment of and research on protons, antiprotons, electrons, positrons, muons etc.

Figure 1.1: Geographical map over CERN, the LHC and its experiments. Placed on the border between France and Switzerland, the laboratory is an international effort in all matters.

The experiments have different demands to energy level of particles, the quality and intensity of the beam, and other parameters. It is clear that the construction of a particle accelerator is a great engineering challenge. Currently, the prioritized project at CERN is the Large Hadron Collider. In this 27 km long circular accelerator, two separate beams will be brought to extreme energy levels and then collided at the centres of four huge particle detectors. The amount of energy packed in such a tiny space creates an environment not unlike the one presumed to have been present during the creation of the universe, according to the big bang theory. A total of 1200 large magnets and three times that of smaller ones needs to be designed, produced and tested for this project, which will be the world's largest machine when finished. Scheduled to be operational in 2007, the LHC will hopefully break new ground for the study of elementary particles.

For more information about the LHC, see the official web page. [2]

1.3 The role of the author

One tool for construction and analysis of accelerator machines is MAD, a software utility which simulates the beam throughput and gives information about machine hardware requirements and beam behaviour.

This report describes a new module in the MAD program, which allows anal-

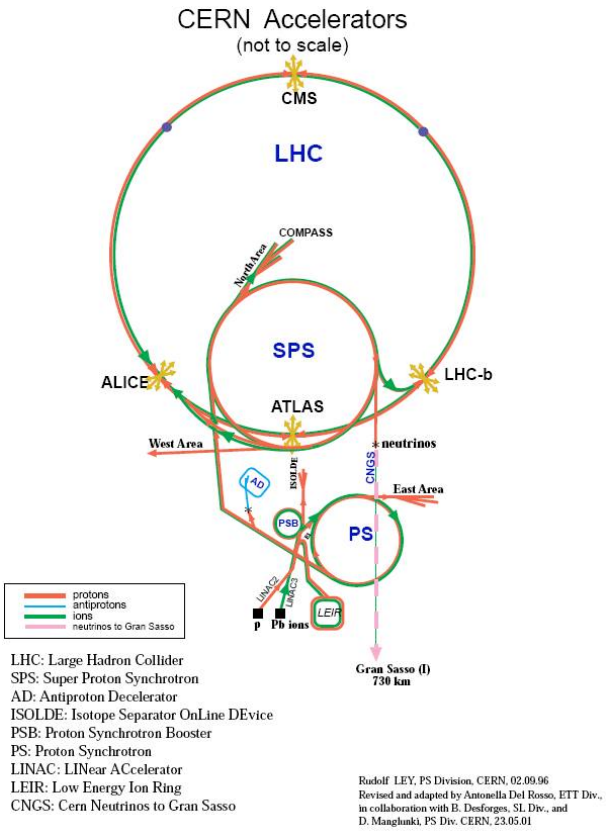


Figure 1.2: Some accelerators and beam lines at CERN. Particles are accelerated in several stages, from rest energy in the ion-sources to top energy in the experiments. A network like this has extreme demands to design spesifications in order to function both as a unity and at level of the individual beam lines.

ysis of the aperture in a machine. The development and coding of this module was the author's responsibility, working under CERN supervisor J. Bernard Jeanneret. The major problem issues were how to model the beam pipe and the beam in the program, how to take into account error tolerances, and finally how to compute the aperture. All this is treated in Chapters 5, 6, 7 and 8. The report also gives some theoretical background that was necessary to learn in order to understand the purpose of the module and to familiarize with concepts and terminology specific for accelerator physics. The beam and the hardware of the machine are presented, and regarding the method of making a model of them in MAD, an example is worked out in Chapter 4. All source code can be found in Appendixes D.3 and D.4.

1.4 Basics on a golden plate

Heavy subjects and wording might scare away the most curious reader. Here is the subject in 200 words:

Particles (Ions, electrons, etc.) are drawn out of a plasma cloud by magnets. They drift in a long vacuum chamber, are accelerated forward by electrical fields and held together as a beam by magnetic fields. After enough acceleration, the particles have a momentum that makes them interesting to perform experiments on, either by crashing two accelerated particles, or by crashing a particle into a stationary target. To keep the particles together as a beam during acceleration is difficult. The magnets used for this must be fine-tuned. Before an accelerator is built, simulations must be done to find out which kind of magnets to use, their strengths and so on. One program that does these kinds of simulations is MAD-X. One of the things we want MAD-X to be able to do, is to calculate the amount of space between the beam and the walls of the vacuum chamber. The basics we need to know in order to do this, are the shape of the beam, the shape of the vacuum chamber, and where the beam is positioned inside this chamber.

The report is written for readers with engineering background. Readers with no interest in programming will learn about the behaviour of the particles during acceleration, the accelerator machine and how to safely run a beam through it. Also about the MAD-X program, the basics of beam and vacuum chamber shape and beam positioning. So just as interesting as reading it as a report on the given project, may be to read it as a basic introduction to particle accelerators in general.

Chapter 2

Concepts of accelerator physics

A short introduction to the fundamentals of accelerators, and explanation of some technical terms. The way of guiding the particles through the machine is presented. More detailed explanations can be found in [3] and [4].

2.1 Bending a particle beam

Particles are accelerated by potential differences. A particle in a circular accelerator has it's energy increased at each turn, or each pass through the accelerating cavity. Magnetic fields are used to guide the particles around the ring during acceleration. The force from a magnetic field is

$$F = e \cdot \mathbf{v} \times \mathbf{B},$$

where e and $\mathbf{v}$ is the charge and speed of a particle in a field of strength $\mathbf{B}$.

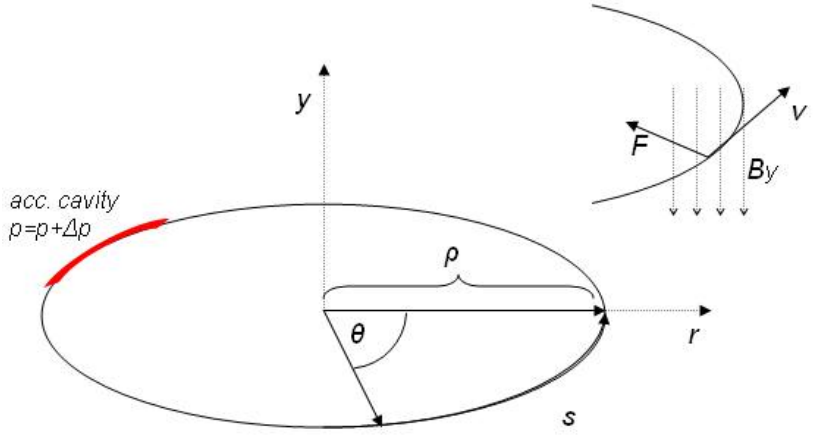


Figure 2.1: A particle moving in longitudinal direction s, kept in a circular trajectory by vertical magnetic fields B_y.

The force needed to keep a particle in a circular path is equal to the centrifugal force of the particle. If we consider only the horizontal plane r, the equation becomes

$$e \cdot v \cdot B_y = \frac{mv^2}{\rho} \Rightarrow B_y = \frac{p}{e \cdot \rho}, \tag{2.1}$$

where B_y is a vertical field, p the particle momentum and ρ the radius of motion. For a real accelerator, B_y is applied by dipolar magnets which are "ramped" (i.e. B_y is increased proportionally to p) as the particles are accelerated. A particle in a perfect machine will stay in the same orbit, the reference orbit, for a large amount of turns.

2.2 Focusing a particle beam

We introduce the distance $x = r - \rho$, where $x = 0$ as long as r does not change from the initial value ρ, see Figure 2.1. Equation 2.1 says that if we now consider a particle with initial conditions $x \neq 0$ and/or $x' \neq 0$, it will stray from the reference trajectory. This is usually the case, after the particles are extracted out of the gas plasma serving as particle source. If $|x|$ becomes too large, the trajectory will not fit in the machine, so in addition to bending magnets, a way of focusing the beam along the trajectory ρ is necessary. Particles with a positive x must be pushed inwards and particles with a negative x must be pushed outwards towards this reference trajectory. Magnets with four poles are used for this.

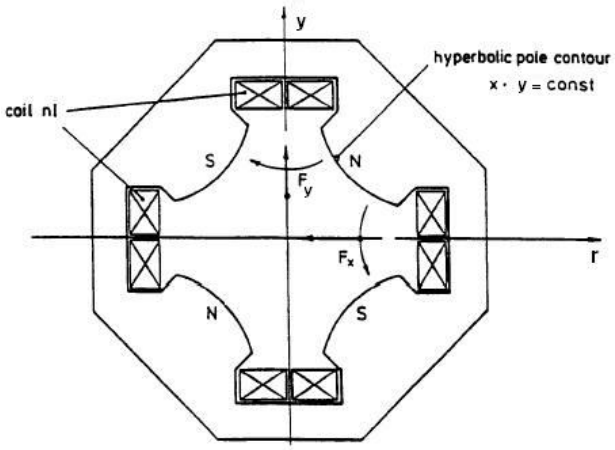


Figure 2.2: Cross-section of a quadrupole magnet. A particle following the reference trajectory passes through the centre. [3].

The horizontal forces of a quadrupole magnet are proportional to $|x|$. Using the right-hand rule on the vertical magnetic fields shows that particles with $|x| \neq 0$ are pushed toward the centre of the magnet. Using the right-hand rule on the horizontal fields shows that particles with a deviation in the vertical plane are pushed away from the centre. The magnet is simultanously focusing in one plane and defocusing in the other. The method of usage is to turn every other

quadrupole 90°, so that if we again consider only the horizontal plane, the effect is focusing - defocusing - focusing etc. The overall effect is that the particles are kept within small distances from the reference trajectory. A beam of particles will have its maximum width in a focusing quadrupole, and maximum heigth in a defocusing one, see Fig 2.3. The movement is called *betatronic oscillations*, and is described by the β-function introduced in Chapter 2.5. The focusing - defocusing - focusing structure is often abbreviated to FODO, and when we talk about a *FODO-lattice* the F and D are quadrupoles and the O's are bending magnets or empty driftspaces.

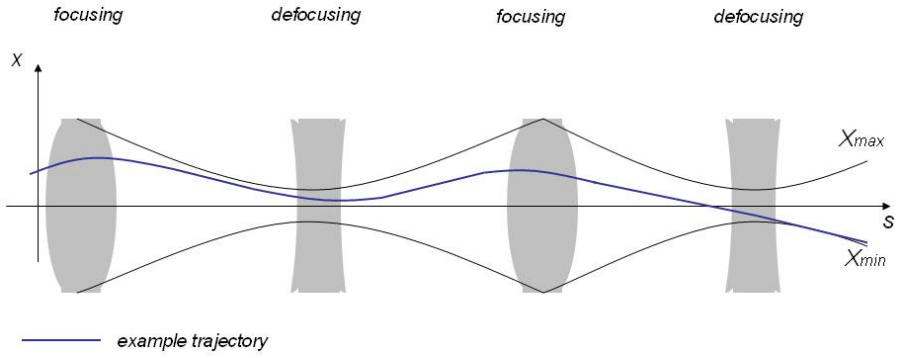


Figure 2.3: The transverse beam size is limited by the quadrupoles. The quantities $X_{max}(s)$ and $X_{min}(s)$ gives the beam envelope in the horizontal plane, $\Delta_x(s)$.

2.3 Phase-space and sigma distribution

Particles have different initial parameters x, x', y, y', coming from the plasma cloud. These parameters are referred to as *phase-space* parameters, since they describe the rate of oscillation change. The initial distribution of these parameters is for practical purposes assumed to be gaussian, hence most particles have small oscillation amplitudes around the reference orbit, fewer have large amplitudes. When talking about a particle beam size, it is always expressed by a certain amount of σ of particles, it being the r. m. s. beam size. Fig 2.3 shows the beam envelope for one sigma value. This gives the boundaries for particles below a specific oscillation amplitude.

2.4 Dispersion

Another source of deviation from the reference orbit is momentum differences, or *dispersion*. From Eq. 2.1 we see that a particle with $p \neq p_0$ will have a circulation orbit with $\rho \neq \rho_0$. The quadrupoles affects also these orbits, so the reference orbits of particles with only momentum deviation have the same periodicity as the beam envelope in Fig 2.3. (In addition, two particles may have the same phase-space coordinates x, x', y, y', but if $p_1 \neq p_2$ they will be affected differently in a magnetic field. Especially a problem in the quadrupoles,

which will focus wrongly. This effect is compensated by magnets with six poles, but this will not be addressed here.) The momentum differences are largely due to the acceleration method. Figure 2.4 shows how a bunch of particles hits the

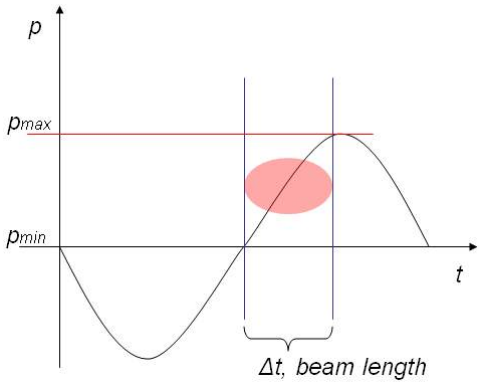


Figure 2.4: Momentum differences due to the acceleration method.

acceleration cavity. The particles in the front end of the bunch receives a smaller Δ_p than particles in the end of the bunch. This means that the foremost particle will move towards the back of the bunch, and the backmost particles towards the front. The gap $\Delta_p = \frac{p_{max}-p_{min}}{2}$ is considered to be constant, and is the worst-case momentum deviation of a particle. Now the dispersion can be defined as $D(s) = \frac{x(s)}{\Delta_p/p_0}$, where $D(s)$ is the relation between particle momentum and position. It can be calculated from the distances between magnets, and their strengths.

2.5 Twiss parameters

We see that to keep track of a particle in an accelerator, several variables must be taken into account. For our purpose, we need a method to easily calculate a particles position at all times while in the machine. Such a tool is the *Twiss matrix*, which is a transfer matrix for the parameters $\beta(s)$, $\alpha(s)$ and $\gamma(s)$. These are parameters describing particle trajectories, and can be derived from the equations of motion for a particle. They can also be shown graphically, if we remember that a particle has phase-space coordinates x, x' in the horizontal plane. In the middle of a focusing quadrupole, $x = x_{max}$ and $x' \approx 0$. Close to this point, the boundaries for the trajectories can be represented as an ellipse with area $\pi \cdot \epsilon$:

It can be shown that the area of the ellipse in Fig 2.5 is constant for all s. The ϵ giving the size of the area is called the *emittance*, an important parameter for beam quality. The $\beta(s)$ and $\gamma(s)$ are determined by the magnet lattice. A mathematical description of the lattice can be done by determining a transfer matrix for all elements. After matrix multiplication and transforming the final matrix to Twiss format, the Twiss parameters can then be calculated for any s. The β describes the change of beam shape, examples are given later. An uncertainty parameter, the *beta beating* factor k_β, is often used in equations

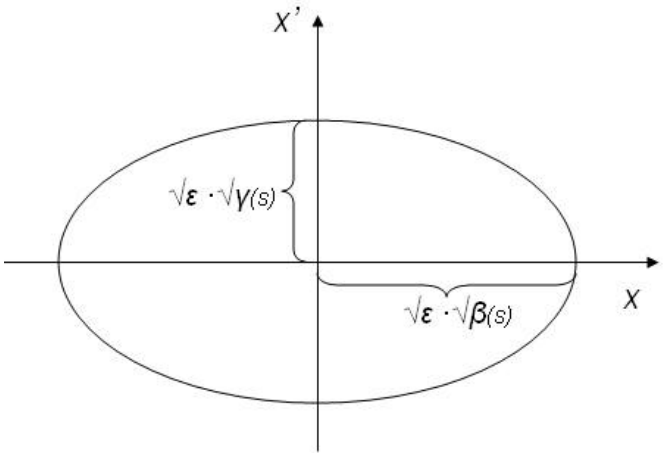


Figure 2.5: Phase-space plot for the horizontal plane close to a focusing quadrupole. The beam envelope in Fig 2.3 is shown here as $\sqrt{\epsilon} \cdot \sqrt{\beta(s)}$.

with β to give a certain tolerance for deviations from the theoretical values.

$$k_\beta = \frac{\beta^{act}}{\beta^{th}}, \tag{2.2}$$

where *th* stands for the theoretical value, and *act* the actual value of $\beta(s)$.

The size of the beam envelope can be found from the Twiss parameters.

$$\Delta(s) = \sqrt{\beta(s) \cdot \epsilon} \cdot k_\beta \tag{2.3}$$

2.6 Apertures

Methods for keeping the size of the beam small, and its path regular, has already been mentioned. The beam must be kept at transverse size that fits in the beam pipe, and this limitation is called the *physical aperture*. A similar term not to be interchanged, is the *dynamic aperture*. This is the maximal betatronic amplitude of particles following regular paths. Particles with larger amplitudes have a more chaotic pattern of motion and impairs the overall beam quality. As a result of this, it is necessary to "clean" the beam for the unwanted particles at regular intervals. For this purpose, most accelerators incorporates physical obstacles called *collimators* in some parts of the beam pipe. More on this in Chapter 6.

Chapter 3

MAD

A presentation of the MAD program, first from a user point of view. Then the structure of the source code is explained, with examples from the aperture module.

3.1 Overview and functionality

Modular Accelerator Design: MAD is a software tool for simulating charged particles' behaviour in accelerators and beam lines. Developed and maintained by the ABP group at CERN, it is one of the worlds most widely used softwares of its kind, put to use at a number of accelerator laboratories. Constantly under development, the current version is the tenth one, hence called MAD-X. The structure of the program is modular in the sense that different parts of the source code are strictly separated and independently manageable, not only as different functions, but as more or less independent software utilities. Each of these modules is a utility for one or more aspects of accelerator design, like Twiss parameter calculations, single particle tracking, closed orbit analysis, or aperture calculations. The flexible environment that allows the different modules to be written in either C or different editions of FORTRAN (77/90/95), ensures mutual compatibility and simplifies software updates. The skeleton basis for the modules is a main module in C called "CONTROL". All major i/o is handled by the main module, which also makes it easier to delegate responsibility for the functionality of different modules among members of the ABP group.

A short description of the modules:

twiss: Computes linear lattice functions (see chapter 2.5), element by element.

makethin: Transforms the given lattice to a thin-lense approximation. All magnets are split into several elements of zero length, but overall performance is matched as close to original as possible.

track: Allows for tracking of trajectory for a single particle with given initial conditions. A number of particles can be tracked, element by element, for thousands of revolutions around a ring, searching for unwanted behaviour or particle loss.

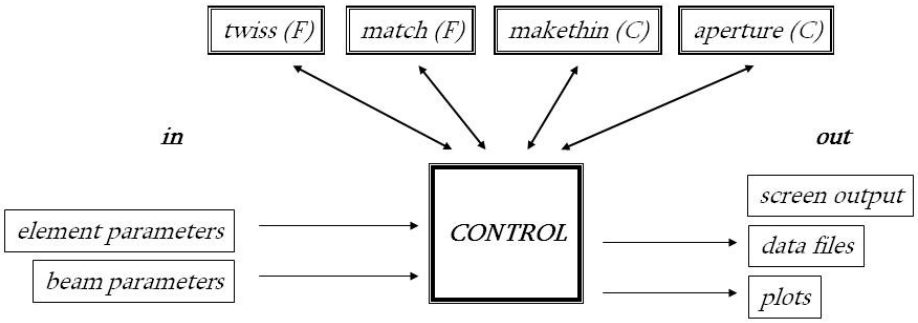


Figure 3.1: Overview of MAD-X data flow with some of the submodules.

match: Gives the possibility to set some parameters of beam performance (beta, dispersion etc.) to target values, and let MAD-X treat some element parameters as "free". A numerical search is performed to find the free parameters that gives the wanted target values.

error assignment: Assigns magnetic field errors, or element alignment errors. The errors can be constant, random, gauss-random, etc, and be assigned to specific elements or entire sections of a lattice.

emit: Adjusts the RF frequencies to obtain a specified average energy error of a particle.

orbit correction: Corrects the closed orbit or another particle trajectory by adjusting corrector magnets.

survey: Computes the coordinates of machine elements in a global reference system, usable for physical installation.

plot: Support module that interpolates and plots values of data tables to postscript files.

IBS: Computes emittance growth rates due to Coulomb scattering of particles inside a beam.

aperture: Tracks a beam halo along the lattice, using physical information of the beam pipe to compute available aperture.

3.2 Input

MAD calculations are based on hardware specifications of the accelerator machine and initial properties of the beam particles. Internally, the hardware is treated as transfer matrices, giving a mathematical description of the beam line. The basis must be given by the user, in the form of a file containing information about the beam, magnets and other hardware. This file is fed to the program on the command line: `madx < filename`. The syntax in the file resembles C.

3.2.1 Initial beam properties

```
title, ‘‘MAD-X example’’;
beam, particle = proton, sequence = ex1, energy = 450.0,
                 exn = 3.75e-6, eyn = 3.75e-6;

call file = ‘‘elements.seq’’;

use, sequence = ex1;
select, flag = twiss, column = name, s, x, betx, dx;
twiss, save, file = twiss1.out;
plot, haxis = s, vaxis = x, betx, dx;
stop;
```

Above is an example of a MAD-X input file. Firstly, the initial beam properties: The type of particles (MAD contains a table of mass and charge properties for many particle types), their energy level in GeV, and transverse emittances in meters. The script reads hardware element parameters from the file elements.seq. Twiss parameters are calculated, a table of parameters s, x, betx and dx is saved to file as twiss1.out, and the data is plotted.

3.2.2 Initial hardware properties

The hardware specifications are mostly concerning the magnetic elements. Their type (the number of poles and their placement gives the shape of the field), strength, length, longitudinal position and more. An accelerator usually contains many elements and has a more or less periodic structure, so MAD features the possibility to read element descriptions in a loop, see the example in chapter 4. For a machine of some size it is also convenient to keep the element descriptions in a separate file (The element input file for the LHC is almost 15000 lines long!). MAD-X provides this by letting one line of input call an entire other file to be read, and it is common usage to separate the element definition to a file with extension `.seq`. MAD-X can also treat elements like accelerating cavities, beam position monitors and collimators. It even allows to insert an arbitrary element in the form of a matrix. Under is an example from a MAD-X input file.

```
//define beam position monitors
bpm: monitor, l = 0.1, apertype = rectellipse,
              aperture = {0.07, 0.05, 0.07, 0.07};
//define bending magnets
mb: rbend, l = 2, k0=0.37, apertype = rectellipse,
           aperture = {0.09, 0.05, 0.09, 0.07};

//define focusing and defocusing quadrupoles
qf: quadrupole, l = 0.5, k1 = 0.024, apertype = rectellipse,
                aperture = {0.07, 0.05, 0.07, 0.07};
qd: quadrupole, l = 0.5, k1 = -0.024, apertype = rectellipse,
                aperture = {0.05, 0.07, 0.07, 0.07};

start_machine: marker, at = 0.0;
```

```
 qf: qf, at = 0.0;
bpm:bpm, at = 0.5;
 mb: mb, at = 1.5;
 mb: mb, at = 4.0;
 qd: qd, at = 7.0;
.................
.................
end_machine: marker at=circum;
```

A monitor with length of 0.1 meter is defined as "bpm". Bending magnets of type rbend with length of 2 meters and strength of 0.37 is defined as "mb". All elements are given an apertype definition and aperture parameters according to physical measurements. The last lines place the previously defined elements at specific longitudinal values. When all elements are defined, they are treated as a whole and refered to as a sequence, through wich we can run a beam or compute Twiss functions.

It is clear that MAD-X must be fed with a large amount of information to be useful. Chapter 4 combines some theory from chapter 2 and this chapter, in an attempt to show how MAD-X provides the transition between physics theory and practice. For further examples and instructions of usage, see the MAD-X User's Guide [5].

3.3 Inside the CONTROL source code

We now go inside the program source code. Input and output is handled by the main data management structure; CONTROL. In order to add a new module to MAD-X, additions must be made in CONTROL to support it properly. As MAD-X runs, the input script file is read line by line, while each word is compared with an internal dictionary and list of commands. Some exerpts from the aperture command definition:

```
char command_def[] =
...
...
" "
"aperture: aperture none 0 0 "
"file     = [s, none, aper1.out], "
"range    = [s, #s/#e, none], "
"exn      = [r, 3.75e-6], "
"nco      = [i, 5], "
"halo     = [r, {6., 8.4, 7.3, 7.3}], "
...
" "
```

Here "aperture:" signifies the start of the command, and "file", "range" etc. are the possible arguments. The rest is argument types, either **s**tring, **l**ogic, **r**eal or **i**nteger, and the default argument values.

3.3.1 Structure variables

As information about the beam and the elements is read, a cobweb of structure variables is created to store it. Magnets, beams, tables, sequences, all have their own structure type. The lattice elements are tied together as a linked list of node structures, each structure containing the description of the element. Submodules uses these lists to access the information from the input file. Information for the entire lattice can be searched for and extracted by starting at a defined node→current, and reading the node names as the node→current is set equal to node→current→next. An excellent example of the structure system is how to retrieve aperture information for each element:

In Appendix D.3, the function `build_pipe` needs the aperture parameters of the current node. It calls the function `get_aperture(struct node* node, char* par)`, which returns aperture parameter varX of a node:

```
*ap1 = get_aperture(current_node, "var1");
```

The node has a `struct element* p_elem`, which points to a struct with information about the element. The element struct has a `struct command* def` which points to a struct with the information from the command defining the element in the first place. The command struct contains a
`struct command_parameter_list* par`, which points to a list of pointers to its parameters. (This list contains "apertype", "length", etc.). The par struct again contains a `struct command_parameter** parameters`, which points to the actual parameter value. To summarize, *ap1 can be written as:
`*ap1 = current_node→p_elem→def→par→parameters[1]→double_value;`

Definitions of the structures mentioned above, and the functions needed to read aperture parameters are included in Appendix D.1. These were already included in MAD-X before project start.

3.3.2 Tables and output

Output is often saved in the form of internal tables. MAD-X creates these tables at need, based on definitions given in the source code. These tables are "public", available for all modules. A good example is the table of computed apertures:

```
/* table descriptors:
type 1 = int,
type 2 = double,
type 3 = string    */

int ap_table_types[] =
{
3, 2, 3,
2, 2, 2,
2, 2, 2, 2,
2, 2, 2,
2, 2, 2, 2, 2, 2, 2,
};

char* ap_table_cols[] =
{
"name", "n1", "apertype",
```

```
"rtol", "xtol", "ytol",
"ap1", "ap2", "ap3", "ap4",
"on_ap", "on_elem", "spec",
"s", "betx", "bety", "dx", "dy", "x", "y",
" "  /* blank terminates */
};
```

A table for any need can be implemented, to store any useful variables. The aperture table provides other functions with any aperture information they might need. For the end-user, the information stored in tables can be sent to the screen or written to a file. Table 3.1 shows the first three parameters as the user receives it, while Table 3.2 shows an example of output directly to the screen. For a full version of a table printed to file, see Appendix D.2.

Table 3.1: An example of output to file; apertures computed inside of an MB magnet. The number of parameters to be written can be specified by the user in the input script.

```
@ NAME           %08s "APERTURE"
@ TYPE           %08s "APERTURE"
@ TITLE          %07s "LHC inj"
@ ORIGIN         %16s "MAD-X 2.10 Linux"
@ DATE           %08s "18/06/04"
@ TIME           %08s "14.39.28"

* NAME                      N1 APERTYPE
$ %s                      %le %s
 "MBRC.4L2.B1"       21.80695002 "RECTELLIPSE"
 "MBRC.4L2.B1"       22.00372148 "RECTELLIPSE"
 "MBRC.4L2.B1"       23.30133439 "RECTELLIPSE"
 "MBRC.4L2.B1"       23.43916607 "RECTELLIPSE"
 ....etc......       ....etc.... ....etc......
```

Table 3.2: An example of output to screen while the program is running; a summary of Twiss functions.

```
++++++ table:  summ

      length       orbit5         alfa      gammatr
  26658.8832           -0            0            0
         q1          dq1      betxmax        dxmax
 64.27999981            0  595.1194205  2.860329515
       dxrms       xcomax       xcorms           q2
   1.4032533            0            0  59.31000026
        dq2      betymax        dymax        dyrms
          0  609.5258812            0            0
      ycomax       ycorms       deltap
           0            0            0
```

Chapter 4

MAD-X practical example

A demonstration on how MAD-X can be used. First some calculations of necessary magnet parameters, then an example of input script files and results after running MAD-X.

4.1 Theory

One use of MAD is the construction of a completely new accelerator. As an example of this, and to demonstrate some theory in practice, here is presented the first design steps. The ring is loosely based on the Proton Synchrotron (PS) ring at CERN. As a link in the injector chain to the LHC, the ring's task is to increase the proton momentum from 1.4 to 25 GeV/c.

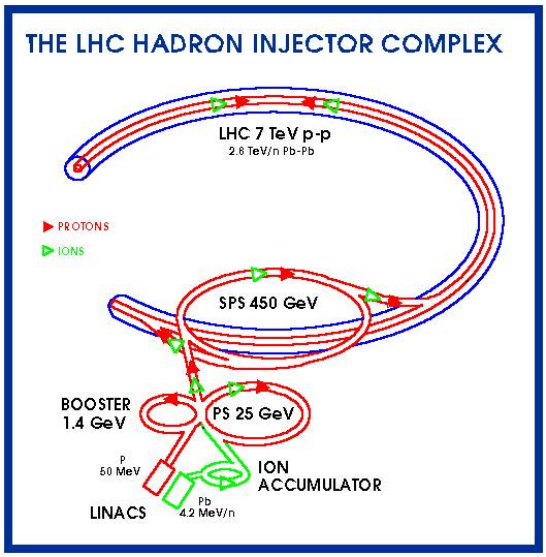


Figure 4.1: The LHC injector chain. All the accelerators have a limited momentum working range, so acceleration is done in several steps. CERN boasts a number of accelerator machines.

We use these parameters:
Circumference: 800 m.
Dipole length: 9 m.
Quadrupole length: 1 m.

The PS ring is one of the finest working antiquities in the world, constructed in 1959 with water cooled magnets. Maximum magnetic field: 1.5 T. The actual bending strength is expressed with an angle α_x.

We start with the bending magnets. To form a complete ring, $\sum_{x=0}^{n} \alpha_x = 2\pi$ must be true for a number of n bending magnets as shown in Figure 4.2.

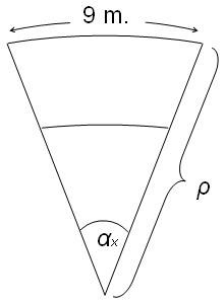


Figure 4.2: Lengths and angle of a bending magnet.

A variant of Equation 2.1 gives the radius of the mean trajectory:

$$\rho = \frac{E}{B} = \frac{p[\frac{GeV}{c}]}{c \cdot 10^{-9} \cdot B[T]} = \frac{25[\frac{GeV}{c}]}{c \cdot 10^{-9} \cdot 1.5[T]} = 55.593[m]$$

The angle for each dipole is:

$$\alpha = \frac{9[m]}{\rho} = \frac{9[m]}{55.593[m]} = 0.162[rad]$$

which means that at least

$$\frac{2\pi}{\alpha} = \frac{2\pi}{0.162[rad]} = 38.811 \approx 39$$

bending magnets are needed. To make the lattice simpler, we settle for 40 magnets. This allows to use two magnets per cell, and have a cell length of 40 meters. The total number of cells is 800/40=20. A regular FODO structure is chosen for the lattice, with equal spacing between the elements.

Another requirement for the PS ring is to keep the emittance beneath a certain maximum, to ensure proper beam quality for the experiments: $\epsilon_{max} = 3 \cdot 10^{-6}[m]$. With a circular beam pipe with radius $8 \cdot 10^{-2}[m]$ and a beta beating factor of $k_\beta = \sqrt{1.15}$, we decide to allow a maximum beam envelope of $X_{max} = 2 \cdot 10^{-2}[m]$ in the focusing quadrupoles. Equation 2.3 gives the corresponding maximum beta value:

$$\hat{\beta}_{QF} = \frac{(X_{max})^2}{k_\beta} \cdot \frac{1}{\epsilon} = \frac{(2 \cdot 10^{-2}[m])^2}{\sqrt{1.15}} \cdot \frac{1}{3 \cdot 10^{-6}[m]} = 100.82[m]$$

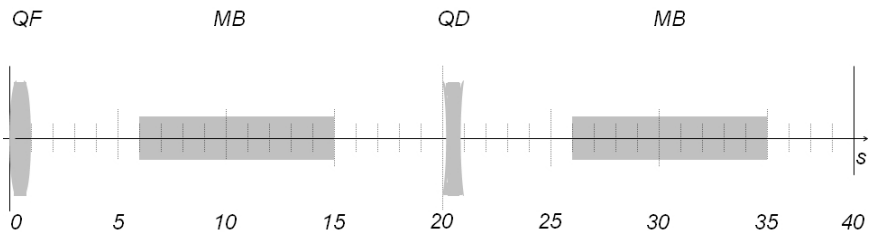


Figure 4.3: The lattice structured as FODO cells.

The expression for phase advance per half cell is:

$$\sin(\phi/2) = \frac{L_{1/2}}{f_{1/2}}$$

where $L_{1/2}$ is the length and $f_{1/2}$ the focal length of a half cell. The expression for the maximum beta function is:

$$\hat{\beta} = f_{1/2} \cdot \frac{1+\sin(\phi/2)}{\cos(\phi/2)} = L_{1/2} \cdot \frac{1+\sin(\phi/2)}{\sin(\phi/2)\cdot\cos(\phi/2)} \tag{4.1}$$

Solving this w.r.t. ϕ gives $\phi = 30.30$. The necessary focal length is now given by:

$$f = \frac{f_{1/2}}{2} = \frac{L_{1/2}}{\sin(\phi/2)\cdot 2} = \frac{40/2}{\sin(30.30/2)\cdot 2} = 38.264 \tag{4.2}$$

The needed quadrupole strength is:

$$k1 = \frac{1}{L_{quad}\cdot f} = \frac{1}{1\cdot 38.264} = 0.026 \tag{4.3}$$

Equations 4.1, 4.2 and 4.3 are taken from [6].

4.2 Practice

All needed parameters are then written in a script file as presented in Chapter 3.2, along with the lattice and beam parameters:

```
circum = 800;
ncell = 20;
lcell = circum/ncell;

//define bending magnets
lmb = 9;
mb: multipole, lrad = dummy, l =  lmb, kn1 = 2*pi/(2*ncell);

//define quadrupoles
lq = 1;
```

```
kqf = 0.026;
kqd = -0.026;
qf: quadrupole, l = lq, k1 = kqf;
qd: quadrupole, l = lq, k1 = kqd;

//sequence
PSex: sequence, refer = centre, l = circum;
start_machine: marker, at = 0;

n = 0;

while (n < ncell)
{
qf: qf, at = n*lcell + lq/2 + 0;
mb: mb, at = n*lcell + lq/2 + 10;
qd: qd, at = n*lcell + lq/2 + 20;
mb: mb, at = n*lcell + lq/2 + 30;

n = n+1;
}
end_machine: marker, at = circum;
endsequence;

beam, particle = proton, sequence = PSex, energy = 25;

use, sequence = PSex;

select, flag = twiss, column = name, s, x, y, betx, bety, dx, dy;
twiss, centre, file = PSextwiss1.out;
setplot, post = 2;
plot, haxis = s, vaxis1 = betx, bety, file = PSex,
      colour = 100, range = qf[1]/qf[2];

stop;
```

The output produced is a plot of beta functions in the first cell (Figure 4.4, and the file psextwiss1.out (Appendix D.2) containing chosen optic parameters. For details, see the MAD-X User Guide [5].

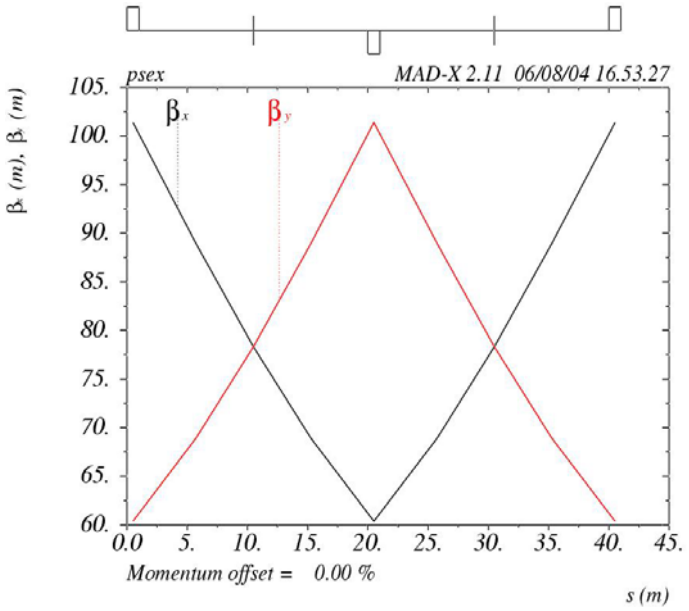


Figure 4.4: The betatronic functions. The magnets are symbolically plotted along the top. Our results are close to the demand of $\hat{\beta}_{QF} \approx 100$. Our lattice can be tweaked to further reduce the beta values, and there is room to add other elements and equipment.

Chapter 5

Beam pipe

First a presentation of the physical device containing the beam, then we look into how MAD-X treats different kinds of pipes.

5.1 Hardware

To keep perfect control of the particles making a beam, it is necessary to avoid them to form too large oscillations. A vacuum as low as possible is needed to minimize particle interaction along the ring, and especially to avoid background noise in the experimental areas. The device which contains the vacuum is the physical limitation for the size of the beam.

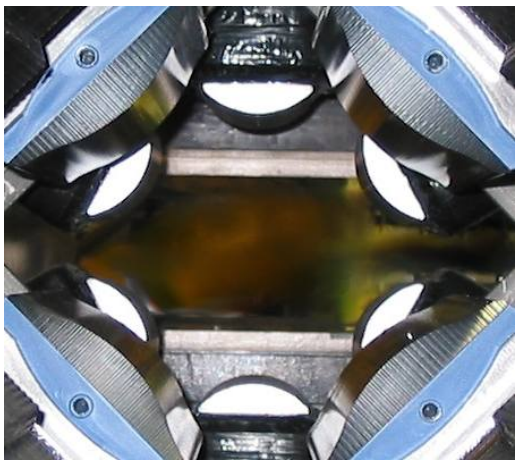


Figure 5.1: From the beam's point of view: A quadrupole (black and blue), a sextupole (black and white) and a dipole (flat and grey). The beam pipe is not yet mounted, but will run straight through the middle. [7]

This device, the *beam pipe*, surrounds the beam through the magnets, as those in Figure 5.1. Another important term is *beam screen*, with which is meant the area of a vertical slice in the beam pipe. (Figure 5.2). Surprisingly small for

many, the beam screen in the LHC is never larger than 45mm in any direction in the superconducting parts of the ring, and 80mm in the warm, straight parts without magnetic fields. The beam screen size is a compromise between which size the halo inside can be squeezed to in order to fit inside it, the difficulties of creating a vacuum in a bigger volume (In the LHC, the beam pipe is 27km long and although it is divided in several separate vacuum chambers, a slight increase in beamscreen diameter amounts to a huge addition in the total volume of magnets), and the economical aspects of building, cooling and operating a bigger machine and magnets. Usually, one wants as much space for the beam as possible to avoid too many particles hitting the walls, and may therefore change the beam screen shape and size. For instance, inside a defocusing quadrupole the beam is large in the vertical direction, and the beam screen may be rotated accordingly.

Figure 5.2: A sliced beam pipe from the LHC, with a rectellipse-like beam screen . Along the straight parts on top and bottom there are cooling pipes with liquid helium too ensure that the magnets nearby are not heated with the energy absorbed by the pipe. [8]

5.2 Physical aperture shapes

A range of different apertypes is necessary to describe the space available to the beam as precisely as possible in all situations. MAD-X is currently operating with six basic types of beam screens, see Table 5.1. This adds to MAD-X' versatility. It can be put to proper use for designing and simulating a variety of accelerators. Which apertype to be used in calculations is read for each element definition as the halo travels along the accelerator. The function `build_pipe()` reads and adjusts the parameters accordingly for numeric treatment.

In order to make a polygonal approximation of the beam screen, it is necessary to first know its exact shape. Each element in an accelerator must be assigned one of the aperture types in Table 5.1. This is done in the sequence file read by MAD-X while processing.

Table 5.1: Apertypes supported by MAD-X [?].

APERTYPE	**# parms.**	**Meaning of parms.**
CIRCLE	1	radius
ELLIPSE	2	horizontal half axis, vertical half axis
RECTANGLE	2	half width, half height
LHCSCREEN	3	half width, half heigth (of rect.), radius (of circle)
RECTELLIPSE	4	half width, half heigth (of rect.), horizontal half axis, vertical half axis (of ell.)
RACETRACK	3	radius, horizontal shift, vertical shift

The LHC-screen and rectellipse definitions are the intersection of two of the other geometrical shapes, either circle and rectangle or ellipse and rectangle. The rectangle cuts the other shape, and for a point to be considered as inside the aperture, it must be inside both the rectangle and the circle/ellipse. Figures 5.3 to 5.8 shows the different beam screen descriptions.

We recognize that all existing aperture types, apart from the racetrack, can be considered as special cases of the rectellipse shape. Only the parameters need to be adjusted, and tables 5.2 to 5.7 explains for each apertype which parameter is read and how the others are adjusted. The adjustments are done in the function `build_pipe`, see Appendix A.3.5.

ap1 = half width rectangle

ap2 = half height rectangle

ap3 = half horizontal axis ellipse (or radius if circle)

ap4 = half vertical axis ellipse

5.2.1 Circle

Table 5.2: Parameter treatment for circle.

ap1	**ap2**	**ap3**	**ap4**
=ap3	=ap3	read	=ap3

A circle (Figure 5.3) is a common description of geometrical aperture. Many beam screens may be approximated by a circle without making too large errors.

5.2.2 Ellipse

The elliptical description (Figure 5.4) may be used i.e. when the beam screen is wider than it is high, but not cut in the vertical direction like the LHCscreen or rectellipse descriptions.

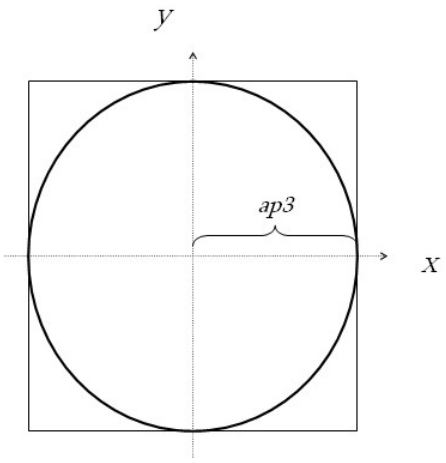


Figure 5.3: Circular beam pipe description.

Table 5.3: Parameter treatment for ellipse.

ap1	**ap2**	**ap3**	**ap4**
=ap3	=ap4	read	read

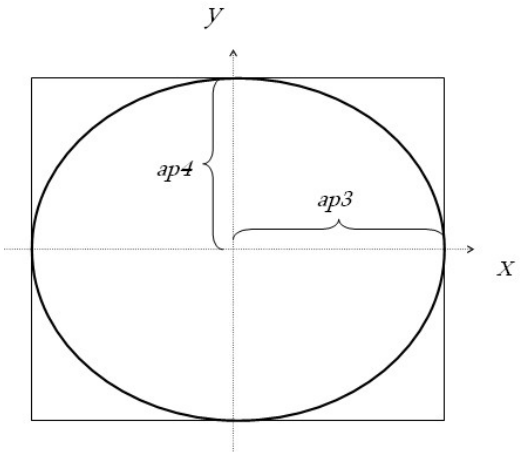


Figure 5.4: Elliptical beam pipe description.

Table 5.4: Parameter treatment for rectangle.

ap1	**ap2**	**ap3**	**ap4**
read	read	$=\sqrt{ap1^2 + ap2^2}$	$=\sqrt{ap1^2 + ap2^2}$

5.2.3 Rectangle

The rectangle (Figure 5.5) is another approach of a beam screen that is wider than it is tall.

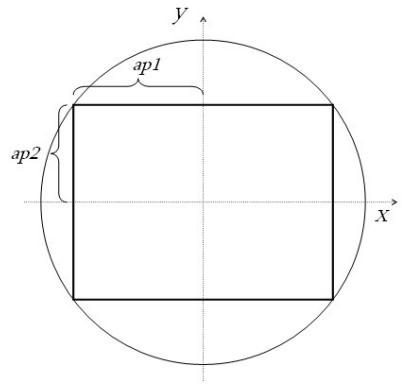


Figure 5.5: Rectangular beam pipe description.

5.2.4 LHCscreen

Table 5.5: Parameter treatment for LHCscreen.

ap1	ap2	ap3	ap4
read	read	read	=ap3

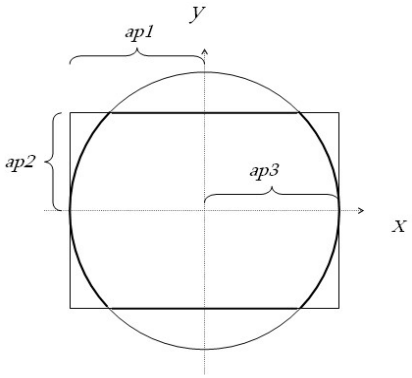


Figure 5.6: The LHCscreen. Beam pipe description for some LHC magnets. Not widely used elsewhere.

The LHCscreen (Figure 5.6) is a circle cut in vertical and/or horizontal directions, and is so a subcase of the rectellipse, see below. The LHCscreen was meant to be the main LHC apertype, and most elements are defined in the MAD-X LHC sequence file with parameters corresponding to a LHCscreen.

5.2.5 Rectellipse

Mechanical properties of some bending magnets made necessary an aperture description somewhat different than LHCscreen. All elements in the MAD-X LHC sequence file is at the time of writing defined with “rectellipse” apertype

Table 5.6: Parameter treatment for rectellipse.

ap1	ap2	ap3	ap4
read	read	read	read

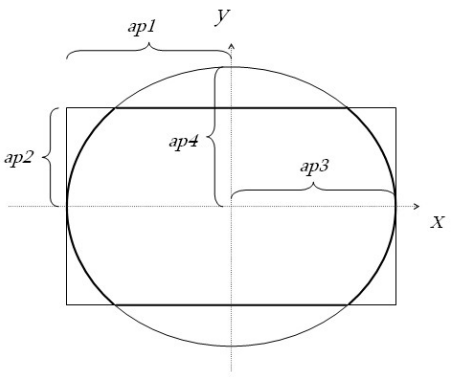


Figure 5.7: The rectellipse. Currently a common beam pipe description in the LHC.

(Figure 5.7). Most of them had their aperture parameters defined when the LHCscreen was used and are therefore simply translated by adding a parameter. They still have a circle as the basic shape.

5.2.6 Racetrack

Table 5.7: Parameter treatment for racetrack.

ap1	ap2	ap3	ap4
xshift	yshift	radius	not used

A so called "racetrack" (Figure 5.8) is the latest addition to the MAD-X apertype family. Some beam screens in the LHC will have this shape, but until now an approximation has been made by using the rectellipse apertype. The racetrack beam screen may be chosen whenever a large aperture is needed in the 45° azimuthal area.

5.3 Rectellipse-based beam screen polygons

For all apertypes the `make_rectellipse` function is called to build the polygon. This function retrieves addresses with the adjusted aperture parameters, two arrays to store x and y coordinates and an address to store the number of coordinate pairs computed for the first quadrant of the finished shape.

```
int make_rectellipse(double* ap1, double* ap2, double* ap3,
double* ap4, int* quarterlength, double tableX[], double tableY[])
```

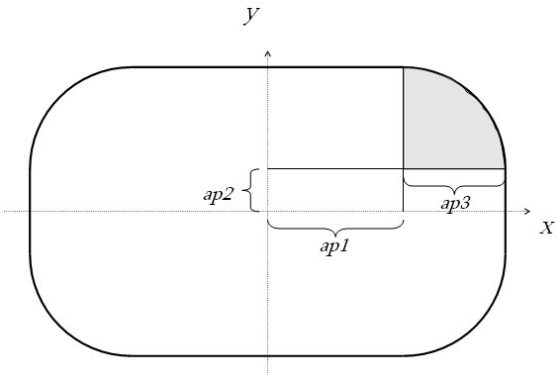


Figure 5.8: Racetrack, a "new" beam pipe description. 45° azimuthal area in grey.

The first quadrant coordinates are later mirrored across the x- and y-axes to complete the polygon.

The parametric representation for a point C at the outline of an ellipse is given by

$$(a\cos\varphi, b\sin\varphi), \text{ or in our case } (ap3\cos\varphi, ap4\sin\varphi)$$

We need only to find a certain number of coordinates on the arc that is actually inside the rectangle. Figure 5.9 shows a common example of the rectellipse. The ellipse is cut by the rectangle in the vertical direction, but not in the horizontal direction (ap1=ap3). We want to start our algorithm with point

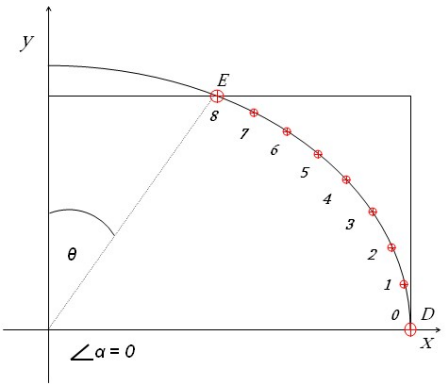


Figure 5.9: A common example of rectellipse. The number of apexes for the polygon is set to 20 in the source code, which gives a close approximation to the geometrical values for all realistic arc lengths.

D (angle $\alpha = 0$) and end at point E (angle θ). At the crossing point D the x value is known and we need to calculate the y value. Vice versa at point E. The equation for an ellipse is used for this:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

or in our case

$$\frac{x^2}{(ap3)^2} + \frac{y^2}{(ap4)^2} = 1.$$

So for

$$y = ap2; \quad x = ap3 \cdot sqrt(1 - \frac{ap2}{ap4}),$$

$$\theta = \frac{\pi}{2} - \tan^{-1}(\frac{ap2}{x})$$

and for

$$x = ap1; \quad y = ap4 \cdot sqrt(1 - \frac{ap1}{ap3}),$$

$$\alpha = \tan^{-1}(\frac{y}{ap1})$$

An arbitrarily chosen number of x and y coordinates from point D and upwards along the arc to point E are then put into the respective tables, and the number of coordinate pairs are saved as "quarterlength". This function is also used for finding the beam halo coordinates, since the beam halo is simply a circle or ellipse cut in the x and y directions. More on that in Chapter 6.4.

The tables with coordinates for the first quadrant are then sent to the function `fill_polygon_quadrants`, which receives a table and its length. The first-quadrant tables are there mirrored across the x and y axes. Extra precaution is taken to do this in the correct order. If PipeX[5] is the first point in the second quadrant, it will have the same coordinate (but different sign for x) as PipeX[4], not PipeX[0]. For a visual example, see chapter 6.4, Fig 6.7 and 6.8. The simple method of mirroring is explained and presented in the source code. Finally, the length of the halo tables is saved for use in other functions.

5.4 Racetrack beam screen polygon

In the case of the racetrack apertype, the tables built are corresponding to the first quadrant of a circular beam screen, and needs a displacement according to the ap1 (= horizontal shift) and ap2 (= vertical shift) values. To do this, the function call is embedded in a loop. Note how the value of ap3, the radius, is sent as the three first parameters to make the initial shape:

```
else if (!strcasecmp(apertype,"racetrack"))
{
  *ap1=get_aperture(current_node, "var1"); /*half width rect*/
  *ap2=get_aperture(current_node, "var2"); /*half height rect*/
  *ap3=get_aperture(current_node, "var3"); /*radius circle*/
  *ap4 = *ap3;

  err=make_rectellipse(ap3, ap3, ap3, ap4,
                           &quarterlength, PipeX, PipeY);

  if (!err)
  {
       /*displaces the quartercircle*/
       for (i=0;i<=quarterlength;i++)
       {
            PipeX[i] += (*ap1);
            PipeY[i] += (*ap2);
       }

       fill_polygon_quadrants(PipeX, PipeY, quarterlength,
                                                    pipelength);
  }
}
```

As shown in Figure 5.10 the result is a PipeX and PipeY table that contain coordinates for the first quadrant. The full shape is made with the function `fill_polygon_quadrant`, as shown in the source code excerpt.

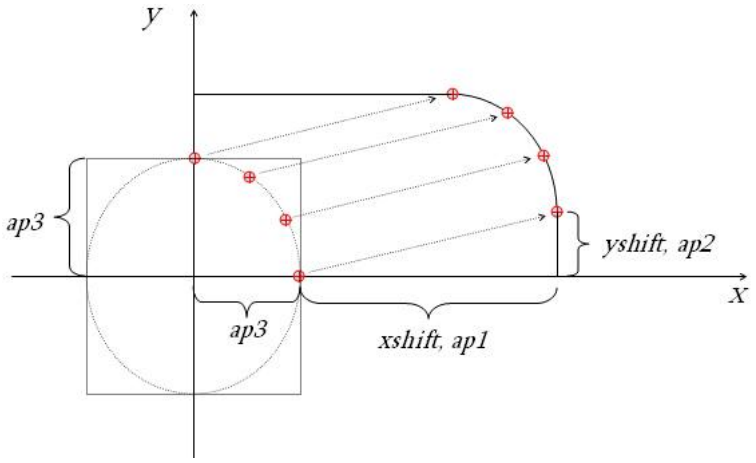


Figure 5.10: The shape from a circular rectellipse calculation is displaced to form a racetrack-like beam screen polygon.

The future may bring even more elaborate beam screens in new accelerators. Support for these should be fairly easy to implement by adding a section for reading parameters in the `build_pipe` function, and a standalone function to make the tables with apex coordinates.

Chapter 6

Beam halo

We finalize our understanding of what a particle beam is. Then we look into problems arising from loss of particles, and how this can be avoided by cleaning the halo with collimators.

6.1 Normalized coordinate system

Accelerated bunches of particles is referred to as a *particle beam*, or simply *beam*. The particles in the beam are always oscillating around their respective central trajectories, but in average the beam has a centre of mass which serves as the reference point for describing its volume. The distribution of particles is usually considered to be gaussian around this centre, and a point in the transverse space is denoted rather with coordinates normalized to the particle density distribution of the beam than with x and y:

$$n_x = \frac{x}{k_\beta \sigma_x} \qquad n_y = \frac{y}{k_\beta \sigma_y}, \text{ [9]} \tag{6.1}$$

where

$$\sigma_{x,y} = \sqrt{\beta_{x,y} \epsilon_{x,y}}$$

is the r. m. s. beam size of the density distribution. The emittance ϵ is a constant value, but β changes along the ring. k_β is an uncertainty factor of the β value. (See chapter 2). The total distance from the beam centre is denoted as:

$$n_r = \sqrt{{n_x}^2 + {n_y}^2}$$

6.2 Beam loss problems

For several reasons we want to remove particles at large n_r values from the beam:

1. To protect the beam pipe and machinery in the accelerator from physical damages.

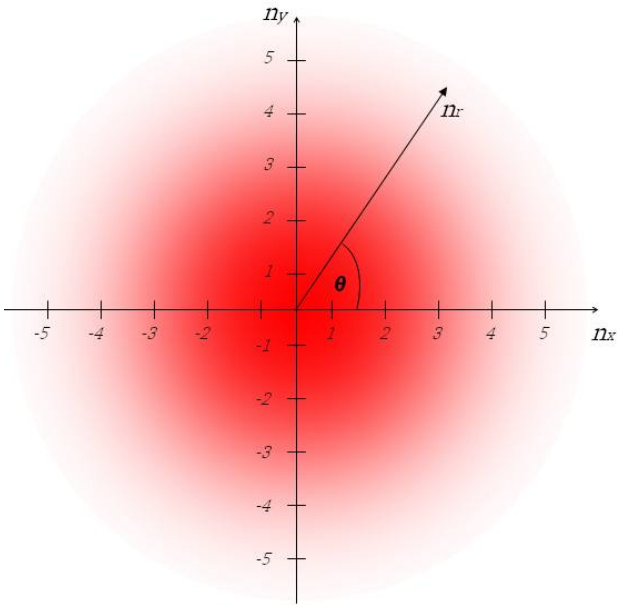


Figure 6.1: The density of particles in the beam has a gaussian distribution. The figure shows how normalized coordinates are used to describe a point in the transverse plane relative to the beam centre.

At high energy levels, a stream of particles hitting the beam pipe wall can cause severe damages. Reparations are both expensive and time consuming, due to the irradiated environment and the necessary level of thoroughness. In the LHC, the total energy per beam will be ca. 350 MJ, enough to melt 500 kg. of copper!

2. To ensure high beam quality for the experiments.

   Even at much lower energies than in the LHC, a particle interacting with the pipe wall can create a shower of other particles, drifting in the pipe. These particles may act as background noise in the experimental areas. This also adds to the radiation level in the machinery.

3. In the case of a superconducting environment, to minimize the amount of energy deposited on the pipe walls.

   Superconducting magnets need to be kept at a low temperature, and are vulnerable to heat energy from the inside. (This also necessitates the cooling system around the beam pipe, see Figure 5.2). This is probably the strictest criterion for beam cleaning in the LHC, where the time-integrated amount of energy deposited in a given length of machinery shall not exceed the limit which increases the temperature of the coil above the critical temperature T_c, which is the temperature limitations arising from the efficiency of the cooling system. If the temperature in a superconducting magnet rises above T_c, the magnet becomes resistive and the temperature will rise quickly due to the large current that is now flowing through a

resistive material. A magnet may be totally destroyed by this "quench" or "burn-out" in the absence of adequate protection. Several hours will be needed to restore working conditions even with adequate protection.

Particles with an oscillation amplitude larger than the dynamical aperture (see Chapter 2) is considered to be part of what is called the beam *halo*. An accelerator is designed to easily fit the dynamic aperture inside the geometrical one, so the three main reasons above concerns the particles constituting the beam halo. One of the main reasons for our aperture computations is that we want to keep track of this halo, to investigate to which degree our goals of protecting the machine is fulfilled. Our method of doing this is by making a polygon approximation of the halo outline, and comparing this outline with the geometrical aperture along the ring. In order to do this, we must study what gives shape to the halo.

6.3 Collimation

The removal of particles must be ensured at all times, since the halo is constantly fed. This feeding can e. g. be due to errors in the magnetic field causing a miskick of particles, particle-particle interactions in the beam or interactions with particles in the residual gas (in a non-perfect vacuum). The removal is obtained by inserting physical obstacles in the beam pipe at convenient spots (Straight sections, before experiment regions, etc.). The number, positioning and material of these obstacles, the *collimators*, is carefully decided in accordance with the wanted performance criteria. In the LHC, these will be 20 and 100 cm long graphite blocks, arranged to cut the halo at approximately $n_r = 6$ (Figure 6.2).

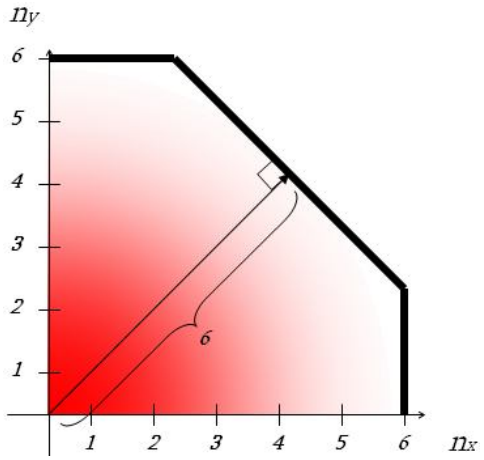


Figure 6.2: Primary collimators shown for the first quadrant.
The beam halo is cleaned at $n_r = 6$ at best.

The cleaning efficiency of a collimator is never 100%, so after passing the primary collimator there are still a fraction of particles left in the halo area. This is now referred to as the secondary halo. This halo is constituted either by

particles that escaped the primary collimator by passing right through it (multiple coulomb scattering), or by new particles created from particle interaction in the collimators (Figure 6.3).

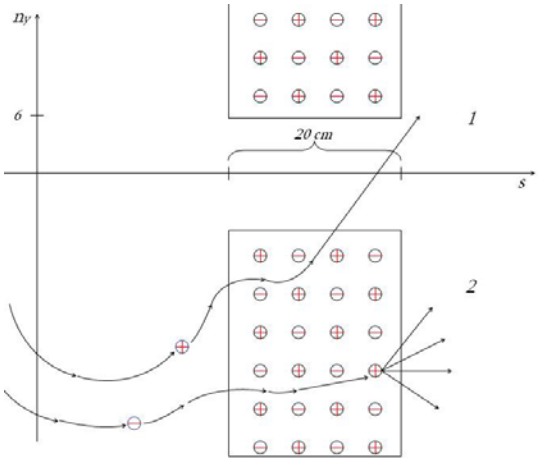


Figure 6.3: Multiple coulomb scattering (1) and direct particle interaction (2).

The same phenomena occurs in eventual secondary or tertiary collimators, so there a tertiary or quartiary halo is also created. The density of particles decreases, so depending on the objective all halos might be interesting to track around the accelerator. The halo polygonization routine implemented in MAD-X is based on the LHC collimation system. This system uses two sets of collimators, a primary and a secondary, where the primary ones are shaped as in Figure 6.2. The secondary collimators are positioned at $n_r = 7$. The halo produced by this system has been carefully studied (Figure 6.4), and the shape and parameters chosen to describe the halos is shown in Figure 6.5.

6.4 Halo polygon

When a parameterized halo is given, coordinates of a polygon must be calculated for numerous purposes. The function `make_rectellipse` receives the values r, h and v for the halo, and fills the tables `HaloX[]` and `HaloY[]` with the coordinates.

We do not add coordinates along the straight sections. A large number of coordinate pairs along the arc are not necessary, but it is important that two of the pairs are placed respectively at the start and at the end of the arc. (Figure 6.6).

The two angles α and θ are calculated in order to know the start and end of the arc, and coordinates in-between are calculated at even intervals and added to the tables. The `make_rectellipse` function uses the ellipse equation is basis for this:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

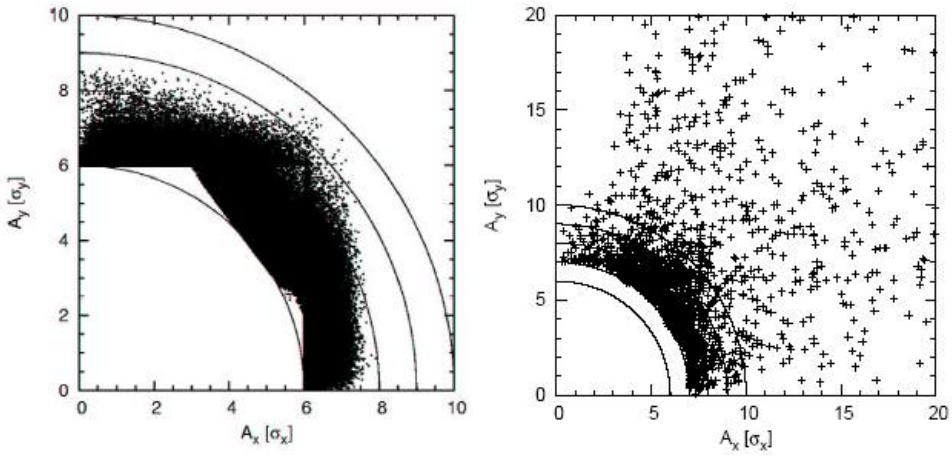


Figure 6.4: Simulated secondary and tertiary halo after primary and secondary collimators in one of the LHC collimation regions. [10]

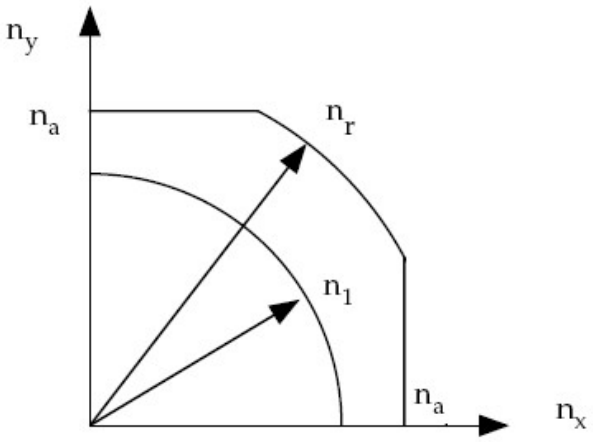


Figure 6.5: First quadrant of primary and secondary halo descriptions used in the LHC.

or in this case

$$\frac{x^2}{(r)^2} + \frac{y^2}{(r)^2} = 1.$$

So for

$$y = v; \quad x = r \cdot sqrt(1 - \frac{v}{r}),$$

$$\theta = \frac{\pi}{2} - \tan^{-1}(\frac{v}{x})$$

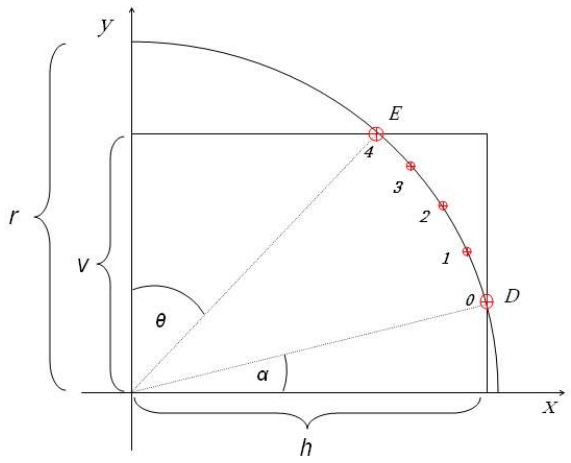


Figure 6.6: The halo is built based on parameters given in an external file.

and for

$$x = h; \quad y = r \cdot sqrt(1 - \frac{h}{r}),$$

$$\alpha = \tan^{-1}(\frac{y}{h})$$

Utilising the symmetry of the beam halo, it is not necessary to calculate more than the first quadrant in this manner. Coordinates in the other quadrants will have the same values but different signs, conversion is done with functions `fill_polygon_quadrants` and `adj_coord_quadrant`, which fill the quadrants and adjust the signs according to the angle,.

The resulting polygon, when drawing straight lines from point 0 → point 1, point 1 → point 2 etc. is what the `aperture_calc` function works with when comparing to a beam pipe polygon. Possible miscalculations due to the difference between the straight-lined polygons and the actual shapes is negligible compared to error margins included in the displacement of the halo. (See chapter 7).

The user also has the possibility to define a halo polygon with a different basic shape by giving as a parameter in the MAD-X aperture command a file containing apex coordinates for the entire polygon. It is, however, difficult to foresee what type of halo cannot be described adequately with the rectellipse approximation, since this gives the opportunity of both circular halos, and halos stretched in the x or y direction.

6.5 Halo oscillations

In the case of LHC, the halo is described as a superposition of a circle and a square, see Figure 6.5. The parameters describing the halo are given by the user in values normalized to beam size. As a standard, MAD-X uses SI-units, so we

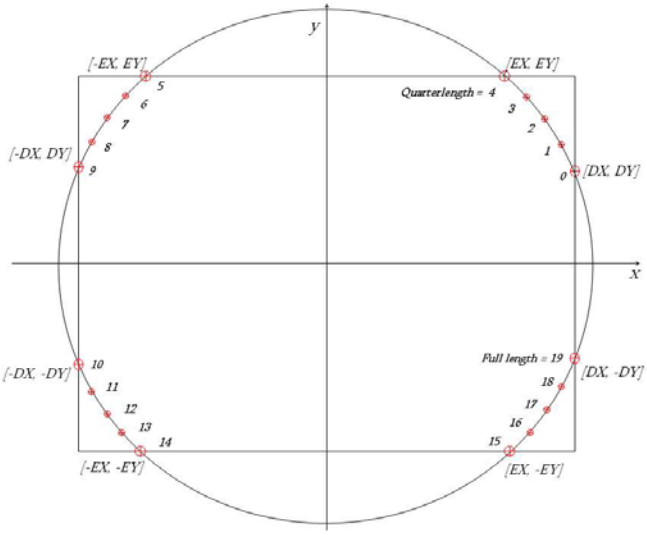


Figure 6.7: The coordinate tables for the first quadrant are expanded by copying coordinates and changing signs according to the new quadrant.

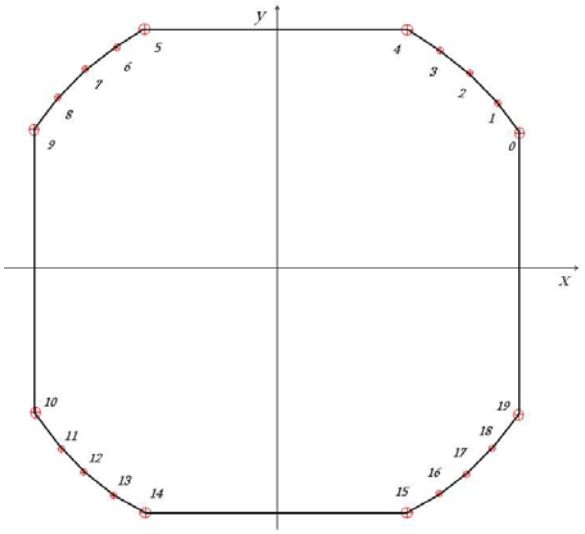


Figure 6.8: Final halo polygon. This is a very close approximation to the theoretical halo as given in the parameter file.

need to convert the halo polygon coordinates. The convertion is done by the function `adj_halo_si`:

$$x = n_x k_\beta \sqrt{\beta_x \epsilon_x} \qquad y = n_y k_\beta \sqrt{\beta_y \epsilon_y} \tag{6.2}$$

As β changes, the shape of the halo polygon should change accordingly, as seen in Figure 6.9. The halo polygon is therefore reconstructed, using equation 6.2, for each change in the $\beta_{x,y}$ values:

```
for (i=0;i<=halolength;i++)
{
     HaloXsi[i]=HaloX[i]*bbeat*sqrt(ex*betx);
     HaloYsi[i]=HaloY[i]*bbeat*sqrt(ey*bety);
}
```

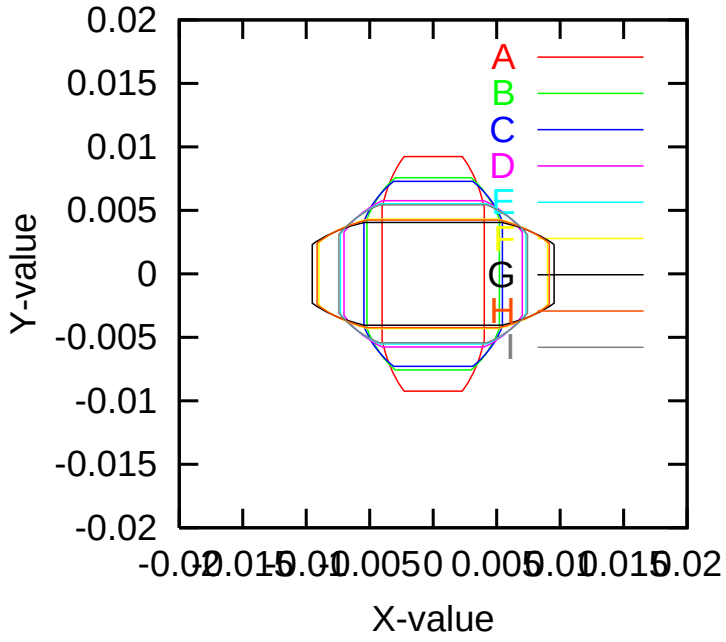


Figure 6.9: Halo shape changes from A=centre of defocusing quadrupole to H=centre of focusing quadrupole, according to the beta function.

Internally, MAD-X operates with four tables, `HaloX[]` and `HaloY[]` for the initial halo constructed from the given paramters r, h and v, and `HaloXsi[]` and `HaloYsi[]` for the coordinates adjusted to SI-units.

Chapter 7

Error tolerance and special cases

An introduction to different sources of error, and how they are taken into account by the aperture module.

7.1 Total displacement

The center of the beam will seldom be exactly superimposed on the center of the beam pipe. This is due to momentum dispersion arising from the optics of the machine, and to a parasitic dispersion component from magnetic field errors and small rotations of magnets. In addition to this, there is an uncertainty in the alignment and physical shape of the beam pipe, and an uncertainty about where the central trajectory of the beam is located at a particular energy level. Also, sometimes the beam is displaced on purpose, e.g. at injection or in experimental regions. All these factors must be included in the aperture calculations, to have a result which is on the safe side as long as the maximum tolerance levels are respected. All these errors are 2-dimensional, they can contribute in both x and y directions. In practice, the errors are therefore summed up, and the halo centre is radially displaced with the sum $\Delta_{x,y}(s)$ for an arbitrary number of angles $\in [0, 2\pi]$(Figure 7.1).

$$\Delta_{x,y}(s) = CO_{x,y}^{peak} + \delta_{x,y}^{tol}(s) + k_\beta \cdot D_{x,y}(s) \cdot \delta_p + [d_{x,y}^{sep}(s) + d_y^{inj}(s) + d_x^{axis}(s)] \tag{7.1}$$

In the following subchapters each constitutional part in equation 7.1 is elaborated, along with its relation to MAD-X.

7.2 Closed orbit errors, $CO_{x,y}^{peak}$

The CO is the design orbit of the "perfect particle" which has energy $p = p_0$ and $x = x' = y = y' = 0$. Particles with an non-zero initial value will perform betatron oscillations around this orbit. In a perfect machine, the perfect particle would pass through the center of all multipole magnets, not feeling any magnetic

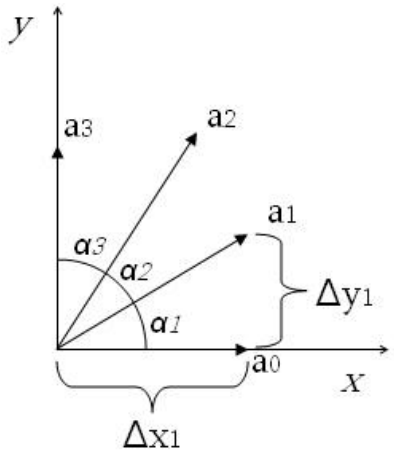


Figure 7.1: The displacement $\Delta_{x,y}(s)$ is computed for each point a_n.

force except the main bending field in the dipoles. In reality, the closed orbit is always shifted and/or complicated by irregularities in the magnetic fields of the magnets. (E.g superconducting magnets will experience residual currents. When shutting down the magnetic field, the current will continue to flow because of the low resistivity, creating a weaker field. When switching the power back on, it will take some time before the current and the residual current stabilizes, meanwhile the field is unstable; $\dot{B} \neq 0$). Figure 7.2 shows the extra kick received by particles performing oscillations around a closed orbit which misses the centre of a quadrupole. These particles experience the force ΔF in addition to the normal focusing force F. This gives the closed orbit a kick, creating oscillation everywhere downstream.

An uncertainty value for the closed orbit trajectory should be defined before calculating aperture. For the LHC, $CO_r^{peak} = 4$mm. This value should be given as a parameter in the aperture command. The parameter given is a radial distance from origin, and is decomposed in an x and y fraction for each angle a_n (Figure 7.1).

7.3 Mechanical and alignment tolerances, $\delta_{x,y}^{tol}(s)$

Each element along a beam line is constructed and mounted in accordance with strict rules of precision. Magnet design and measuring techniques are carried out as carefully as possible. Still, the perfect machine has yet to exist. We distinguish two separate possibilities of error:

1. Mechanical errors in the magnet body.
2. Alignment errors in the mounting on the beam line.

Maximum tolerance for each error in each magnet is set according to the need of precision under operation. The normal description of tolerance is based on the alignment errors, and shown as a circle in the x,y plane. Mechanical

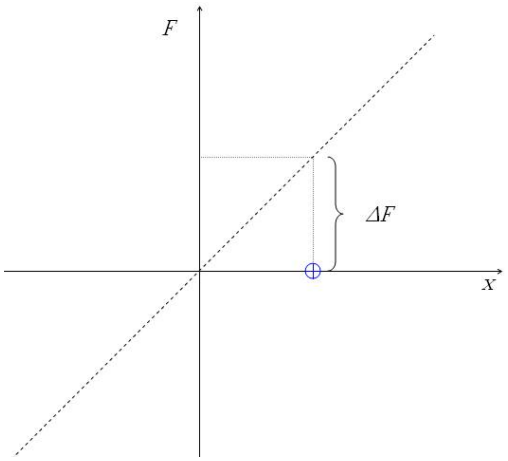


Figure 7.2: The magnetic field in a quadrupole is linearly changing and proportional to the distance from origin ($F = k \cdot x$). The dotted line shows the force ΔF a particle with distance x to origin experiences in a quadrupole. If the distance x is due to closed orbit or tolerance errors, the ΔF is considered a parasitic kick since it adds to the total bending strength of the machine.

errors in the magnet body are shown in tests to have even distribution around the magnet centre, so to include these the circle radius is enlarged. An exeption from this is the new LHC bending magnets. The LHC is the first ring at CERN where the bending dipole magnets are so long that they are not built with a straight body. Each dipole has a sagitta bend of 9 mm in the horizontal plane. This tends to enlarge the size of mechanical errors in this plane, see Figure 7.3. A circle is therefore no longer the best geometrical description of the maximum error around the magnet. A new description of maximum tolerance was for this reason recently defined in the LHC optics group. The beam aperture was kept, while making the tolerance narrower vertically and larger horizontally.

7.3.1 Racetrack algorithm

The new description, called *racetrack* (Figure 7.4), is made up of a rectangle and two semicircles. To describe a circle one can set $g = s = 0$, so this shape can also describe a usual circular tolerance definition. The function retrieves the g, s, r and angle values, and for each angle it calculates the distance from origin to the racetrack outline. The distance is for each angle returned as an x-parameter and a y-parameter.

To simplify the algorithm, all calculations are carried out as if they were in the first quadrant and are later adjusted to describe the full revolution. Describing the outline with one set of equations would be complicated, some sort of step-function would be needed. It is easier to divide the shape in three parts; vertical straight - arc - horizontal straight. A different set of equations is used for each part, determined by comparing angle ϕ with angles α_0 and α_1, see Figure 7.5. The algorithm is made bearing in mind that depending on the values for g and s, there might not be a vertical or horizontal straight part.

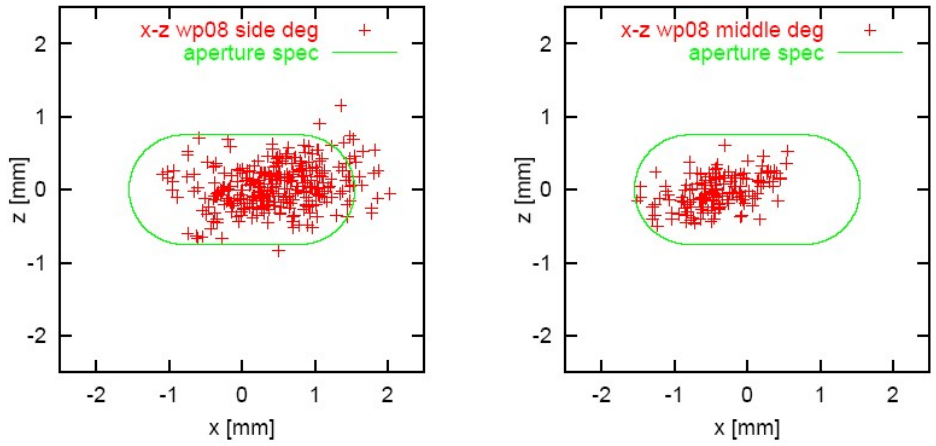


Figure 7.3: Mechanical and alignment error plots of sample bending magnets for the LHC. The maximum deviation in every magnet along their lengths is displayed in an horizontal(x)-vertical(z) plane. Here compared to the racetrack that gives the maximum allowed error.

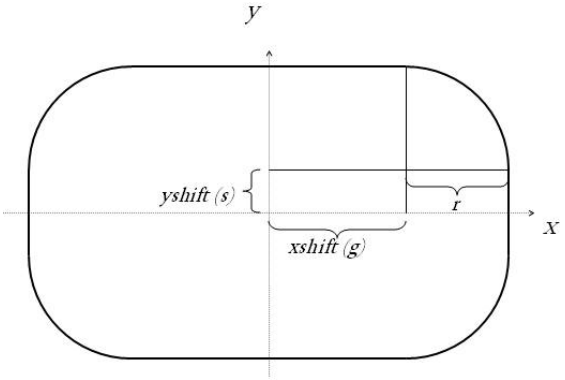


Figure 7.4: The xshift, yshift and radial parameters of the racetrack.

Precautions must be made when using trigonometric functions. The vector approach used in functions `check_if_inside` and `build_newhalo` can not be taken, since we now know the angle and want to find the vectors, not vice versa. For some angles, non-real values will be returned, namely "inf" or "nan" (not a number). In combination with the always present issue of dividing by zero, we need to think through wich values are possible to occur for each variable storing a radian value. Table 7.1 gives a list of dangerous radian values, computed with UNIX' standard GCC compiler, version 2.96.

"inf" has the advantage of being considered by the compiler to be bigger than any number. An "inf" value can therefore be compared successfully with other numeric variables using operands > or <. "nan" can not be used reasonably in any arithmetic operation.

Table 7.1: Problematic radian values.

ϕ	$\sin\phi$	$\cos\phi$	$\tan\phi$	$\sin^{-1}\phi$	$\cos^{-1}\phi$	$\tan^{-1}\phi$
0.00	0.00	1.00	0.00	0.00	1.57	0.00
1.57	1.00	0.00	inf	nan	nan	1.00
3.14	0.00	-1.00	-0.00	nan	nan	1.26
4.71	-1.00	-0.00	inf	nan	nan	1.36
6.28	-0.00	1.00	-0.00	nan	nan	1.41

In the first quadrant, we only have the two angle values $\phi = 0.00$ and $\phi = \pi/2$ to worry about, in addition to taking care not to divide by zero. The angles α_0 and α_1 dividing the quadrant in three parts is always calculated:

$$\alpha_0 = \tan^{-1}(s/(g+r))$$

$$\alpha_1 = \tan^{-1}((r+s)/g)$$

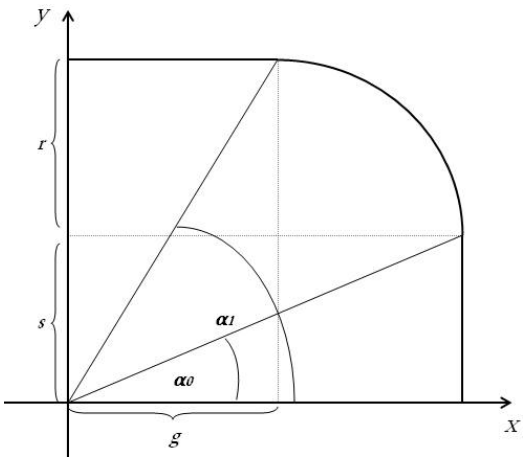


Figure 7.5: The racetrack outline is divided into three parts; vertical part, arched part and horizontal part.

If there is no xshift, $\alpha_1 = \tan^{-1}(inf)$, which is 1.57 rad, or 90.0 degrees. With no yshift, $\alpha_0 = \tan^{-1}(0.00)$, or 0.00 degrees. These results shows that the algorithm is able to treat circle outlines also.

The equations for calculating distances up to α_0:

$$x = g + r;$$

$$y = g + r \cdot \tan(\phi)$$

If $\phi = 0, \tan(\phi) = 0$, and the y parameter is correct.

Distances along the arc: We know length r, and can find angle α and length q (Figure 7.6). Angle α is the absolute value of the difference between the angle we want to calculate the distance to the outline for, and the angle to the centre of the radial part of the figure, namely α_2.

$$\alpha_2 = \tan^{-1}(s/g)$$

$$\alpha = |\phi - \alpha_2|$$

$$q = \sqrt{g^2 + s^2}$$

This is used to find the length of line l:

$$r/\sin\alpha = q/\sin\gamma$$

$$\gamma = \sin^{-1}(q/r \cdot \sin\alpha)$$

$$\theta = \pi - (\alpha + \gamma)$$

$$l/\sin\theta = r/\sin\alpha \Rightarrow l = r \cdot \sin\theta/\sin\alpha$$

And, finally, the x and y coordinates:

$$x = l \cdot \cos\phi$$

$$y = l \cdot \sin\phi$$

Obviously, we would have a division-by-zero problem with the former approach if $\sin(\alpha)$=0, which is to say, in our case, that α=0. But then there would be no triangle, but a straight line. In this case, we know that line l stretches from origin to the outline through the centre of the radial part:

$$x = (r + q) \cdot cos\phi$$

$$y = (r + q) \cdot sin\phi$$

7.4 Dispersion, $k_\beta \cdot D_{x,y}(s) \cdot \delta_p$

Here, k_β is a parameter describing the *beta beating*. This is a correction factor of the β value, taking into account worst-case deviations from the theoretical value. Since the amplitude of the halo oscillations (chapter 6.5) are proportional to $\sqrt{\beta}$, aperture calculations must be done with k_β-corrected values. The k_β is read from the input file of parameters, and is for the LHC assumed to be maximum 1.1.

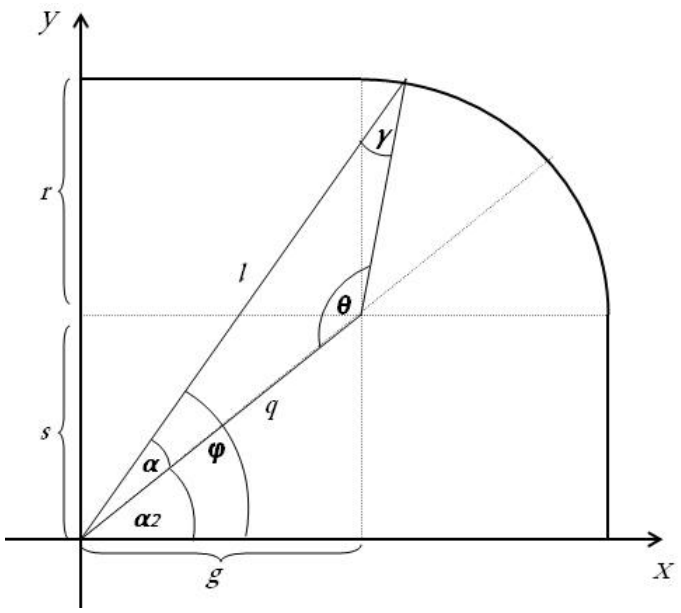


Figure 7.6: First quadrant of a racetrack. The sine rule is used to calculate distances from origo to the arched part of the outline.

The parameter $D_{x,y}(s)$ is the dispersion parameter presented in chapter 2. Wanting to allow for error margins, it is necessary to add a parasitic component to the "normal" dispersion.

$$D_{x,y}(s) = D^{lin}_{x,y}(s) + D^{par}_{x,y}(s)$$

Parasitic dispersion is a dispersion origining from coupling effects in quadrupoles. The coupling effects arise from slight misalignments of the magnet poles, or errors in the magnetic field, causing a particle with a deviation in one plane to also receive a small parasitic kick in the other plane. As a rule of thumb, $D^{lin}_{x,y}(s)$ is proportional to $\sqrt{\beta_{x,y}(s)}$ for all s. By knowing this proportion for a particular s, e.g. in a focusing quadrupole, the $D^{lin}_{x,y}(s)$ can be found for all s as long as $\beta_{x,y}(s)$ is known. This is used to calculate $D^{par}_{x,y}(s)$ as a fraction of $D^{lin}_{x,y}(s)$.

$$\frac{D^{par}_{x,y}(s)}{\sqrt{\beta_{x,y}(s)}} \approx \frac{k_D \cdot D_{QF}}{\sqrt{\beta_{QF,x}}},$$

where k_D is the parameter giving the fraction of parasitic dispersion. For aperture calculation purposes a "worst case" coupling factor k_D of 27% is used [9]. D_{QF}, $\beta_{QF,x}$ and k_D can be given as parameters in the aperture command. Finally, we have in Equation 7.2 an expression for $D^{par}_{x,y}(s)$. The $D^{lin}_{x,y}(s)$ is computed by the Twiss module.

$$D^{par}_{x,y}(s) = k_D \sqrt{\frac{\beta_{x,y}(s)}{\beta_{QF,x}} D_{QF}} \tag{7.2}$$

Total dispersion is calculated for only the first quadrant, and in addition the momentum distribution is symmetric around $\delta p = 0$, so to correctly add the dispersion to the displacement of the beam centre, the absolute value of dispersion must be given sign corresponding to the angle of calculation.

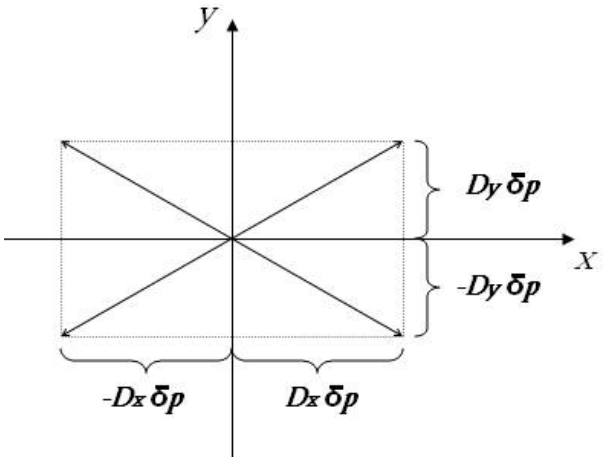


Figure 7.7: Displacement due to dispersion and δp. The worst-case displacement for each quadrant is shown as a vector arrow and decomposed in x- and y-vectors.

Table 7.2 is an outprint from MAD-X. The dispersions for x and y planes change sign according to which quadrant the computation angle is in.

Table 7.2: Displacement adjusted to worst-case for quadrant.

angle	dispxadj	dispyadj
0.000000	2.588831	0.251759
0.392699	2.588831	0.251759
0.785398	2.588831	0.251759
1.178097	2.588831	0.251759
1.570796	-2.588831	0.251759
1.963495	-2.588831	0.251759
2.356194	-2.588831	0.251759
2.748894	-2.588831	0.251759
3.141593	-2.588831	-0.251759
3.534292	-2.588831	-0.251759
3.926991	-2.588831	-0.251759
4.319690	-2.588831	-0.251759
4.712389	2.588831	-0.251759
5.105088	2.588831	-0.251759
5.497787	2.588831	-0.251759
5.890486	2.588831	-0.251759

7.5 Purposely made displacements, $[d_y^{inj}(s) + d_x^{axis}(s) + d_{x,y}^{sep}(s)]$

Sometimes the beam centre is displaced with regard to the beam pipe centre of practical reasons. The situations MAD-X must be able to take into account are at injection, in regions between separation kickers and interaction points of experiments, and when a magnet is shifted or tilted so its centre does not coincide with the ring axis. All three special cases are represented in the injection region for beam 2 in the LHC, Figure 7.8. Similar situations occur in many accelerators.

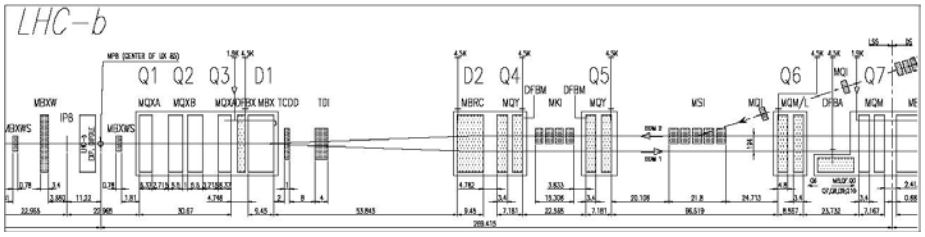


Figure 7.8: Rightmost is the injection of beam 2. Leftmost is IP8, where the two beams collide in the LHC-b experiment. [10] [11]

7.5.1 Injection and axis displacements

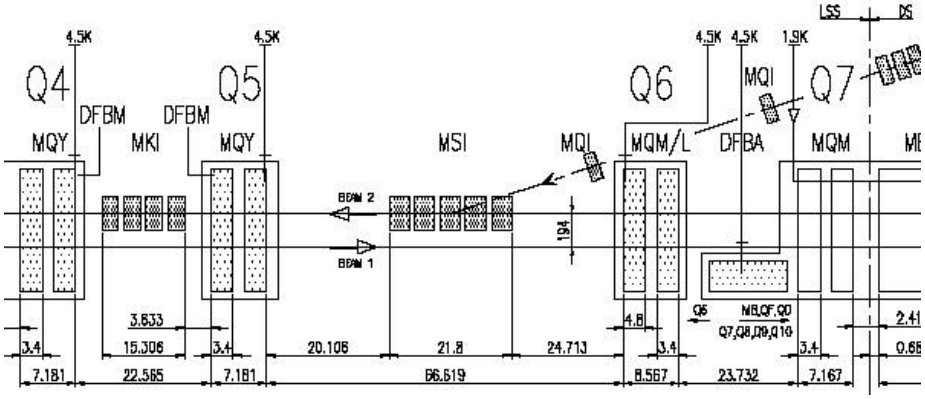


Figure 7.9: Enlarged right part of Fig 7.8; Injection region for beam 2.

Figure 7.9 shows the injection system for beam 2, seen from above. In the MSI section, the injected beam is aligned horizontally with the circulating beam, and inside the MKI section it is aligned vertically. The MKI's task is to kick the injected bunch upwards and into place on the circulation orbit. Two severe errors may occur here (Figure 7.10).

1. MKI does not kick the injected bunch, so it continues on a straight trajectory crossing the circulation orbit and finally hitting the pipe wall.

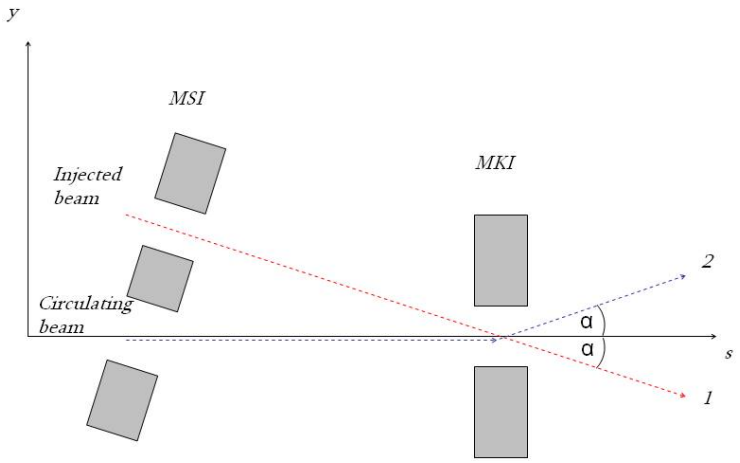


Figure 7.10: Possible errors in MKI. This particular case is LHC-specific, but similar situations will occur in other accelerators/colliders.

2. MKI kicks a circulating bunch out of its orbit.

Both errors possibilities must be studied to avoid the accidents mentioned in chapter 6.2. This is done by adjusting the kicker strength in the MAD-X sequence file. The distance $d_y(s)$ to the central orbit (the kicker works only in the vertical plane) is read from the Twiss output and added to Δ_y.

Another slightly delicate aperture problem arises in the D1 and D2 kicker magnets. These are the magnets that shifts the two beams into a common beam pipe (Figure 7.11). The beam trajectory changes direction significantly inside the magnets, so the aperture is far from constant (Figure 7.12). These problems will be addressed in the last part of the project, in 2005. The solutions will be included in the CERN-version of this report.

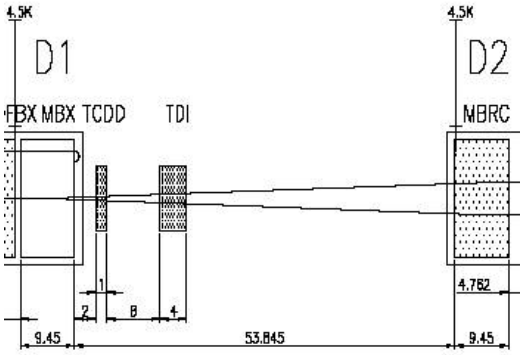


Figure 7.11: Enlarged middle part from Figure 7.8; D1 and D2 kickers.

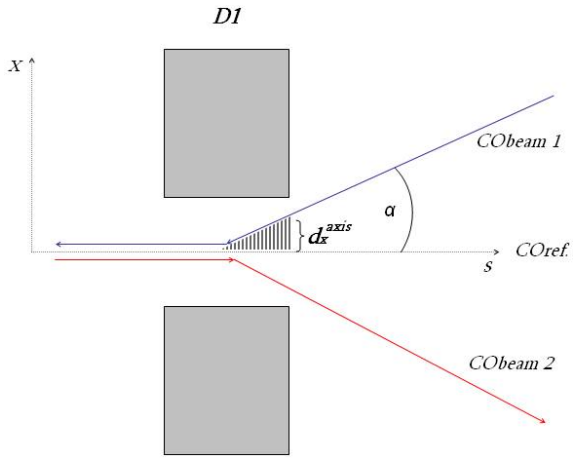


Figure 7.12: The angle between CO_{ref} and CO_{beam1} makes an aperture bottleneck.

7.5.2 Separation displacement

In regions just before and after beam colliding, the two beams have one common beam pipe (Figure 7.13). In these parts, beam positions are as before referenced

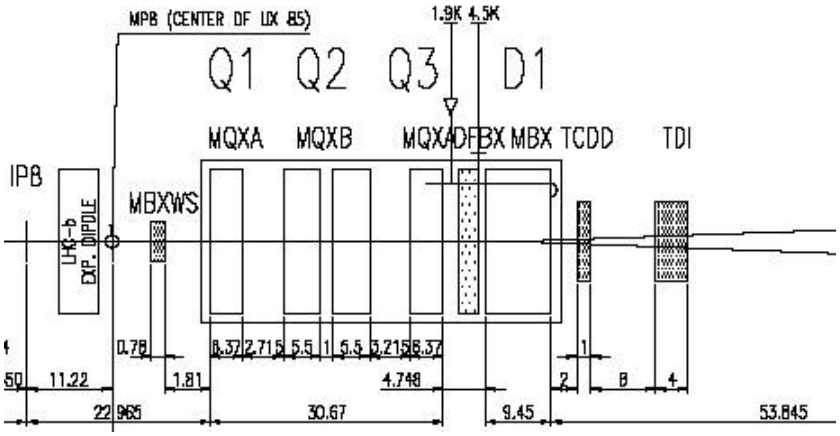


Figure 7.13: Enlarged left part from Figure 7.8; The quadrupole triplet to the right of IP8. The two beams share a common beam pipe.

to the center of the beam pipe, but the beams are displaced so to not interact directly before reaching the collision point (Figure 7.14). The separation is contained in the x and y values read from the Twiss output, since MAD-X automatically takes the common beam pipe into account.

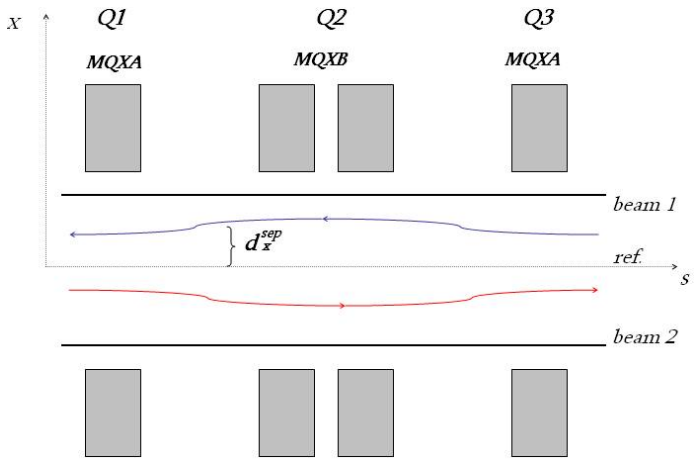


Figure 7.14: Inside the triplet the beam trajectories are guided by the quadrupoles, separated from the centre with a distance d_x^{sep}.

Chapter 8

Calculation procedure

The beam pipe and beam halo are now both approximated with polygons, and the coordinates are saved in their respective tables. The halo is shaped according to optic functions, and displaced w.r.t. the pipe centre according to uncertainty and tolerance margins. The next step is calculating, for a particular point along the accelerator, and with the shape and displacement of the halo, the largest possible halo that will fit inside the pipe.

8.1 Escaping beam halo?

Before the search for the largest possible halo can start, it is convenient to perform a check whether the centre of the beam halo is still inside the beam pipe. In the case it is outside, the aperture calculation will give incorrect answers. This investigation is based on the observation that the sum of angles between a point and a polygons apexes taken two by two determines whether the point is inside or outside the polygon. (Presuming the adding of the angles is done in fixed order with sign). See Figure 8.1.

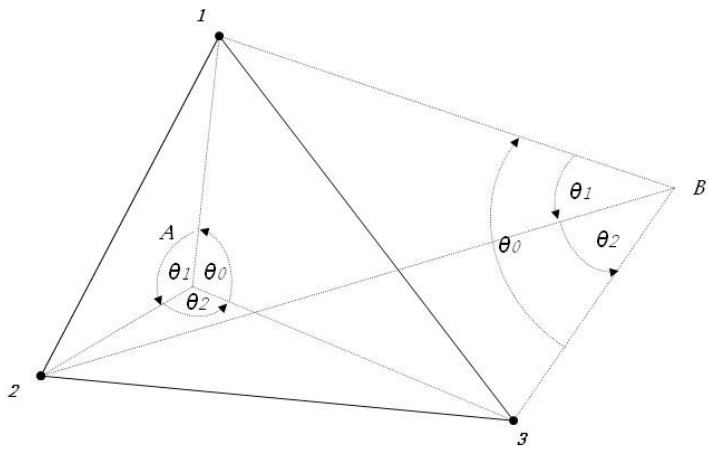


Figure 8.1: Summing up the angles gives either 0 or 2π.

For a polygon of n+1 apexes, $\sum_{i=0}^{n} \theta_i = 2\pi$ if the test point is inside the polygon, or $\sum_{i=0}^{n} \theta_i = 0$ if it is outside. To add up the angles with 360° rotation, we utilize the function `atan2(y,x)`. The regular `atan(a)` only takes a tangent as a parameter, and calculates the angle from that. This function can only return values between $0 \rightarrow \pi$. The `atan2` takes as parameters a sine and a cosine value, and determines the quadrant of the result from their signs.

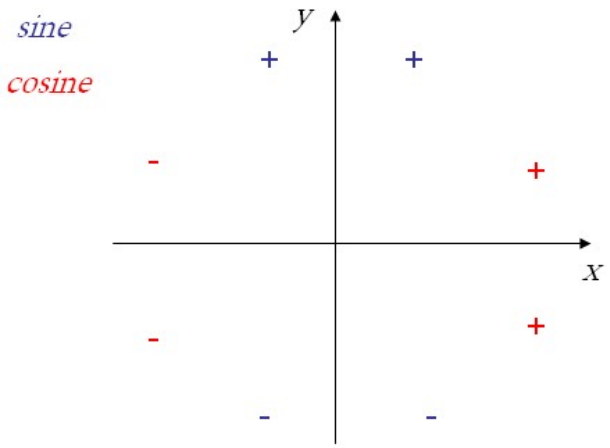


Figure 8.2: The sign of sine and cosine function of $\alpha \in [0, 2\pi]$.

To calculate the sine and cosine values with a sign, we use formulas 8.1 and 8.2[1].

$$\sin\phi = \frac{a_1 b_2 - a_2 b_1}{|\vec{a}||\vec{b}|} \tag{8.1}$$

$$\cos\phi = \frac{a_1 b_1 + a_2 b_2}{|\vec{a}||\vec{b}|} \tag{8.2}$$

The algorithm doing the work is coded in the function `check_if_inside()`.

8.2 "Dented" beam pipes: not simply connex polygons

Methodically, the algorithm for finding the largest possible halo is fairly simple. The distance from halo centre to the first apex ($i = 0$) in the halo is calculated (l_i), and the equation for a line going through these points is derived. This line is then compared with all lines making the pipe polygon to find their respective intersection coordinates. The distance h_i between halo centre and intersection are then divided by l_i, to find the maximal ratio of enlargement (Figure 8.3). This procedure is then repeated for all apexes i in the halo polygon, and the smallest ratio of all apexes is the maximal enlargement ratio for this halo to just touch the pipe at this particular longitudinal position.

[1] Formulas proven in Appendix C

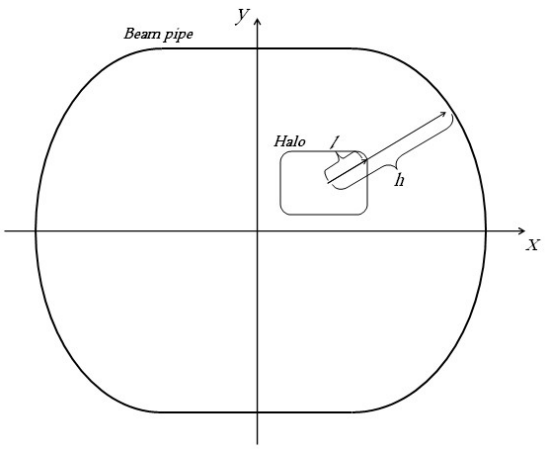


Figure 8.3: Largest possible size of halo is determined by $\frac{h}{l}$.

There is one complication to this solution; polygons which are not simple connexes. (Geometrical definition of "simply connex": A figure in which any two points can be connected by a line segment, with all points on the segment inside the figure.) Figure 8.4 shows what happens when a beam pipe polygon is not a simple connex. The halo is expanded in such a way that it is larger than the external polygon in the area where the latter is dented inwards.

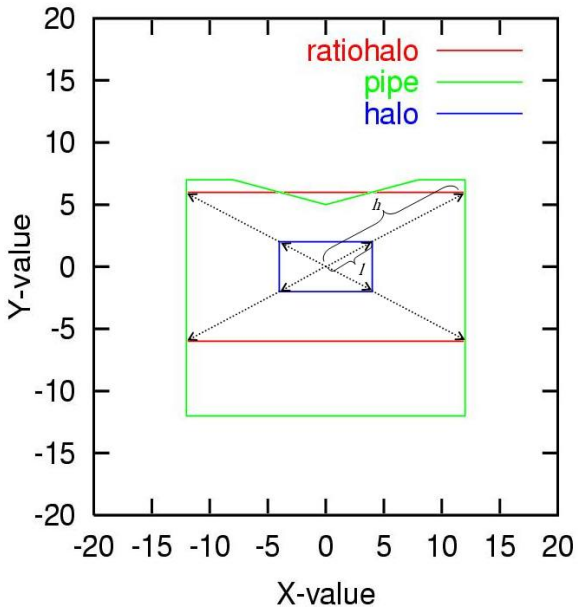


Figure 8.4: A maximum enlargement of the initial halo polygon based only on the apexes of the halo polygon.

To make the module able to treat all kinds of polygons, we need to strategically add apexes to the halo polygon wherever the beam pipe polygon might

have an inward dent. This is done by drawing a line from halo centre to each apex on the pipe polygon. An apex with its coordinates on the intersection point line-halo is added to the table of halo polygon apexes. The result is that the halo polygon has a few "excessive" points on straight sections as shown in Figure 8.5, but the algorithm used for expansion will now never miss a dent in the beam pipe.

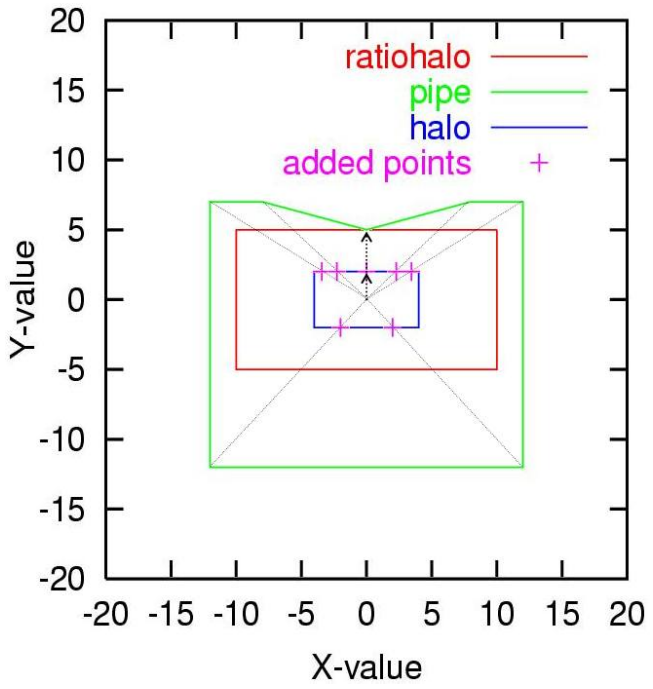


Figure 8.5: A maximum enlargement where also the apexes of the beam pipe is taken into account.

8.3 Tool functions

Adding extra apexes to the halo polygon and then comparing it to the pipe polygon, should bee seen as the two main parts of the calculation method. Based on the same geometrical technique of drawing and comparison of lines and points, the algorithm consists mainly of a few short functions. When it comes to the code, this should help making it easy to read and understand. The functions doing the work described in this chapter are `aperture_calc`, `intersection`, `linepar` and `online`.

8.3.1 Line equations

The basic equation describing a line in a two-dimensional plane is;

$$y = ax + b \tag{8.3}$$

Or defined w.r.t. a point (x_0, y_0);

$$y - y_0 = a(x - x_0) + b \tag{8.4}$$

Deducing parameters for a line through two points (x_1, y_1) and (x_2, y_2) with known coordinates is done quickly (see also Appendix A.3.11):

$$a = (y_1 - y_2)/(x_1 - x_2)$$
$$b = y_1 - a \cdot x_1 = y_2 - a \cdot x_2$$

Intersection point coordinates x_m and y_m between two lines are calculated as:

$$x_m = (b_1 - b_2)/(a_2 - a_1)$$
$$y_m = a_1 \cdot x_m + b_1$$

Vertical lines

One of the problems we run into when using this approach for intersection point calculation, is that when two lines are parallel, they have no intersection point. This is only a problem when $a_1 = a_2$ and $b_1 = b_2$, i.e. they are the same line. The solution is then to set the coordinates of the pipe apex as the intersection point.

Vertical lines are a weak point for this calculation approach, since the slope a will be infinity. A small list of checks for vertical lines and corresponding action is therefore added to the code, see Appendixes A.3.11 and A.3.10. As a general rule we state that if one of the lines *"halo centre - halo apex"* or *"pipe apex n - pipe apex n+1"* are vertical, then the x-coordinate of the intersection point defines the vertical line. The y-coordinate is then calculated from inserting the x-coordinate into the equation for the not-horizontal line. On the rare occasion that *both* lines are vertical they will only intersect if they have the same x-value. The intersection coordinates are then set equal to the coordinates of the start of the pipe line.

8.3.2 Intersection point verification

We know now the coordinates where the two lines meet. To check if the intersection point is on the polygon (i. e. in between the two current apexes) and not somewhere else along the line, we again use the formula for cosine calculation[2];

$$\cos(\phi) = \frac{\mathbf{CA} \cdot \mathbf{CB}}{|\mathbf{CA}||\mathbf{CB}|} \tag{8.5}$$

Here C is the intersection point, and A and B the start and end points of the segment, see Figure 8.6. If C is on the segment, $\angle ACB = \pi$. Else, $\angle ACB = 0$, or $\cos \angle ACB = -1$ or 1 respectively .

So Equation 8.5 will give either a 1 or a -1, depending on whether (x_m, y_m) is between the two apexes or on one of the sides. In the case that (x_m, y_m) is identical to one of the apexes, it will return 0, and the point is regarded as on the segment.

[2] See Appendix C

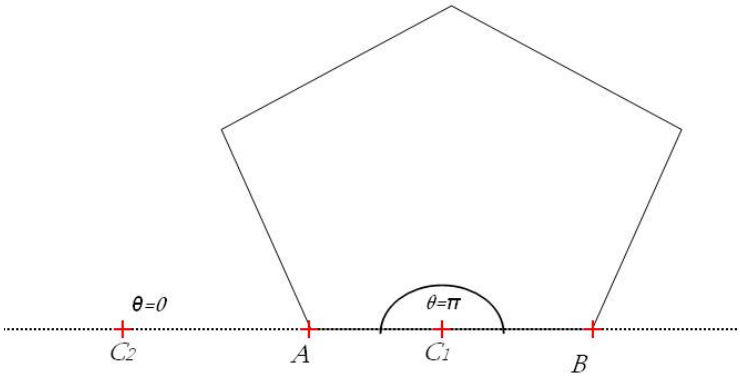


Figure 8.6: C is known to be on the line running through A and B. But is it on the segment AB?

8.4 Summary

Finally, the minimum ratio $r = \frac{h}{l} = |halo\,centre - pipe\,apex| \;/\; |halo\,centre - halo\,apex|$ gives the maximum allowed size of the halo at the current longitudinal position in the machine. The parameters describing the halo at this point may be multiplied by the ratio r, to give the beam size in σ at its maximum. After doing this for an entire magnet or region of machine, it is possible to predict if the beam pipe has a risk of absorbing so much energy from the halo particles that one of the accidents in Chapter 6 may occur.

Chapter 9

Output and project results

The aperture module is presented from the user point of view. As part of the pre-project a module requirement specification was written. It is here reproduced with some comments on goal achievements.

9.1 Output for the end-user

MAD-X can give three kinds of output to the user: Screen output, tables and plots based on the tables. To the screen, the module writes the aperture bottleneck, the name of the element containing it and its position. The output tables are filled with aperture-related parameters, such as apertypes, apertures, tolerances and Twiss parameters. Plots can be created by the user by including a call to the MAD-X plot-module in the input script, declaring which columns in which tables to plot.

For the aperture command in the input script, the user can set a number of parameters as explained in the user's guide. This includes the possibility to choose a start-point and an end-point for the computation. Presented is two examples, one of a regular LHC arc FODO cell and one from the injection region of beam 2, see Figure 7.9. We use the newest (as of Nov 15, 2004) LHC sequence files, version 6.5, with the manual addition of tolerances as they are not yet implemented in the official files. Also, all elements are not assigned an aperture because of lack of data and/or the lack of need to know the aperture for these elements.

A user's guide for the aperture module can be found in Appendix B. This will in the future be a part of the MAD-X user's guide [5]. The output tables and input files for the following examples can be found in Appendix D.5.

9.1.1 Regular arc cell

The regular arc cells are the simplest example, with few elements and a very regular aperture. Figure 9.1 shows the aperture from mb.a14r1.b1 to mcdo.a16r1.b1. The elements are symbolically plotted with dotted lines at the bottom. The user may give a "spec" value in the input, which is drawn as a line in the plot to easily compare apertures with the minimum specification. The screen ouput for this

example reads: APERTURE LIMIT: mq.14r1.b1:1, n1: 6.65091, at: 596.887, which is the quadrupole in the middle of the plot.

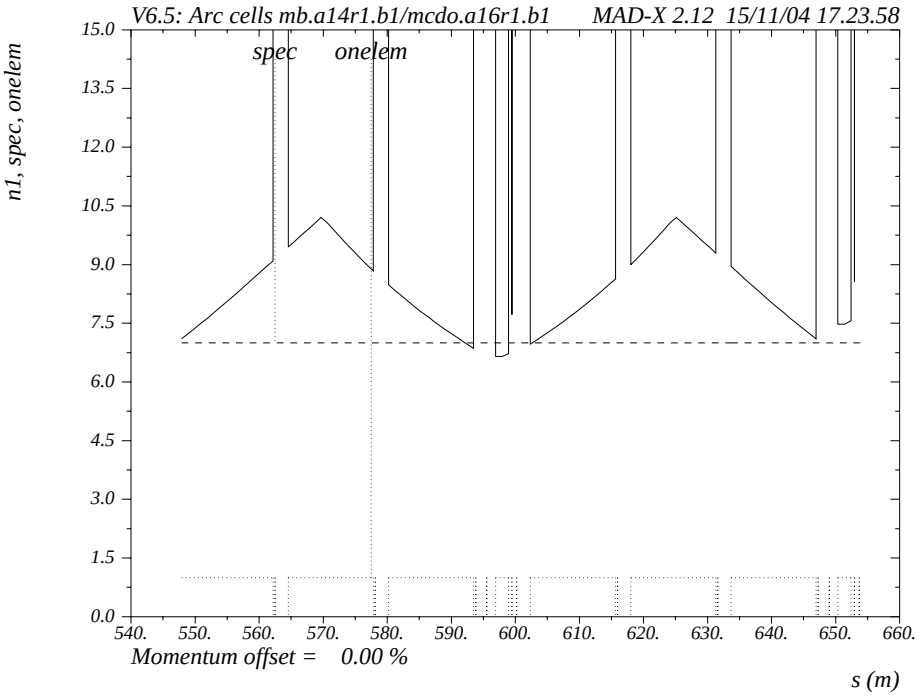


Figure 9.1: Aperture in a regular arc cell at. The six long blocks are dipoles, the two shorter ones are quadrupoles. The small aperture spikes after the quadrupoles are small chromatic sextupole magnets.

9.1.2 Collision region

An experiment region contains many smaller magnets and instruments, with large differences in aperture. The experiment region for the LHC-b detector(s) is shown in Figure 9.2, which shows two main blocks of aperture limitations. Each of the two blocks consists of a bending magnet (widest element) and four quadrupoles (Q1, Q2 and Q3 in Figures 7.13 and 7.14). Between the main parts, at $s \approx 3335$ is the point where the beams are collided. Here, the beam should be very small, to have as many particle collisions as possible. In order to achieve a low β-value here, the beam must be displaced towards the edges of the physical aperture before and after the collision point. The screen output reads: APERTURE LIMIT: mqxb.b2r2:1, n1: 6.86994, at: 3371.93, which means that the bottleneck is in the second set of quadrupoles.

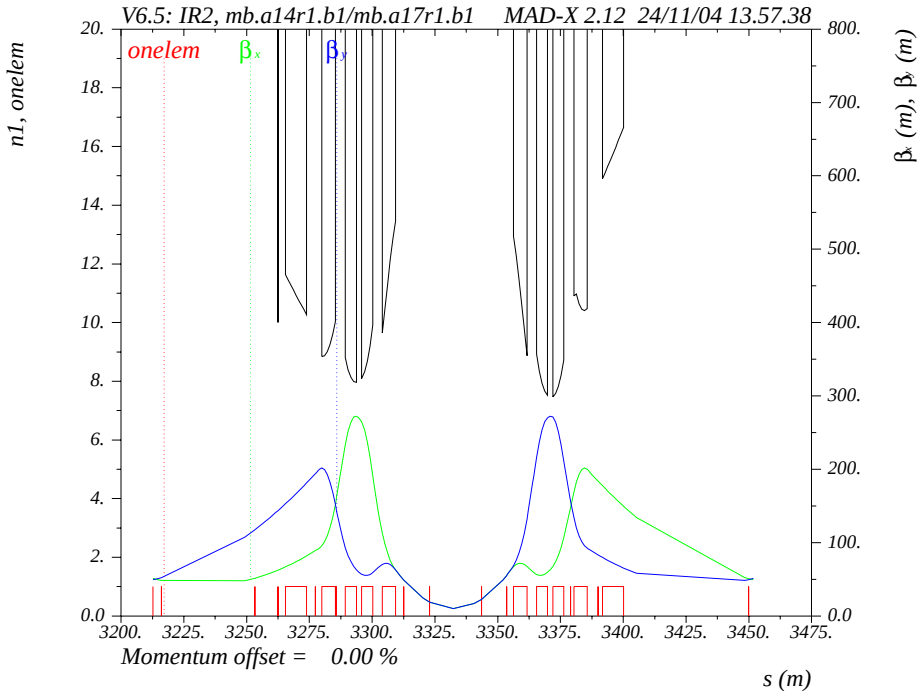


Figure 9.2: Aperture around the experiment point of LHC-b, from bpmwb.4l2.b2 to bpmwb.4r2.b2. The two beams have a common beam pipe in this region, the plot is valid only for beam two. The plot also shows how different table columns can be plotted in the same plot. We see how the beam must be displaced in the quadrupole triplets to achieve high density at the collision point.

9.2 Project status

As of mid-november 2004, the module is capable of aperture analysis of any accelerator structure implementable in MAD-X. Improvements can still be made, and more work will be done in the future to totally fulfil the module requirements.

9.2.1 Requirements and achievements

Finally, a summation of project achievements. The module requirement specifications were listed in the preproject, Appendix D.7. Here is a review and comments on these.

The module shall:

- Propagate some optics function across the machine elements, in particular those one which describe the beam envelope.

Achieved: Yes.
The module calls functions that slices each node and uses the Twiss module to compute optic parameters at short intervals in the machine.

- Displace the beam center with respect to the axis of the vacuum chamber in order to take into account mechanical and alignment errors, as well as beam closed orbit errors.

Achieved: Yes.
Mechanical and alignment tolerances, CO uncertainty, dispersion and beam separation in quadrupole triplets, all this is taken into account.

- Compute the aperture left free for the beam in normalized beam size units, with regards to both horizontal and vertical dimension.

Achieved: Yes.
The beam and beam pipe are described as polygons, the beam polygon displaced according to previous requirement, and the aperture is computed for both dimensions.

- Allow for the treatment of special cases, in particular when the axis of an element does not coincide with the beam axis.

Achieved: February 2005.
Not fully investigated, but these elements might not be compatible with the slicing functions used for the rest of the machine. A standard for MAD-X treatment of tilted elements must be decided.

- Provide aperture data for other MAD-X modules, in particular in order to allow for optics modelig with aperture constraints.

Achieved: Fully finished december 2004.
The aperture data are saved in an internal MAD-X table which can be accessed by any function. Also, n1 values are written to the twiss table for compatibility with the matching module.

- Provide a graphic output of the calculated aperture along the ring.

Achieved: Yes.
The aperture table is compatible with the plot module (and is actually the only table plotable apart from the Twiss table!), and produces its own style of machine symbols and a spec line to enhance the readability of the plot.

9.2.2 Future plans

The aperture module will be maintained and further developed in the ABP group. Future plans include:

1. Finalizing and bug-testing compatibility with the matching module to allow numerical search for a specific aperture, or to use the aperture as a constraint. This will let the user set an aperture in an element and make MAD-X search for the correct magnet strengths which gives this aperture, or to search for i.e. a low beta value at one place without going above a set aperture value at another place. An example of regions where low beta values are important is at experiment and injection points, as in Figure 9.2, which shows how the shrink of the beta values at the collision point can limit the aperture in the neighbouring magnets. Scheduled for december 2004.

2. Upgrade the module to be able to handle elements which are tilted or not centered on the closed orbit, as discussed in Chapter 7.5.1. Scheduled for january and february 2005.

The aperture module keeper will from march 2005 be J. Bernard Jeanneret.

Glossary

AB/ABP: Accelerators and Beam/Accelerators and Beam Physics.

Aperture: Unless otherwise stated: The amount of space between a particle beam and the beam pipe. Also called geometrical acceptance. With physical aperture is meant the beam screen surrounding the halo. Dynamical aperture is the limit where particle oscillations have a chaotic behaviour.

Apertype: Short for aperture type.

Beam screen: The area of a vertical slice of the beam pipe.

CERN: Organisation Européenne pour la Rechearche Nucléaire / European Organization for Nuclear Research

Collimator: A physical obstacle used to stop particles in the beam halo.

Dispersion: Trajectory difference from the reference particle due to difference in momentum.

Element: A piece of an accelerator, e.g. a magnet or an instrument. Everything else than empty driftspaces.

eV: electronVolt. The energy gained by an electron accelerated by a potential difference of 1 Volt. $eV/c = m \cdot c = momentum$

Hadron: Heavy subatomic particle, e.g. a proton or neutron.

Halo: The part of the beam outside the dynamical aperture.

i/o: input/output.

LHC: Large Hadron Collider.

Longitudinal direction: The direction along the machine.

MAD-X: Modular Accelerator Design, version 10.

Node: All elements of an accelerator, and the driftspaces, are called nodes inside MAD-X. Artificial nodes like markers (start or end) can also be created.

Optics: The focusing effects the magnets exerts on the particles is comparable to the optical effects of light through a prism.

STUC: Sør-Trøndelag University College

Superconduction: A state when the resistance in a material is for practical purposes equal to zero. This is achieved in the LHC when the Niob-Titanium coils are cooled to the operating temperature of 1.9° Kelvin.

Synchrotron: A circular accelerator which keeps the beam in a path that on the average has a fixed radius. The LHC and the circular accelerators in its injector chain are all synchrotrons.

TeV: Tera-electronVolt = 1 000 000 000 000 eV. 7 TeV $\approx$ the energy of a flying mosquito.

Trajectory: A (possible) path of a particle.

Transverse direction: All directions perpendicular to the longitudinal one.

Bibliography

[1] CERN webpage: http://www.cern.ch

[2] LHC webpage: http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/

[3] J. Rossbach and P. Schmüser, *Basic course on accelerator optics*, 1992. This article can be found in the "CERN Accelerator School, 5th General Acc. Physics course". (CERN 94-01, 26 Jan 1994, Vol. I)

[4] H. Wiedemann, *Particle accelerator physics.* Springer-Verlag 1993.

[5] Hans Grote, Frank Schmidt et al., *MAD-X Users Guide*, http://frs.home.cern.ch/frs/Xdoc/uguide.html for newest version. (Also included as Appendix D.6.)

[6] O. Brüning and W. Herr, *Problems and solutions of the exercises in the optics course at the CAS 2003 in Zeuthen, 15.-26.9.2003.*

[7] Image library of Superconducting magnet division at BNL: http://www.bnl.gov/magnets/

[8] O. Gröbner, *Beam screen design concepts*, a presentation given at VLHC Magnet Workshop, 24-26 May 2000.

[9] J. B. Jeanneret and R. Ostojic, *Geometrical acceptance in LHC Version 5.0*, LHC Project Note 111, 1997.

[10] *The LHC Design Report, Vol. I The LHC Main Ring*, CERN-2004-003

[11] LHC-b webpage: http://lhcb-public.web.cern.ch/lhcb-public/default.htm

Appendix A

Aperture module documentation

To give an overall view of the structure of the aperture module, pseudo code is written for the two most important functions. This includes function calls with variables and a reference number to the flowchart in Figure A.1. Another function with several subfunctions is `build_pipe`, but its structure is straightforward enough to be understood from the source code and comments. For maximum profit, the pseudo code should be read together with the real code.

A.1 Pseudo-code: aperture

```
struct aper_node* aperture(char *table, struct node* use_range[], struct
table* tw_cp, int tw_cnt[])


{   /* variables read from input script, or default value */
    halofile, pipefile, exn, eyn, dqf, betaqfx, nmom, dp, dparx, dpary,
    cor, bbeat, nco, nhalopar, interval, spec, mass, energy.
    ex, ey and dangle are calculated.

    external_file(halofile, halox, haloy) is called to check for the
    presence of a file with halo coordinates and read in the coordinates
    if successful.  [1]
    If a file is not present, the halo parameters are used to make
    halo coordinate tables, with a call to make_rectellipse(hhalo,
    vhalo, rhalo, rhalo, halo_q_length, halox, haloy) [2]
    Then fill_polygon_quadrants(halox, haloy, halo_q_length, halolength)
    is called to make the other three quadrants of the halo.  [3]

    /* Get initial twiss parameters, from the end of the node before
    the start-node */
    read_twiss_param(current_sequ→tw_table→name, jslice, name, s,
    x, y, betx, bety, dx, dy) reads the twiss parameters for the end
    of the previous node.  [4]
```

```
The twiss table row counter is increased.

adj_halo_si(ex, ey, betx, bety, bbeat, halox, haloy, halolength,
haloxsi, haloysi) is called to change the shape of the halo according
to the beta functions, and adjust it to SI-units.  [5]

Dispersions (normal and parasitic) for the beginning of the first
node are calculated from twiss parameters and input.

/*end of setup of first node start-values*/

As long as the stop flag is low:

{   The name of the node is saved.
    Node name converted to C-format by trim_ws(name, NAME_L).
    [6]

    Length of the node is fetched with node_value("l").  [7]
    double_from_table is called to fetch the s-value at the end
    of the node from the Twiss table.  [8]
    The s-value at the start of the node is calculated.
    The "current s" value is set equal to the start s value.

    The apertype of the node is saved.
    Node apertype converted to C-format by trim_ws(apertype, NAME_L).
    [9]

    A check for whether the current node is a drift is performed,
    and if this is the case the on_elem flag is set to -inf.

    /* read data for tol displacement of halo */
    get_node_vector("aper_tol",ntol,aper_tol) is called to read
    the tolerance values for the current node.  [10]
    If exactly 3 values are read, the tolerances are used in further
    calculations.  If not, the tolerances are set to 0.

    A check for an input file with pipe coordinates is performed
    with external_file(pipefile, pipex, pipey).  [11] If there
    is no such file, then build_pipe(apertype, ap1, ap2, ap3,
    ap4, pipelength, pipex, pipey) tries to make a pipe polygon
    [12].If this too is not successful:
    {   n1 is set to inf, on_ap is set to -inf.

        read_twiss_param(jslice, name, s, x, y, betx, bety, dx,
        dy) reads new twiss parameters.  [13]
        write_aperture_table(name, n1, r, xshift, yshift, apertype,
        ap1, ap2, ap3, ap4, minratio, nr, s, x, y, betx, bety,
        dx, dy, at) writes information to an internal table.  [14]
        on_ap is reset to default value 1.
```

```
   n1 is written to the Twiss table.
   The twiss table row counter is increased.
   /* Dispersion and si-adjusted halo is calculated to have
   values correct for the start of next node */
   Dispersions (normal and parasitic) are calculated from
   twiss parameters and input.
   adj_halo_si(ex, ey, betx, bety, bbeat, halox, haloy, halolength,
   haloxsi, haloysi) is called to change the shape of the
   halo according to the beta functions, and adjust it to
   SI-units.  [15]

}
However, if either an external file with pipe coordinates
was given, or coordinates could be calculated from the element
apertures with build_pipe, the n1 computation routine starts:
{  node_n1 is given initial value inf.

   The number of necessary slices is calculated.
   interp_node(nint) is called to slice the node.  [16]
   embedded_twiss() is called to make a Twiss table for the
   slices.  [17]

   For every slice:
   {  minratio and ratio are set to start value inf.

      If the slice is not the first in the node:
      { read_twiss_param("embedded_twiss_table", jslice,
        name, s, x, y, betx, bety, dx, dy) reads new twiss
        parameters.  [18]
        The "current s" is moved to the end of the slice.
        adj_halo_si(ex, ey, betx, bety, bbeat, halox, haloy,
        halolength, haloxsi, haloysi) adjusts the halo according
        to Twiss parameters.  [19]
        Dispersion (normal and parasitic) for both planes
        is also calculated.
      }
      The aperture is calculated for several angles in each
      slice:
      { adj_coord_quadrant(angle, dispx, dispy, dispxadj,
        dispyadj) adjusts dispersion to worst-case for the
        current angle.  (Sets correct signs).  [20]
        Finds displacement due to closed orbit uncertainty.
        race(xshift, yshift, r, angle, tolx, toly) finds displacement
        due to tolerance uncertainty.  [21]
        adj_coord_quadrant(angle, tolx, toly, tolxadj, tolyadj)
        sets sign to the tolerance values according to the
        angle.  [22]
        Total displacement due to uncertainties is calculated
```

```
                in both planes.
                The pipe polygon, halo polygon and total displacement
                is sent to aperture_calc(deltax, deltay, ratio, haloxsi,
                haloysi, halolength, pipex, pipey, pipelength, minratio)
                for calculation of the aperture in the slice.  [23]
                If the aperture found is smaller than the smallest
                previously found for other angles in the slice, it
                is saved.
              }
              Aperture parameters are calculated from the smallest
              aperture found.
              write_aperture_table(name, n1, r, xshift, yshift, apertype,
              ap1, ap2, ap3, ap4, minratio, nr, s, x, y, betx, bety,
              dx, dy, at) writes information to an internal table.
              [24]
              If the n1 found is the smallest found for this node,
              it is saved along with the corresponding Twiss parameters.
            }
            write_aperture_table(minimum, node_n1, r, xshift, yshift,
            apertype, ap1, ap2, ap3, ap4, minratio, nr, s, x, y, betx,
            bety, dx, dy, at) writes information to an internal table.
            This is the summary for the node.  [25]
            reset_interpolation(nint) is called to de-slice the node
            and consider it as a unity again.  [26]
            the node_n1 is written to the Twiss table.
            The twiss table row counter is increased.
            If the node_n1 is the smallest found for this range of
            elements, its position, aperture and tolerance parameters
            are saved.  This is the aperture bottleneck.
          }
          The stop flag is set either if the current node is the end
          of the range, or if
          advance_node() is not able to jump to the next node.  [27]
          Else, the loop begins again with the next node.
        }
        The return value is a pointer to a struct containing the information
        saved about the bottleneck node.
}
```

A.2 Pseudo-code; aperture_calc

```
double aperture_calc(double p, double q, double* minhl, double haloxcp[],
double haloycp[], int halolength, double pipex[], double pipey[], int
pipelength, double minratiocmp)
```

```
{   Halo polygon is displaced.
    Counter set to zero before future use.
    check_if_inside(p, q, pipex, pipey, dist_limit, pipelength) is
    called to investigate whether p,q is inside the pipe polygon.
    [21.1] If it is:

    {   /* Adding of extra apexes on the halo starts */
        For every apex on the halo:
        {   Add the old coordinate values.
            Then for every apex on the pipe polygon:
            {   linepar(p, q, pipex[i], pipey[i], a1, b1) finds parameters
                for the line centre -> pipe apex.  [21.2]
                linepar(halox[j], haloy[j], halox[j+1], haloy[j+1],
                a2, b2) finds parameters for a line on the halo polygon.
                [21.3]
                intersection(a1, b1, a2, b2, pipex[i], pipey[i], halox[j],
                haloy[j], ver1, ver2, xm, ym) finds the intersection
                coordinates for these lines.  [21.4]
                online(xm, ym, halox[j], haloy[j], halox[j+1], haloy[j+1],
                dist_limit) finds whether the intersection point is
                on the halo polygon.  [21.5]
                online(p, q, pipex[i], pipey[i], xm, ym, dist_limit)
                finds whether the intersection point and the current
                pipe apex is on the same side of the halo centre.  [21.6]
                If the last two tests are passed, the intersection
                coordinates are added as an apex in the new halo polygon.
            }
        }
        Number of apexes on the new halo is saved.

        /* The new halo is then compared with the pipe, to find the
        largest possible aperture /*
        For every apex on the pipe polygon:
        {   Treat every apex on the new halo polygon:
            {   linepar(p, q, newhalox[j], newhaloy[j], a1, b1) finds
                parameters for the line halo centre -> halo apex.  [21.7]
                linepar(pipex[i], pipey[i], pipex[i+1], pipey[i+1],
                a2, b2) finds parameters for a line on the pipe polygon.
                [21.8]
                intersection(a1, b1, a2, b2, newhalox[j], newhaloy[j],
                pipex[i], pipey[i], ver1, ver2, xm, ym) finds the intersection
                coordinates for these lines.  [21.9]
                online(xm, ym, pipex[i], pipey[i], pipex[i+1], pipey[i+1],
                dist_limit) finds whether the intersection point is
                on the pipe polygon.  [21.10]
                online(p, q, newhalox[j], newhaloy[j], xm, ym, dist_limit)
                finds whether the intersection point and the current
                halo apex is on the same side of the halo centre.  [21.11]
                If the last two tests are passed, the intersection
```

```
                coordinates are used to calculate the possible expansion
                of the halo polygon inside the pipe.  If this is the
                smallest expansion ratio yet found, it is saved.
            }
        }
        If p,q is outside the pipe, the expansion ratio is set to
        zero.
    }
}
```

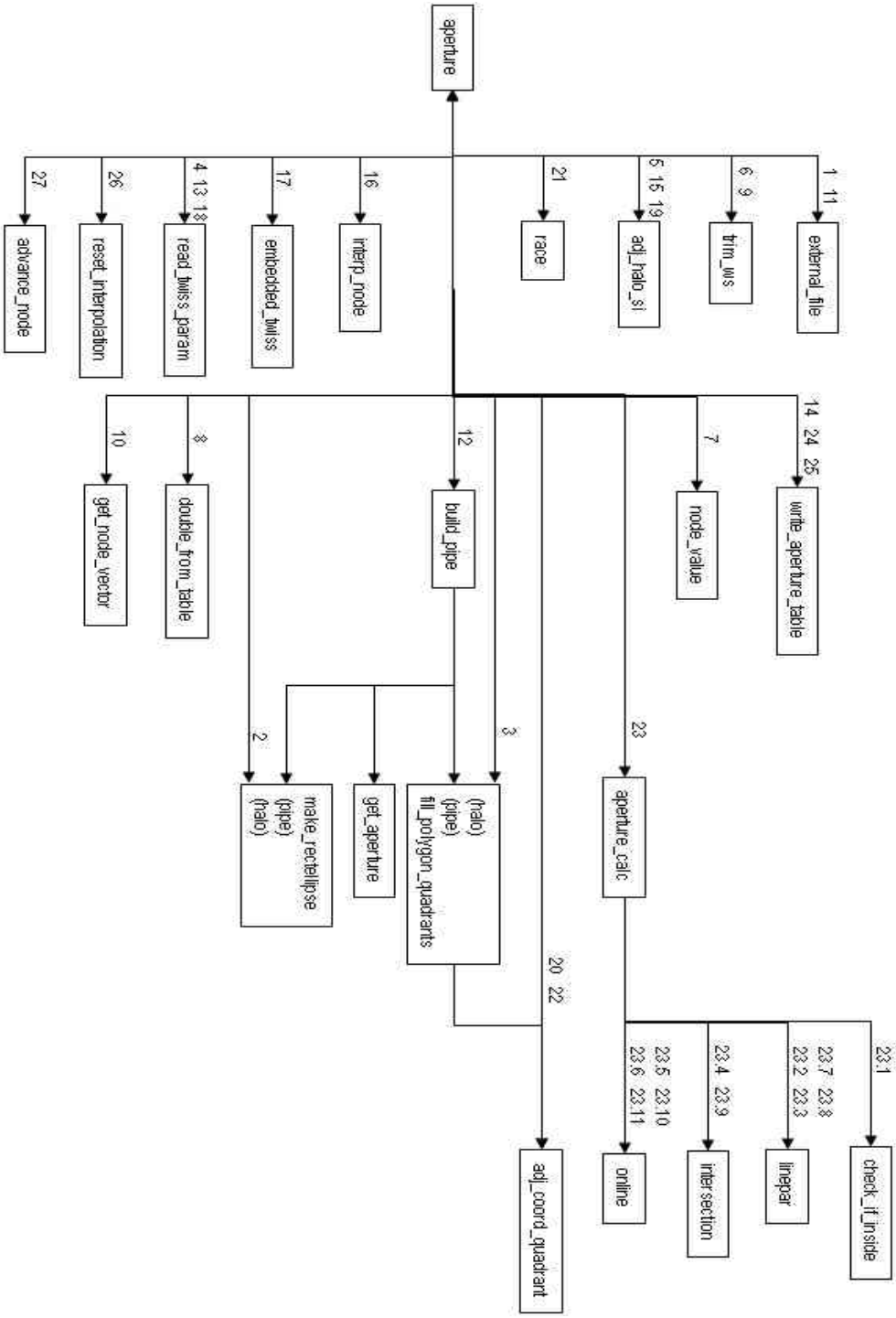


Figure A.1: Overview of the aperture module functions, including a few functions shared with other modules. The main data flows are numbered, see pseudo code for variable names.

A.3 Functions

A short description of all functions, their parameters, internal variables and return values. All functions are written by the author.

A.3.1 adj_coord_quadrant

Adjusts the sign of x and y coordinates according to which quadrant they belong in.

```
void adj_coord_quadrant (double angle, double x, double y,
                         double* xquad, double* yquad)
```

Description

Receives a coordinate pair and an angle value. The signs of the coordinates are changed according to which quadrant the angle is in. The coordinates must be given as if in the first quadrant, i.e. only positive values. The function is used to make an entire polygon after calculations for the first quadrant has been done, and to displace the dispersion values to worst-case for each quadrant.

Parameters

Table A.1: adj_coord_quadrant function parameters.

Parameter	Description
double angle	The angle in radians.
double x	x-coordinate
double y	y-coordinate
double* xquad	x-coordinate with adjusted sign
double* yquad	y-coordinate with adjusted sign

Internal variables

Table A.2: adj_coord_quadrant internal variables.

Variable	Description
int quadrant	Indicating the quadrant: 1, 2, 3 or 4.

Return value

No return value. Changes values of xquad and yquad.

Remarks

No remarks.

A.3.2 adj_halo_si

Adjusts halo coordinates from normalized to particle density to si-units (meters).

```
void adj_halo_si(double ex, double ey, double betx,
                 double bety, double bbeat, double halox[],
                 double haloy[], int halolength,
                 double haloxsi[], double haloysi[])
```

Description

Receives tables of normalized x- and y-coordinates and the optics parameters necessary to transform them into si-units. The transfer formula itself is simple, given by Eq. 6.1.

Parameters

Table A.3: adj_halo_si function parameters.

Parameter	Description
double ex	emittance in the x-plane, in meters
double ey	emittance in the y-plane, in meters
double betx	beta value in the x-plane, in meters
double bety	beta value in the y-plane, in meters
double bbeat	beta beating factor (Eq. 2.2)
double halox[ ]	array with x-coordinates of halo, in sigma
double haloy[ ]	array with y-coordinates of halo, in sigma
int halolength	number of coordinate pairs
double haloxsi[ ]	array with x-coordinates of halo, in meters
double haloysi[ ]	array with y-coordinates of halo, in meters

Internal variables

Table A.4: adj_halo_si internal variables.

Variable	Description
int j	counter

Return value

No return value.

Remarks

No remarks.

A.3.3 aperture

The main function of the aperture module.

```
struct aper_node* aperture(char *table, struct node* use_range[],
                           struct table* tw_cp, int tw_cnt[])}
```

Description

Gives the superior structure of the aperture module. Supervises input of parameters from files and output to tables. Jumps from node to node along the ring, retrieving data, slicing nodes and calculating aperture. Handles displacement due to error tolerances (Chapter 7), orders aperture calculations to start and keeps track of the minimum aperture for each node.

Parameters

Table A.5: aperture function parameters.

Parameter	Description
char *table	name of the aperture table
struct node* use_range	pointer to array with start and end node
struct table* tw_cp	pointer to a copy of the current Twiss table
int tw_cnt[ ]	array of pointer to n1 column and row in Twiss table

Internal variables

Return value

Returns a pointer to the node containing information on the node with the smallest n1 value.

Remarks

Space in memory for the arrays containing polygon coordinates is alloceted statically, once and for all at the start of the function. The same is the case in the `aperture_calc` function. Dynamic allocation would be to slow, since the polygons are recalculated thousands of times during a normal run. Increase in table size may be done by altering the definition of `MAXARRAY` in the file `madxd.h`.

Table A.6: aperture internal variables, part 1.

Variable	**Description**
int stop	Flag to stop the loop over nodes
int i	Saves return values from slicing functions
int nint	Number of slices for the node
int jslice	Counter for node slices
int halo_q_length	Number of coordinate pairs in 1. quadrant of halo
int halolength	Number of coordinate pairs in halo polygon
int pipelength	Number of coordinate pairs in pipe polygon
int namelen	Containing constant NAME_L
int nhalopar	Number of halo parameters given
int ntol	Number of tolerance parameters given
double on_ap	Flag indicating whether current node has a valid aperture
double on_elem	Flag indicating whether current node is a drift or not
double mass	Mass per particle in the beam
double energy	Energy per particle in the beam
double exn, eyn	Normalized emittance in horisontal and vertical planes
double dqf, betqfx	Linear dispersion and beta value in x-plane in a focusing quadrupole
double dp	Momentum deviation used for dispersion calculation
double dparx, dpary	Fraction of parasitic dispersion relative to linear dispersion
double cor	Circular closed orbit tolerance
double bbeat	Beta beating factor (Eq. 2.2)
double nco	Number of computation angles per quadrant
double halo[ ]	Array for halo parameters
double interval	Approximate length of node slices
double spec	Line in plot
double ex, ey	Emittance in x- and y-planes.
double s	Length along the accelerator, in meters
double x, y	Distance from reference trajectory, in meters
double betx, bety	Beta values, in meters
double dx, dy	Dispersion values, in meters
double ratio, minratio	Aperture calculated for a node, and its aperture limit so far
double n1, nr	Aperture parameters, see Figure 6.5
double length, at	Length and longitudinal position of a node, in meters
double node_...	Minimum values for the node
double aper_tol[ ]	Array for tolerance parameters
double ap	Pipe parameters, see Chapter 5.2, in meters

Table A.7: aperture internal variables, part 2.

Variable	**Description**
double dispx, dispy	Sum of linear and parasitic dispersion
double cox, coy	Circular closed orbit tolerance, and x and y coordinate pairs around the circle
double tolx, toly	Sum of mechanichal and alignment tolerances, in x and y coordinates
double dispxadj, dispyadj	total dispersion adjusted to a specific quadrant
double coxadj, coyadj	cox and coy adjusted to a specific quadrant
double tolxadj, tolyadj	tolx and toly adjusted to a specific quadrant
double angle	Counter for radian values.
double dangle	Interval for the angle counter
double deltax, deltay	Total displacement, see Chapter 7
double xshift, yshift, r	Parameters for racetrack, in meters. See Chapter 5.4
double halox[ ], haloy[ ]	Arrays with coordinates of halo, in sigma
double haloxsi[ ], haloysi[ ]	Arrays with coordinates of halo, in meters
double pipex[ ], pipey[ ]	Arrays with coordinates of pipe, in meters
char *halofile[ ]	Pointer to name of file with halo parameters
char *pipefile[ ]	Pointer to name of file with pipe parameters
char *minimum	Pointer to string with name of minimum slice in each node
char apertype[ ]	Apertype for the different nodes (Table 5.1 or [5]).
char name[ ]	Name for the different nodes
struct aper_node limit_node	Structure containing information about bottleneck node
struct aper_node* lim_pt	Pointer to bottleneck node

A.3.4 aperture_calc

Receives two polygons (halo and pipe) and calculates the maximum size of the halo, while still fitting inside the pipe.

```
double aperture_calc(double p, double q, double* minhl,
                     double haloxcp[], double haloycp[],
                     int halolength,
                     double pipex[], double pipey[],
                     int pipelength, double minratiocmp)
```

Description

Receives halo and pipe polygon, and the centre of the halo polygon relative to the pipe polygon. An imaginary line is drawn from the halo centre to each apex on the pipe, and a the coordinates where this line crosses the halo polygon is added as a new apex on the halo. Then new lines are drawn, from halo centre to each halo apex and further. The distance from halo centre to the coordinates where the new lines cross the pipe polygon is compared with the distance halo centre to the corresponding halo apex. The minimum ratio is saved, since this is the maximum number the halo size can be multiplied with, and still fit inside the pipe.

Parameters

Table A.8: aperture_calc function parameters.

Parameter	Description
double p, q	x- and y-plane displacement of halo polygon, in meters
double* minhl	The smallest ratio found
double haloxcp[ ], haloycp[ ]	Arrays with coordinates of halo, before displaced with p and q, in meters
int halolength	Number of coordinate pairs (apexes) on the halo
double pipex[ ], pipey[ ]	Arrays with coordinates of the pipe, in meters
int pipelength	Number of coordinate pairs (apexes) on the pipe

Internal variables

Return value

Returns a 0 on completion, returns -1 if the halo centre is outside the beam pipe.

Remarks

No remarks.

Table A.9: aperture_calc internal variables.

Variable	**Description**
int i	Counter, mostly used for pipe arrays
int j	Counter, mostly used for halo arrays
int c	Counter, general
int ver1, ver2	Flags indicating whether a line is vertical
int newhalolength	Length of the halo polygon after the adding of new apexes
double halox[ ], haloy[ ]	Arrays with coordinates of halo polygon, after displacement, in meters
double newhalox[ ], newhaloy[ ]	Arrays with coordinates of halo polygon after adding of new apexes, in meters
double dist_limit	Precision limit when determining whether two values are alike
double a, b	Line parameters
double xm, ym	Meeting point coordinates between two lines
double l	Distance from halo centre to an apex on the halo
double h	Distance from halo centre to pipe wall, through the apex on the halo

A.3.5 build_pipe

Provides a pipe polygon, given the apertype and aperture parameters for a node.

```
int build_pipe(char* apertype, double* ap1, double* ap2,
               double* ap3, double* ap4, int* pipelength,
               double pipex[], double pipey[])
```

Description

Receives the apertype of a node, and reads the corresponding aperture parameters. Sets up the data and sends it to another function which calculates coordinates for the firs quadrant. A third function is then called to make the total polygon.

Parameters

Table A.10: build_pipe function parameters.

Parameter	Description
char* apertype	Apertype for the different nodes (Table 5.1 or [5])
double* ap	Pipe parameters, see Chapter 5.2, in meters
int* pipelength	Number of coordinate pairs on the pipe polygon
double pipex[ ], pipey[ ]	Arrays with coordinates of the pipe polygon, in meters

Internal variables

Table A.11: build_pipe internal variables.

Variable	Description
int i	Counter
int err	Error flag, contains the return value from the function that calculates the geometry
int quarterlength	Number of coordinate pairs in the first quadrant of the pipe polygon

Return value

Returns `err`, which is 0 for successful completion.

Remarks

The racetrack apertype is treated as a circle. The coordinates in the first quadrant are then displaced before making the complete polygon.

A.3.6 check_if_inside

Calculates whether a point is inside or outside of a polygon.

```
int check_if_inside(double p, double q,
                    double pipex[], double pipey[],
                    double dist_limit, int pipelength)
```

Description

Receives x- and y-coordinates for a point, and coordinates for a polygon in the same space. Calculates then whether the point is inside or outside of the polygon. The method is explained in Chapter 8.1.

Parameters

Table A.12: check_if_inside function parameters.

Parameter	Description
double p, q	x- and y-plane displacement of halo polygon, in meters
double pipex[ ], pipey[ ]	Arrays with coordinates of the pipe polygon, in meters
double dist_limit	Precision limit when determining whether two values are alike
int pipelength	Number of coordinate pairs in pipe polygon

Internal variables

Table A.13: check_if_inside internal variables.

Variable	Description
int i	Counter
double n12	Numerator in equation for sine and cosine alfa
double salfa, calfa	sine and cosine alfa
double alfa	angle result with sine and cosine alfa

Return value

Returns a 0 if point is outside or on polygon, 1 if it is inside.

Remarks

No remarks.

A.3.7 external_file

Checks for the presence of an external file with either halo or pipe coordinates.

```
int external_file(char *file, double tablex[], double tabley[])
```

Description

Receives a filename, checks for the presence of this file, and then tries to read coordinates from it.

Parameters

Table A.14: external_file function parameters.

Parameter	Description
char *file	Name of file
double tablex[ ], tabley[ ]	Arrays to store coordinates read from the file

Internal variables

Table A.15: check_if_inside internal variables.

Variable	Description
int i	Counter for number of coordinate pairs

Return value

Returns i, the length of the arrays.

Remarks

No remarks.

A.3.8 fill_aperture_header

Fills the header of the aperture table.

```
void fill_aperture_header(struct table* aper_t,
                          struct aper_node* lim)}
```

Description

Receives a pointer to the aperture table, and a pointer to the bottleneck node. Fills the header of the aperture table with information on the values used for computation and the minimum aperture found.

Parameters

Table A.16: fill_aperture_header function parameters.

Parameter	Description
struct table* aper_t	Pointer to the aperture table
struct aper_node* lim	Pointer to the bottleneck node

Internal variables

Table A.17: fill_aperture_header internal variables.

Variable	Description
int i	Counter for characters and parameters
int h_length	Number of header lines
double dtmp	Temporary buffers for double values
double vtmp	Temporary buffer for double vector
char tmp[ ]	Temporary buffer for string
char *stmp	Temporary buffer for string

Return value

No return value.

Remarks

No remarks.

A.3.9 fill_polygon_quadrants

Makes a complete polygon from first quadrant coordinates.

```
void fill_polygon_quadrants(double polyx[], double polyy[],
                            int quarterlength, int* halolength)}
```

Description

Receives two table with coordinates for the first quadrant, and mirrors the across the x and y axes. Uses simple copying of values, and calls then the function `adj_coord_quadrant` to adjust the signs correctly.

Parameters

Table A.18: fill_polygon_quadrants function parameters.

Parameter	Description
double polyx[ ], polyy[ ]	Arrays containing polygon coordinates
int quarterlength	Number of coordinate pairs in the first quadrant
int* halolength	Number of coordinate pairs in total polygon

Internal variables

Table A.19: fill_polygon_quadrants internal variables.

Variable	Description
int i	Counter for quadrants 2, 3 and 4
int j	Counter for quadrant 1

Return value

No return value.

Remarks

No remarks.

A.3.10 intersection

Calculates the intersection point of two infinitely long lines.

```
void intersection(double a1,double b1,double a2,double b2,
                  double x1,double y1,double x2,double y2,
                  int ver1,int ver2,double* xm,double* ym)
```

Description

Receives line parameters a and b for two lines, and calculates their intersection point. Receives also start and endpoints for the linepieces, and whether the lines are vertical. In the case of a vertical, line, the intersection coordinates are set equal to the start point coordinates of the other line.

Parameters

Table A.20: intersection function parameters.

Parameter	Description
double a1, b1, a2, b2	Parameters describing the two lines
double x1, x2, y1, y2	Start and end coordinates for the line pieces
int ver1, ver2	Flag set in case a line is vertical
double* xm, ym	Intersection coordinates

Internal variables

No internal variables.

Return value

No return value.

Remarks

The function does not support all kinds of parallell lines, but this is not necessary for our purpose. In the case of parallell lines, xm will go to infinity, and will therefore not pass the tests in function `online`.

A.3.11 linepar

Calculates parameters a and b for a line going through two points.

```
double x1, double y1, double x2, double y2, double* a, double* b
```

Description

Receives coordinates for two points in space, and calculates the parameters a and b describing the line going through them.

Parameters

Table A.21: linepar function parameters.

Parameter	Description
double x1, y1	Coordinates for the first point
double x2, y2	Coordinates for the second point
double* a, b	Parameters describing the line

Internal variables

Table A.22: linepar internal variables.

Variable	Description
int vertical	A flag set if the line through the points is vertical

Return value

Returns the value of vertical, which is 1 if the line is vertical, else 0.

Remarks

No remarks.

A.3.12 make_rectellipse

Makes a polygon based on given parameters.

```
int make_rectellipse(double* ap1, double* ap2,
                     double* ap3, double* ap4,
                     int* quarterlength,
                     double tablex[], double tabley[])
```

Description

Chapter 5.2 shows how most beam pipe apertypes can be parametrized as a rectellipse. The halo polygon might also be a kind of rectellipse. Chapter 5.3 explains the theory behind the function. Only coordinates for the first quadrant are found.

Parameters

Table A.23: make_rectellipse function parameters.

Parameter	Description
double* ap	Pipe parameters, see Chapter 5.2, in meters
int* quarterlength	Number of coordinate pairs in the first quadrant
double tablex[ ], tabley[ ]	Arrays with polygon coordinates

Internal variables

Table A.24: make_rectellipse internal variables.

Variable	Description
double x, y	Coordinates for the points used to find angles alfa and theta
double angle	Counter for radian values
double alfa	Angle up to first point in the arc, in radians
double theta	Angle from last point in the arc to $\pi/2$, in radians
double dangle	Interval for the angle counter
double napex	One less than number of points on the arc

Return value

Returns a 0 on successful completion, returns -1 if the parameters given were corrupted.

Remarks

No remarks.

A.3.13 online

Determines whether a point is on a linepiece.

```
double online(double xm, double ym, double startx, double starty,
              double endx, double endy, double dist_limit)
```

Description

Receives start- and end-coordinates for a line piece, and coordinates for a single point in space. Uses Equation 8.5 to calculate whether the point is on the linepiece.

Parameters

Table A.25: online function parameters.

Parameter	Description
double xm, ym	Coordinates for the single point
double startx, starty	Start coordinates for the linepiece
double endx, endy	End coordinates for the linepiece
double dist_limit	Precision limit when determining whether two values are alike

Internal variables

Table A.26: online internal variables.

Variable	Description
cosfi	Result from Eq. 8.5

Return value

Returns `cosfi`, which is -1 if the test point is on the linepiece. If not, it is $>$ -1.

Remarks

No remarks.

A.3.14 pro_aperture

The "set-up" function for the aperture module.

```
void pro_aperture(struct in_cmd* cmd)
```

Description

The function is called from MAD-X' command interpreter, and receives the aperture command. Initialises an aperture table. Sets pointers to correct row and column in the Twiss table, for reading of Twiss parameters and writing of n1. Calls the `aperture` function which starts the computation. Prints the resulting tables to files.

Parameters

Table A.27: pro_aperture function parameters.

Parameter	Description
struct in_cmd* cmd	Pointer to the aperture command, with its parameters

Internal variables

Table A.28: pro_aperture internal variables.

Variable	Description
struct aper_node* limit_node	Pointer to the bottleneck node
struct node *use_range[ ]	Pointer to array of nodes containing start and end node
struct table* aperture_table	Pointer to the aperture table
struct table* tw_cp	Pointer to the Twiss table
char *file	Aperture table output filename
char *range	The range in text, as read from the command
char tw_name[ ]	Array to contain the name of the Twiss table
char *table	Aperture table name
int tw_cnt[ ]	Array with values of row and column in the Twiss table

Return value

No return value.

Remarks

No remarks.

A.3.15 race

Calculates the displacement of the beam centre due to tolerance uncertainties.

```
void race(double xshift, double yshift, double r,
          double angle, double* x, double* y)
```

Description

Receives parameters as shown in Figure 7.5, and the angle for which the distance from outline to centre shall be calculated. Chapter 7.3.1 gives a thorough explanation of the algorithm. Calculations are done for first quadrant only.

Parameters

Table A.29: race function parameters.

Parameter	Description
double r	Radius for the arched part of the racetrack
double xshift, yshift	Displacement of the arched part
double angle	The distance from centre to outline is calculated for this angle
double* x, y	Calculated distances in both planes

Internal variables

Table A.30: race internal variables.

Variable	Description
double angle0	Angle to the first coordinate on the arc
double angle1	Angle to the last coordinate on the arc
double angle2	Angle to the centre of radius
double alfa	See Figure 7.5
double gamma	See Figure 7.5
double theta	See Figure 7.5

Return value

No return value.

Remarks

This function is not used for making a racetrack pipe polygon. The function `make_rectellipse` serves that purpose.

A.3.16 read_twiss_param

Reads twiss parameters from an internal twiss table.

```
void read_twiss_param(char* table int* jslice, double* s,
                      double* x, double* y,
                      double* betx, double* bety,
                      double* dx, double* dy)
```

Description

Receives the name of the table and the row to be read. For the sequence Twiss table, the row is dependant of the range. For the sliced node Twiss table, the row is dependant of which slice to read. Reads then Twiss parameters from the internal table.

Parameters

Table A.31: read_twiss_param function parameters.

Parameter	Description
char* table	Name of the table to read
int* jslice	Number of the slice to be read
double* s	Length along the accelerator, in meters
double* x, y	Distance from reference trajectory, in meters
double* betx, bety	Beta values, in meters
double* dx, dy	Dispersion values, in meters

Internal variables

No internal variables.

Return value

No return value.

Remarks

No remarks.

A.3.17 trim_ws

Converts a string from FORTRAN format to C format.

```
void trim_ws(char* string, int len)
```

Description

The MAD-X source code provides a developer with the function `node_string`, which returns the string value of a given parameter. Since MAD-X needs to be compatible with FORTRAN, string values have a fixed length. A string variable of length 24, holding the word "rectellipse" would have 9 whitespaces at the end. To convert the string to C-format, the function replaces the first whitespace with a "`\0`".

Parameters

Table A.32: trim_ws function parameters.

Parameter	Description
char* string	The string to convert
int len	Length of the string in FORTRAN format

Internal variables

Table A.33: trim_ws internal variables.

Variable	Description
int c	Counter

Return value

No return value.

Remarks

No remarks.

A.3.18 write_aperture_table

Writes aperture and twiss parameters to an internal table.

```
void write_aperture_table(char* name, double* n1, double* rtol,
                          double *xtol, double* ytol,
                          char* apertype,
                          double* ap1, double* ap2,
                          double* ap3,double* ap4,
                          double* on_ap, double* on_elem,
                          double* spec,
                          double* s, double* x, double* y,
                          double* betx, double* bety,
                          double* dx, double* dy)
```

Description

Receives aperture and twiss parameters, and writes them to an internal table. The user chooses in the input script which parameters to print in the output table.

Parameters

Table A.34: write_aperture_table function parameters.

Parameter	Description
char* name	Name of the node
double* n1	n1 aperture
double* rtol, xtol, ytol	Tolerance values
char* apertype	Apertype for the node
double* ap	Pipe parameters, see Chapter 5.2, in meters
double on_ap	Flag indicating whether current node has a valid aperture
double on_elem	Flag indicating whether current node is a drift or not
double spec	Line in plot
double s	Length along the accelerator, in meters
double x, y	Distance from reference trajectory, in meters
double betx, bety	Beta values, in meters
double dx, dy	Dispersion values, in meters

Internal variables

No internal variables.

Return value

No return value.

Remarks

No remarks.

Appendix B

Aperture module user's guide

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH
APERTURE

Computes the n1 values for a piece of machine. Each element is sliced into thick subelements at intervals, and the aperture is computed for each slice. The computation is based on the last Twiss table, so it is important to run the Twiss and aperture commands on the same period or sequence, see the aperture example. Also showed in the example is how N1 values can be plotted.

The minimum n1 for each element is written to the last Twiss table, to allow for matching by aperture.

```
APERTURE,
file=filename,
halofile=filename,
pipefile=filename,
range=range,
exn=real,
eyn=real,
dqf=real,
betaqfx=real,
dp=real,
dparx=real,
dpary=real,
cor=r,
bbeat=real,
nco=integer,
halo={real,real,real,real},
interval=real
spec=real;
```

where the parameters have the following meaning:

file: Output file. Default = aper1.out

halofile: Input file w/halo polygon coordinates. Will suppress an eventual halo parameter. Default = none

pipefile: Input file w/pipe polygon coordinates. Default = none

range: Range given by elements. Default = #s/#e

exn: Normalised horizontal emittance. Default = 3.75e-6

eyn: Normalised vertical emittance. Default = 3.75*e-6

dqf: Peak linear dispersion [m]. Default = 2.086

betaqfx: Beta x in standard qf [m]. Default = 170.25

dp: Bucket edge at the current beam energy. Default = 0.0015

dparx: Fractional horizontal parasitic dispersion. Default = 0.273

dpary: Fractional vertical parasitic dispersion. Default = 0.273

cor: Maximum radial closed orbit uncertainty [m]. Default = 0.004

bbeat: Beta beating coefficient applying to beam size. Default = 1.1

nco: Number of azimuth for radial scan. Default = 5

halo: Halo parameters: {n, r, h, v}. n is the radius of the primary halo, r is the radial part of the secondary halo, h and v is the horizontal and vertical cuts in the secondary halo. Default = {6, 8.4, 7.3, 7.3}

interval: Approximate length in meters between measurements. Actual value: nslice = nodelength/interval, nslice is rounded down to closest integer, interval = nodelength/nslice. Default = 1.0

spec: Aperture spec, for plotting only. Gives the spec line in the plot. Default = 0.0

A more detailed description can be found in "Computation of accelerator aperture and its application to LHC".

iwaarum, November 19, 2004

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH
APERTURE EXAMPLE

The aperture module needs a Twiss table to operate on. It is important not to USE another period or sequence between the Twiss and aperture module calls, else aperture looses its table. One can choose the ranges for Twiss and aperture freely, they need not be the same.

```
use, period=lhcb1;
select, flag=twiss, range=mb.a14r1.b1/mb.a17r1.b1,
        column=keyword, name, parent, k0l, k1l, s, betx, bety, n1;
twiss, file=twiss.b1.data, betx=beta.ip1, bety=beta.ip1,
                           x=+x.ip1, y=+y.ip1, py=+py.ip1;

select, flag=aperture, column=name,n1,x,dy;
aperture, range=mb.b14r1.b1/mb.a17r1.b1, spec=5.235;
plot,table=aperture,noline,vmin=0,0,vmax=30,300,haxis=s,vaxis1=n1,on_elem,vaxis2=betx,bety,co
```

The select command can be used to choose which columns to print in the output file. Column names: name, n1, apertype, rtol, xtol, ytol, ap1, ap2, ap3, ap4, on_ap, on_elem, spec, s, betx, bety, dx, dy, x, y,

where ap# means for all apertypes but racetrack:
ap1 = half width rectangle
ap2 = half heigth rectangle
ap3 = half horizontal axis ellipse (or radius if circle)
ap4 = half vertical axis ellipse

For racetrack, the aperture parameters will have the same meaning as the tolerances:
ap1 and xtol = horizontal displacement of radial part
ap2 and ytol = vertical displacement of radial part (ytol)
ap3 and rtol = radius (rtol)
ap4 = not used

On_elem indicates whether the node is an element or a drift, and on_ap whether it has a valid aperture. The Twiss parameters are the interpolated values used for aperture computation.

When one wants to plot the aperture, the table=aperture parameter is necessary. The normal line of hardware symbols along the top is not compatible with the aperture table, so it is best to include noline. Plot instead the column on_elem along the vaxis to have a simple picture of the hardware. Spec can be used for giving a limit value for n1, to have something to compare with on the plot. This example provides a plot, where we see the n1, beta functions and the hardware symbolized by on_elem.

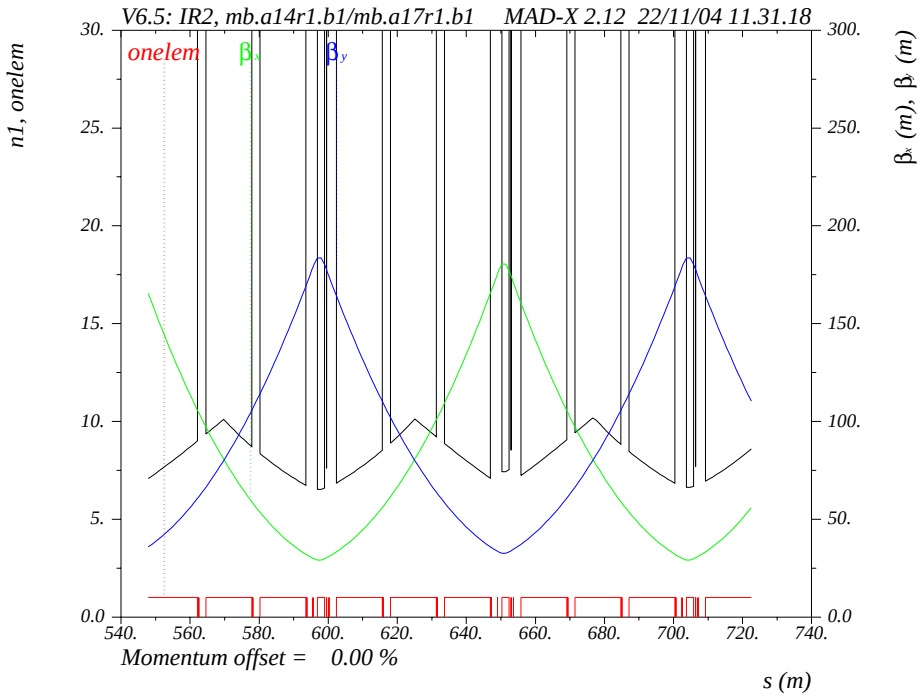


Figure B.1: Plot example.

Appendix C

Sine and cosine proof

Using the built-in `sin()` and `cos()` functions in C is slower than calculating angles from a gradient. In the function `check_if_inside` we have used the latter approach, and the equations are based on general trigonometric deductions.

The sinus is calculated from Equation C.1:

$$\sin\phi = \frac{a_1 b_2 - a_2 b_1}{|\mathbf{a}||\mathbf{b}|} \tag{C.1}$$

To prove that this is a correct assumption we start by looking at Figure C.1, showing two vectors **a** and **b** in the x,y plane.

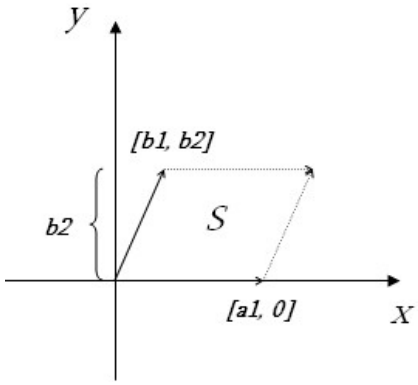


Figure C.1: The area S given by two vectors.

The area S is seen to be

$$S = a_1 b_2 = |\mathbf{a}||\mathbf{b}| \sin\theta \tag{C.2}$$

To have an expression for any surface S as a function of the angle θ between the vectors, we construct a matrix for rotating the vectors around origin.

To rotate vector **a**, each subvector $(a_1, 0)$ and $(0, a_2)$ projected on the x- and y-axis respectively is rotated. $(a_1, 0)$ rotated by θ gives $(a_1\cos\theta, a_1\sin\theta)$. $(0, a_2)$ rotated by θ gives $(-a_2\sin\theta, a_2\cos\theta)$. The complete rotated vector $\mathbf{a}'$ is

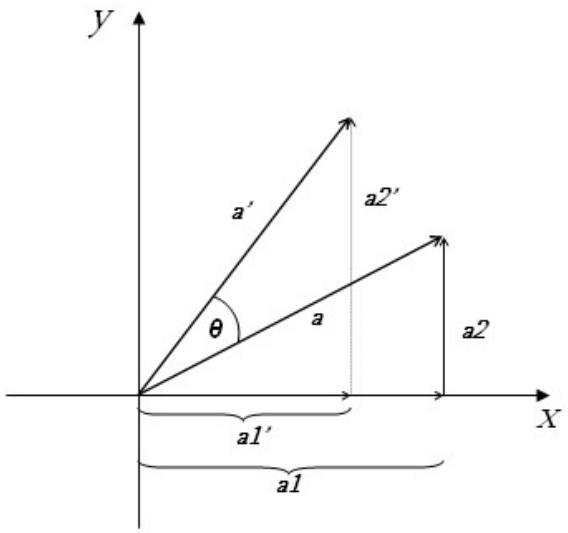


Figure C.2: The vector **a** rotated by θ radians.

thus given by

$$(a_1 \cos\theta - a_2 \sin\theta, a_1 \sin\theta a_2 \cos\theta) \tag{C.3}$$

Writing (C.3) in matrix form gives:

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{C.4}$$

The two by two matrix is called the *rotation matrix*. Multiplying by this matrix now rotates the original vectors **a** and **b** by an angle θ.

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a_1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_1 \cos\theta \\ a_1 \sin\theta \end{pmatrix} = \begin{pmatrix} a'_1 \\ a'_2 \end{pmatrix} \tag{C.5}$$

and for **b** :

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} b_1 \cos\theta - b_2 \sin\theta \\ b_1 \sin\theta + b_2 \cos\theta \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \end{pmatrix} \tag{C.6}$$

We now define: $S' = a'_1 b'_2 - a'_2 b'_1$. Expanding for a' and b' gives

$$S' = a_1 \cos\theta(b_1 \sin\theta + b_2 \cos\theta) - a_1 \sin\theta(b_1 \cos\theta - b_2 \sin\theta)$$

$$= a_1 \cos_1 \sin\theta + a_1 \cos_2 \cos\theta - a_1 \sin_1 \cos\theta + a_1 \sin_2 \sin\theta$$

$$= a_1 b_2 \cos^2\theta + a_1 b_2 \sin^2\theta$$

$$= a_1 b_2$$

So we get $S' = S$. (We might also say that $S' = S$ since $\left|\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\right| = \cos^2\theta + sin^2\theta = 1$, and the vectors do therefore not change their modulus after a rotation).

We now accept that $a_1b_2 - a_2b_1 = S = |\mathbf{a}||\mathbf{b}|\sin\theta$. This is formalized in three dimensions as the cross product, or external product of $\mathbf{a}$ and $\mathbf{b}$:

$$|\mathbf{a} \wedge \mathbf{b}| = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \end{vmatrix} = 0 \cdot \mathbf{i} + 0 \cdot \mathbf{j} + (a_1b_2 - a_2b_1) \cdot \mathbf{k}, \tag{C.7}$$

where $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are unit vectors along the x, y and z axis respectively. Then

$$S = |\mathbf{a} \wedge \mathbf{b}|$$

The cosine expression can be deduced in a similar way:

$$\cos\phi = \frac{a_1b_1 + a_2b_2}{|\mathbf{a}||\mathbf{b}|} \tag{C.8}$$

We study again Figure C.1. We see that $a_1 = |\mathbf{a}|, a_2 = 0, b_1 = |\mathbf{b}|\cos\theta$ and $b_2 = |\mathbf{b}|\sin\theta$. The dot product, or internal product of vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by:

$$\mathbf{ab} = a_1b_1 + a_2b_2 = |\mathbf{a}||\mathbf{b}|\cos\theta \tag{C.9}$$

and in our case, with $a_2 = 0$:

$$\mathbf{ab} = a_1b_1 = |\mathbf{a}||\mathbf{b}| \tag{C.10}$$

We now define two vectors rotated by θ: $\mathbf{a'b'} = a'_1b'_1 + a'_2b'_2$. Expanding with $\mathbf{a'}$ and $\mathbf{b'}$ from equations C.5 and C.6 gives:

$$a_1\cos\theta(b_1\cos\theta - b_2\sin\theta) + a_1\sin\theta(b_1\sin\theta + b_2\cos\theta)$$

$$= a_1\cos_1\cos\theta - a_1\cos_2\sin\theta + a_1\sin_1\sin\theta + a_1\sin_2\cos\theta$$

$$= a_1b_1\cos^2\theta + a_1b_1\sin^2\theta$$

$$= a_1b_1$$

So $\mathbf{ab} = \mathbf{a'b'}$. This result is equal to $|\mathbf{a}||\mathbf{b}|\cos\theta$, so we now accept that $a_1b_1 + a_2b_2 = \mathbf{ab} = |\mathbf{a}||\mathbf{b}|\cos\theta$.

Appendix D

Appendixes on CD-ROM

D.1 Important structures in CONTROL

Included on the project CD: \nodestruct\nodestruct.txt

D.2 File output from MAD example

Included on the project CD: \madexample\psextwiss1.out

D.3 Aperture module source code

Included on the project CD: \aperfuncts*.*

All coding done from scratch, except the function `check_if_inside`, which was based on a FORTRAN module by J. Bernard Jeanneret, and the function `fill_aperture_header`, which is based on the MAD-X routine that fills the Twiss table header.

D.4 Total MAD-X source code

Total MAD-X source code, including the aperture module. Please read copyright.html before use.

Included on the project CD: \MADsource\madX*.*

To run, copy madX directory to harddrive.

Compile command: "make"
Run command: "madx < v65.madx"

Creates files twiss.b1.data, aper1.out and arc.ps.

D.5 Tables from Chapter 9

Included on the project CD: \results*.*

D.6 MAD-X User's Guide

Included on the project CD: \madx_manual\madx_manual.pdf

D.7 Preproject

Included on the project CD: \preproject\preproject.doc

D.8 Electronic version

This report in electronic format.

Included on the project CD: \project\CAA.ps or \project\CAA.pdf