RF-Track Reference Manual *(draft)*

Version 2.0.4

Andrea Latina

AUTHOR AND CONTACT:

Andrea Latina
Beams Department
CERN
CH-1211 GENEVA 23
SWITZERLAND

andrea.latina@cern.ch

Copyright © 2020 by CERN, Geneva, Switzerland

Copyright and any other appropriate legal protection of this computer program and associated documentation reserved in all countries of the world. Organisations collaborating with CERN may receive this program and documentation freely and without charge. CERN undertakes no obligation for the maintenance of this program, nor responsibility for its correctness, and accepts no liability whatsoever resulting from its use. Program and documentation are provided solely for the use of the organisation to which they are distributed. This program may not be copied or otherwise distributed without permission. This message must be retained on this and any other authorised copies. The material cannot be sold. CERN should be given credit in all references.

Started in April 2020

Contents

User Manual

1. Introduction

RF-Track is a new tracking code developed at CERN for the optimization of particle accelerators, which offers outstanding flexibility and rapid simulation speed.

RF-Track can simulate beams of particles with arbitrary energy, mass, and charge, even mixed, solving fully relativistic equations of motion. It can simulate the effects of space-charge forces, both in bunched and continuous-wave beams. It can transport the beams through common elements as well as through “special” ones: 1D, 2D, and 3D static or oscillating radio-frequency electromagnetic field maps (real and complex), flux concentrators, and electron coolers. It allows element overlap, and direct and indirect space-charge calculation using fast parallel algorithms.

RF-Track is written in optimized and parallel C++ and uses the scripting languages Octave and Python as user interfaces. This manual presents its functionalities and the underlying physical models as well as their mathematical and numerical implementation. General knowledge of Octave or Python is recommended to get the best out of RF-Track. For this, we recommend consulting the documentation of these powerful tools.

1.1 Getting started

RF-Track was developed on macOS in C++14, but it runs on any GNU/Linux and other POSIX-compliant systems. This section of the manual will describe how to prepare a suitable environment for compiling and running RF-Track.

RF-Track is a binary module loadable within both the scientific language Octave [1], and Python with its numerical package NumPy [4, 5]. The user interacts with RF-Track through Octave or Pyhton scripts. Two libraries are necessary to compile and run RF-Track: the GNU Scientific Library [3], and the FFTW library [2].

All the mentioned packages are open-source and readily available to most package managers, both for macOS and various Linux distributions. In this section, we will use as examples MacPorts on macOS, and APT on Ubuntu.

1.1.1 Conventions used in this manual

This manual contains numerous examples which can be typed at the keyboard. Some of these examples will refer to shell commands, others to Octave or Python scripts. For practical reasons, all examples of RF-Track simulation scripts will be given in Octave. The conversion from Octave to Python should be straightforward.

Color code

In this manual we use a color code to distinguish between shell, Octave, or Python commands. Shell commands entered at the terminal are shown in light gray:

```
$ command
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign `$` is used as the standard shell prompt, although some systems may use a different character.
A command entered at the Octave command line is shown in a light ocher box:

```
octave:1> command
```

The first string is the standard Octave prompt and should not be typed. It will be, however, omitted in most examples.
A command entered at the Python command line is shown in a light blue box:

```
>>> command
```

The three characters >>> are the standard Python prompt and should not be typed.

In the examples, variables whose name starts with a capital letter will normally refer to vectors or matrices (e.g. `X`, `XP`, `Tmatrix`, ...), whereas variables whose name is in lowercase will refer to scalars (e.g. `mass`, `charge`, `time`, ...).

1.1.2 Preparing the environment

On macOS

On macOS, package managers such as Fink, MacPorts, Spack, or Homebrew can provide a suitable environment for compiling and running RF-Track. We use MacPorts as an example. Macports can be installed following the instructions in `https://www.macports.org/`. If it is already installed on your system, we recommend you update all packages before to install RF-Track:

```
$ sudo port selfupdate
$ sudo port upgrade outdated
```

Then, you must install the packages needed by RF-Track. You can give the following command:

```
$ sudo port install fftw-3 gsl clang
```

If you intend to use RF-Track within Octave, you should also install this package:

```
$ sudo port install octave
```

If you intend to use RF-Track within Python, you should make sure Python is installed on your system, together with its library NumPy:

```
$ sudo port install py-numpy
```

On Linux

We give the example Ubuntu, as it is a widely used Linux distribution. We assume the use of APT as package manager.
It is good practice to update all packages before you install RF-Track:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Then, you must install the packages needed by RF-Track. You can give the following command:

```
$ sudo apt install libgsl-dev fftw-dev
```

If you intend to use RF-Track within Octave, you should also install this package:

```
$ sudo apt install liboctave-dev
```

If you intend to use RF-Track within Python, you should make sure Python is installed on your system, together with its library NumPy:

```
$ sudo apt install libpython3-dev python3-dev python3-numpy
```

1.1.3 Obtaining RF-Track

The source code for RF-Track can be downloaded from the RF-Track Gitlab repository:
`https://gitlab.cern.ch/alatina/rf-track-2.0/`
or using the command:

```
$ git clone https://gitlab.cern.ch/alatina/rf-track-2.0.git
```

1.1.4 Compiling RF-Track

Before you compile RF-Track you need to prepare a suitable environment and install the necessary packages. This can be done using the standard command `./configure` from the directory `rf-track-2.0`:

```
$ cd rf-track-2.0
```

If you want to use RF-Track within Octave, you must give the following command:

```
$ ./configure --enable-octave
```

If you want to use RF-Track within Python, you must give command:

```
$ ./configure --enable-python
```

You can use both options simultaneously, if you intend to use RF-Track with both Python and Octave. You can give the command `./configure --help` for more options.
Now you can compile RF-Track:

```
$ make -jN
```

where `N` is the number of simultaneous threads your system can afford to use during compilation.

Using RF-Track

There is no need for RF-Track to be installed using commands such as `make install`. It is enough that your Octave or Python scripts point to the directory `rf-track-2.0`, where RF-Track was compiled and its binary files are located. For using RF-Track with Octave, you should start your Octave scripts with the command:

```
octave:1> addpath('/Users/alatina/Codes/rf-track-2.0');
```

(you can just add this command to the file `.octaverc` in your home directory, to run it automatically whenever Octave is launched).

In Python, you should give the commands:

```
>>> import os,sys
>>> BIN = os.path.expanduser("/Users/alatina/Codes/rf-track-2.0")
>>> sys.path.append(BIN)
```

1.1.5 Loading RF-Track

To load RF-Track in Octave, it is sufficient to give the command:

```
octave:1> RF_Track;
```

To load RF-Track in Python, it is sufficient to to give the command:

```
>>> import RF_Track
```

1.2 Using RF-Track

1.2.1 An example program

The following Octave script demonstrates how RF-Track works. In the example, a FODO cell is created, and a beam is tracked through it to plot the Twiss parameters and the final phase space. The FODO lattice is designed to have 90 degrees phase advance, and the beam is a bunch of electrons matched to the cell. The bunch is then tracked through the cell and the Twiss parameters, β_x and β_y, are plotted. The lines are numbered, and each block is commented on below.

```
%% Load RF-Track
RF_Track;

%% Beam parameters
mass = RF_Track.electronmass; % particle mass in MeV/c^2
population = 1e10; % number of particles per bunch
Q = -1; % particle charge in units of e
Pref = 5; % reference momentum in MeV/c
B_rho = Pref / Q; % beam magnetic rigidity in MV/c

%% FODO cell parameters
Lcell = 2; % cell length in m
Lquad = 0.0; % m, zero length quadrupole (we use a thin quadrupole)
Ldrift = Lcell/2 - Lquad; % drift space between the two quadrupoles
```

```
mu = 90; % phase advance per cell in deg
k1L = sind(mu/2) / (Lcell/4); % 1/m, quadrupole focusing strength
strength = k1L * B_rho; % MeV/m, quadrupole strength

%% Create the elements
Qf = Quadrupole(Lquad/2, strength/2); % half focusing thin quadrupole
Qd = Quadrupole(Lquad, -strength); % defocusing thin quadrupole
Dr = Drift(Ldrift); % drift space
Dr.set_tt_nsteps(100); % number of steps for the transport table

%% Create the lattice
FODO = Lattice(); % Create a new object Lattice() called FODO
FODO.append(Qf); % 1/2 F
FODO.append(Dr); % O
FODO.append(Qd); % D
FODO.append(Dr); % O
FODO.append(Qf); % 1/2 F

%% Define Twiss parameters
Twiss = Bunch6d_twiss();
Twiss.beta_x = Lcell * (1 + sind(mu/2)) / sind(mu); % m
Twiss.beta_y = Lcell * (1 - sind(mu/2)) / sind(mu); % m
Twiss.alpha_x = 0.0;
Twiss.alpha_y = 0.0;
Twiss.emitt_x = 1; % mm.mrad, normalized emittances
Twiss.emitt_y = 1; % mm.mrad

%% Create the bunch
B0 = Bunch6d(mass, population, Q, Pref, Twiss, 10000);

%% Perform tracking
B1 = FODO.track(B0);

%% Retrieve the Twiss plot and the phase space
T = FODO.get_transport_table('%S %beta_x %beta_y');
M = B1.get_phase_space('%x %xp %y %yp');

%% Make plots
figure(1)
hold on
plot(T(:,1), T(:,2), 'b-')
plot(T(:,1), T(:,3), 'r-')
legend({ '\beta_x ', '\beta_y ' })
xlabel('S [m]')
ylabel('\beta [m]')

```

```
61 figure(2)
62 scatter(M(:,1), M(:,2), '*')
63 xlabel('x [mm]')
64 ylabel('x'' [mrad]')
```

Line 2 stars the games, loading RF-Track into Octave. RF-Track displays a welcome message that includes some useful information: the version number, the libraries RF-Track was compiled with, the contact information, and the copyright notice. This call has the function to make available to Octave the entire set of RF-Track commands, as well as its predefined constants (see the following section for details).

Lines 5-17 define some useful Octave constants related to our problem. Notice, in line 5, the use of the RF-Track's constant `electronmass`, which contains the mass of the electron, here expressed in MeV/c^2.

Lines 20-23 define the elements of our FODO cell: the focusing quadrupole, the defocusing quadrupole, and the drift space between them. The quadrupoles are thin, and for practical reasons, the focusing one is divided into two, so that our FODO cell starts with half a focusing quadrupole and ends with the other half. This block of lines shows the first two RF-Track commands we encountered: `Quadrupole()` and `Drift()`. These are two objects that create a quadrupole magnet and a drift space. Line 23 uses a *method* of the object Drift, which specifies that the drift must be divided into 100 steps. Dividing into steps, here, is just for nicely tracking the Twiss parameters[1].

Lines 26-31 define the FODO lattice itself. An object of type `Lattice()` is created, with name FODO. At creation, a `Lattice()` is an empty sequence. Lines 27 through 31 append to `FODO` the elements that compose the cell. It is important to understand that RF-Track, internally, stores in `FODO` a *copy* of these elements, not the element themselves. Therefore, modifying `Qf`, `Qd`, or `Dr` after the appending them will not affect `FODO`.

Lines 34-44 define the Twiss parameters and create the bunch `B0`, which is an instance of the object `Bunch6d()`. The Twiss parameters are assigned to `Twiss`, an instance of `Bunch6d_twiss()`. `B0` is created with 10'000 macro-particles.

Lines 47-51 track the bunch `B0` through the FODO cell, and store the out-coming bunch in `B1`, another object of type `Bunch6d()`. Lines 50 and 51 retrieve the Twiss parameters and the phase space and store them in two Octave variables `T` and `M`.

Lines 53-65 plot the results using the standard Octave plotting commands.

Figure 1.1 shows the output of the example: the Twiss parameters, β_x and β_y, and the phase space plot.

1.2.2 Physical units

Internally, RF-Track stores each physical quantity in the most suitable units for numerical computation in accelerator physics. Table 1.1 shows the units grouped by conceptual category. Regarding positions and angles, notice that all beam-related quantities are in units of millimeters or milliradians, whereas all machine-related quantities are expressed in meters and radians.

1.2.3 Predefined constants

A number of constants are predefined within RF-Track, for the user's convenience:

[1]The two letters "`tt`" mean that the specified number of steps refers to the so-called *tracking table*, a table created during tracking to follow the evolution of various beam quantities along the lattice, including the Twiss parameters.

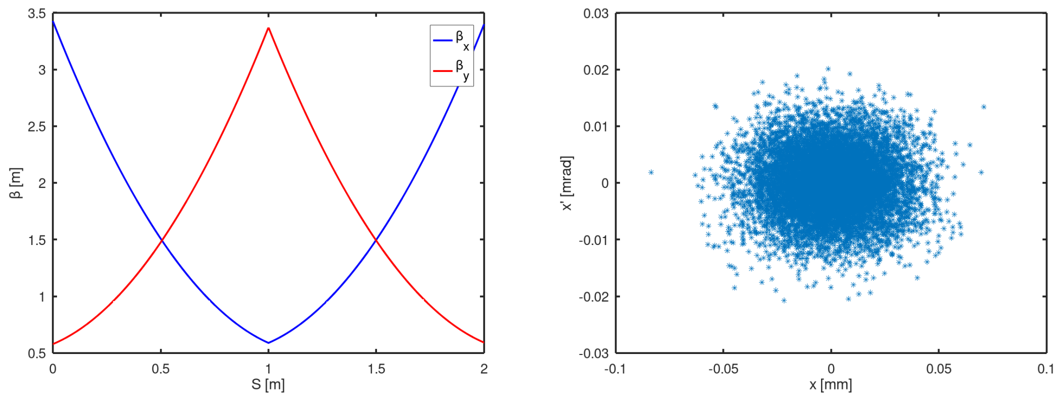


Figure 1.1: The output of the example.

Table 1.1: RF-Track physical units.

Quantity	**Symbols**	**Unit**
Bunch population	N	number of particles
Particle mass	m	MeV/c^2
Particle charge	Q	e
Particle positions	x, y, z	mm
Particle angles	x', y'	mrad
Particle momenta	P_x, P_y, P_z, P	MeV/c
Particle energy	E	MeV
Time	t	mm/c
Element offsets and positions	X_o, Y_o, Z_o	m
Element pitch	X'_o	rad
Element yaw	Y'_o	rad
Element roll	Z'_o	rad

```
clight % speed of light, in m/s
muonmass % muon mass in MeV/c^2
protonmass % proton mass, in MeV/c^2
electronmass % electron mass, in MeV/c^2
s, ms, us, ns, ps, fs; % various units if time, in mm/c
C, mC, uC, nC, pC % various units of charge, in e
```

For example if one needs to define a time interval of 5 ps, say `dt`, one can simply write:

```
octave:1> dt = 5 * RF_Track.ps
dt = 1.4990 % 5 ps in mm/c
```

1.2.4 Run-time parameters and options

Once RF-Track is loaded, a few variables become available to the user to customize the way RF-Track operates.

Threads

```
number_of_threads % the number of threads that RF-Track must use
max_number_of_threads % the maximum number of threads available [read-only]
```

RF-Track is a parallel application. By default RF-Track uses the maximum number of threads on your machine, `max_number_of_threads`, however the user can change this number (normally to reduce it) by setting the variable `number_of_threads`.

Random Number Generator

```
rng_set(name) % change the random number generator (RNG)
rng_set_seed(seed) % set the seed for the RNG
rng_get() % return what is the current RNG
```

Random numbers are used in a multitude of contexts in scientific computation. RF-Track uses high-quality random number generators. Using these three options the user can set the desired random number generator and its initial seed. The full list of the available random number generators is available in Appendix, Tab. A.2.

Version number

```
version % the RF-Track version number [read-only]
```

Returns the RF-Track version number.

1.3 Further information

1.3.1 Citing RF-Track in publications

We have invested a lot of time and effort in creating RF-Track, please cite it when using it.
To cite RF-Track in publications use:

```
Andrea Latina
"RF-Track Reference Manual", CERN, Geneva, Switzerland, June 2020
DOI: 10.5281/zenodo.3887085
```

A BibTeX entry for LaTeX users is:

```
@techreport{,
  address = {Geneva, Switzerland},
  author = {Latina, Andrea},
  doi = {10.5281/zenodo.3887085},
  institution = {CERN},
  title = {RF-Track Reference Manual},
  year = {2020}
}
```

1.3.2 Acknowledgments

The author is very thankful to all those who contributed to RF-Track, especially those who were brave enough to use the code before this manual existed. A special thank goes to Roberto Corsini of CERN, for his encouragement and support, and to Costanza Agazzi, Stefano Benedetti, Mohsen Dayyani Kelisani, and Yongke Zhao for their essential contributions, which provided RF-Track with many of its unique features.

2. Beams

RF-Track implements two different particle tracking methods: tracking *in time* and tracking *in space*. The tracking *in time* is preferable in space-charge-dominated regimes, where the relative positions of the particles in space matters. The tracking *in space* suits better space-charge-free regions, where particles are independent of each other and they can be transported simultaneously from the entrance plane of an element to its end, element by element.

RF-Track provides two distinct beam types to implement these two models: `Bunch6dT` for tracking in time, and `Bunch6d` for tracking in space. A dedicated tracking environment exists for both of these two beam types: `Lattice` for `Bunch6d`, and `Volume` `Bunch6dT`. This chapter will describe them in detail.

2.1 Beam models

2.1.1 Bunch6d

When `Bunch6d` is used, tracking is performed using the accelerator longitudinal coordinate, S, as the integration variable. This corresponds to what one studies in accelerator physics textbooks, and it is the base for matrix-based beam optics. In this model, all particles belong to the same plane at a given longitudinal coordinate, S, and they are then transported to the element end using the arrival time as the longitudinal coordinate. In `Bunch6d`, the beam is represented by a set of macro-particles whose state vector is an extended trace space:

$$\left(x, \quad x', \quad y, \quad y', \quad t, \quad P, \quad m, \quad Q, \quad N\right)$$

The meaning of each symbol follows with the units used internally to store the information:

x, y	transverse coordinates	[mm]
x', y'	transverse angles	[mrad]
t	arrival time at S	[mm/c]
P	total momentum	[MeV/c]
m	mass	[MeV/c^2]
Q	charge of the single particle	[e]
N	number of single particles in each macro-particle	[#]

`Bunch6d` also stores the longitudinal coordinate S of the bunch.

Constructors

A new `Bunch6d` can be created at least in two ways: from a set of Twiss parameters, or directly from a beam matrix with the phase space. Multi-specie bunches can be created using a beam matrix of the extended phase space. Follows the list of constructors:

```
B = Bunch6d(mass, population, charge, Pref, Twiss, nParticles, sigmaCut=0 );
B = Bunch6d(mass, population, charge, [ X XP Y YP T P ] );
B = Bunch6d( [ X XP Y YP T P MASS Q N ] );
B = Bunch6d( [ X XP Y YP T P MASS Q ] );
B = Bunch6d();
```

The possible arguments are:

`mass`	the mass of the single particle	[MeV/c^2]
`population`	the total number of real particles in the bunch	[#]
`charge`	charge of the single particle	[e]
`Pref`	reference momentum	[MeV/c]
`Twiss`	instance of object `Bunch6d_twiss` (see section 2.2.1)	
`nParticles`	number of macro-particles in bunch	[#]
`sigmaCut`	if > 0 cuts the distributions at `sigma_cut` sigmas	[#]
`X`	column vectors of the horizontal coordinates	[mm]
`Y`	column vectors of the vertical coordinates	[mm]
`T`	column vector of the arrival times	[mm/c]
`P`	column vector of the total momenta	[MeV/c]
`XP`	column vectors of the x' angles	[mrad]
`YP`	column vectors of the y' angles	[mrad]
`MASS`	column vector of masses	[MeV/c^2]
`Q`	column vector of single-particle charges	[e]
`N`	column vector of numbers of single particles per macro particle	[#]

A default constructor, with no arguments, allows creating an empty beam. The return value is an object of type `Bunch6d`.

Reading the phase space

The particles coordinates can be inquired using the method `get_phase_space()`. This method allows one to retrieve the bunch information in multiple ways.

`%x`	horizontal coordinate	[mm]
`%y`	vertical coordinate	[mm]
`%t`	arrival time, t	[mm/c]
`%dt`	relative arrival time, $t - t_0$	[mm/c]
`%z`	longitudinal coordinate w.r.t. the reference particle	[mm]
`%deg@MHz`	longitudinal coordinate in degrees at specified frequency in MHz, e.g. `%deg@750` for degrees at 750 MHz	[deg]
`%K`	kinetic energy	[MeV]
`%E`	total energy	[MeV]
`%P`	total momentum	[MeV/c]
`%d`	relative momentum, $\delta = (P - P_0)/P_0$	[permille]
`%xp`	horizontal angle, P_x/P_z	[mrad]
`%yp`	vertical angle, P_y/P_z	[mrad]
`%Px`	horizontal momentum, P_x	[MeV/c]
`%Py`	vertical momentum, P_y	[MeV/c]
`%Pz`	longitudinal momentum, P_z	[MeV/c]
`%px`	normalized horizontal momentum, P_x/P_0	[mrad]
`%py`	normalized vertical momentum, P_y/P_0	[mrad]
`%pz`	normalized longitudinal momentum, P_z/P_0	[mrad]
`%pt`	normalized relative energy difference, $p_t = (E - E_0)/P_0c$	[permille]
`%Vx`	horizontal velocity	[c]
`%Vy`	vertical velocity	[c]
`%Vz`	longitudinal velocity	[c]
`%m`	mass	[MeV/c^2]
`%Q`	charge	[e^+]
`%N`	number of particles per macro particle	[#]

Table 2.1: List of identifiers accepted by `Bunch6d::get_phase_space()`.

```
M = B.get_phase_space(format_fmt='%x %xp %y %yp %t %Pc', which='good');
```

The two arguments are:

`format_fmt`	this parameter allows the user to specify what phase space representation should be returned
`which`	this parameter specified whether all particles should be returned, including the lost ones, of just the `'good'` ones

The default values correspond to the internal representation of the beam, for all the "good" particles. Table 2.1 shows all possible identifiers. The return value `M` contains the requested phase space.

Modifying the phase space

The particles coordinates can be inquired using the method `get_phase_space()`. This method allows one to retrieve the bunch information in multiple ways.

```
B.set_phase_space( [ X XP Y YP T P ] );
```

The only argument is a 6-column matrix containing the phase space.

2.1.2 Bunch6dT

Tracking “in time” uses directly the time, t, as the integration variable. When `Bunch6dT` is used, the equations of motion are integrated in time. The 6d phase-space coordinates of each particle are $(x,\ y,\ S,\ P_x,\ P_y,\ P_z)$. `Bunch6dT` maintains the clock common to all particles, t, which each integration step updates. In `Bunch6dT`, the beam is represented by a set of macro-particles whose state vector is an extended phase space:

$$\left(X,\quad P_x,\quad Y,\quad P_y,\quad S,\quad P_z,\quad m,\quad Q,\quad N,\quad t_0\right)$$

In this model, all particles belong to the same plane at a given longitudinal coordinate, S, typically located at the entrance of an element. They are then transported to the element end using the arrival time as the longitudinal coordinate. The meaning of each symbol follows, with the units used to store the information internally:

X, Y, S	transverse and longitudinal coordinates	[mm]
P_x, P_y, P_z	transverse and longitudinal momenta	[MeV/c]
m	mass	[MeV/c^2]
Q	charge of the single particle	[e]
N	number of single particles in each macro-particle	[#]
t_0	creation time	[mm/c]

An important difference with respect to `Bunch6d` is the presence of t_0, the creation time of each particle. This enables, for instance, the simulation of cathodes and field emission. `Bunch6dT` also stored the time t at which the bunch is taken.

Constructors

A new `Bunch6dT` can be created at least in two ways: from a set of Twiss parameters, or directly from a beam matrix with the phase space. Multi-specie bunches can be created using a beam matrix of the extended phase space. Follows the list of constructors:

```
B = Bunch6dT(mass, population, charge, Pref, Twiss, nParticles, sigma_cut=0 );
B = Bunch6dT(mass, population, charge, [ X Px Y Py S Pz ] );
B = Bunch6dT( [ X Px Y Py S Pz MASS Q N T0 ] );
B = Bunch6dT( [ X Px Y Py S Pz MASS Q N ] );
B = Bunch6dT( [ X Px Y Py S Pz MASS Q ] );
B = Bunch6dT();
```

The possible arguments are:

`mass`	the mass of the single particle	[MeV/c^2]
`population`	the total number of real particles in the bunch	[#]
`charge`	charge of the single particle	[e]
`Pref`	reference momentum	[MeV/c]
`Twiss`	instance of object `Bunch6d_twiss` (see section 2.2.1)	
`nParticles`	number of macro-particles in bunch	[#]
`sigmaCut`	if > 0 cuts the distributions at `sigma_cut` sigmas	[#]
`X`	column vectors of the horizontal coordinates	[mm]
`Y`	column vectors of the vertical coordinates	[mm]
`S`	column vectors of the longitudinal coordinates	[mm]
`Px`	column vectors of the horizontal momenta	[MeV/c]
`Py`	column vectors of the vertical momenta	[MeV/c]
`Pz`	column vectors of the longitudinal momenta	[MeV/c]
`MASS`	column vector of masses	[MeV/c^2]
`Q`	column vector of single-particle charges	[e]
`N`	column vector of numbers of single particles per macro particle	[#]
`T0`	column vector of creation times	[mm/c]

A default constructor allows creating an empty beam. The return value is an object of type `Bunch6dT`.

Reading the phase space

The particles coordinates can be inquired using the method `get_phase_space()`. This method allows one to retrieve the bunch information in multiple ways.

```
M = B.get_phase_space(format_fmt='%X %Px %Y %Py %S %Pz', which='good');
```

The two arguments are:

`format_fmt`	this parameter allows the user to specify what phase space representation should be returned
`which`	this parameter specified whether all particles should be returned, including the lost ones, of just the `'good'` ones

The default values correspond to the internal representation of the beam, for all the "good" particles. Table 2.2 shows all possible identifiers. The return value `M` contains the requested phase space.

Modifying the phase space

The particles coordinates can be inquired using the method `get_phase_space()`. This method allows one to retrieve the bunch information in multiple ways.

```
B.set_phase_space( [ X PX Y PY S PZ ] );
```

The only argument is a 6-column matrix containing the phase space.

`%X`	horizontal coordinate	[mm]
`%Y`	vertical coordinate	[mm]
`%S`	longitudinal coordinate	[mm]
`%Z`	longitudinal coordinate w.r.t. the reference particle	[mm]
`%deg@MHz`	longitudinal coordinate in degrees at specified frequency in MHz, e.g. `%deg@750` for degrees at 750 MHz	[deg]
`%K`	kinetic energy	[MeV]
`%E`	total energy	[MeV]
`%P`	total momentum	[MeV/c]
`%d`	relative momentum, $\delta = (P - P_0)/P_0$	[permille]
`%t0`	creation time	[mm/c]
`%xp`	horizontal angle, P_x/P_z	[mrad]
`%yp`	vertical angle, P_y/P_z	[mrad]
`%Px`	horizontal momentum, P_x	[MeV/c]
`%Py`	vertical momentum, P_y	[MeV/c]
`%Pz`	longitudinal momentum, P_z	[MeV/c]
`%px`	normalized horizontal momentum, P_x/P_0	[mrad]
`%py`	normalized vertical momentum, P_y/P_0	[mrad]
`%pz`	normalized longitudinal momentum, P_z/P_0	[mrad]
`%pt`	normalized relative energy difference, $p_t = (E - E_0)/P_0c$	[permille]
`%Vx`	horizontal velocity	[c]
`%Vy`	vertical velocity	[c]
`%Vz`	longitudinal velocity	[c]
`%m`	mass	[MeV/c^2]
`%Q`	charge	[e^+]
`%N`	number of particles per macro particle	[#]

Table 2.2: List of identifiers accepted by `Bunch6dT::get_phase_space()`.

2.1.3 Conversion

From Bunch6d to Bunch6dT

An object of type `Bunch6d` can be converted into an object of type `Bunch6dT` by calling the dedicated constructor:

```
B0T = Bunch6dT(B0);
```

Where `B0` is an object of type `Bunch6d` and `B0T` an object of type `Bunch6dT`. When `Bunch6d` is converted into `Bunch6dT` all particles are set to the same longitudinal coordinate, `B0.S`, and the original longitudinal distribution, which `B0` carries as distribution of arrival times, is transferred to `Bunch6dT` as distribution of creation times.

From Bunch6dT to Bunch6d

An object of type `Bunch6dT` can be converted into an object of type `Bunch6d` by calling the dedicated constructor:

```
B0 = Bunch6d(B0T);
```

Where B0T is an object of type Bunch6dT and B0 an object of type Bunch6d. When Bunch6dT is converted into Bunch6d all particles are projected to an average longitudinal plane. The projection follows a straight line along each particle's trajectory, and the time spent by each particle to reach such plane determines the initial arrival time.

2.1.4 Coasting beams

Both types Bunch6d and Bunch6dT can represent coasting beams. The method set_coasting() allows the user to specify that the beam is coasting:

```
B.set_coasting(L);
```

The only argument of this method is:

L	the period length	[mm]

If a beam is coasting each particle is followed and preceded by a virtually infinite number of particles, separated by periods of length L. This property of the beam affects the space charge calculation and has no other effect.

2.2 Twiss parameters

2.2.1 Bunch6d_twiss

The structure Bunch6d_twiss gathers all the information to generate a beam from the Twiss parameters. This structure contains the following items:

```
T = Bunch6d_twiss();
T.emitt_x; % mm.mrad, normalised horizontal emittance
T.emitt_y; % mm.mrad, normalised vertical emittance
T.emitt_z; % mm.permille, normalised longitudinal emittance
T.alpha_x;
T.alpha_y;
T.alpha_z;
T.beta_x; % m, horizontal beta function
T.beta_y; % m, vertical beta function
T.beta_z; % m, longitudinal beta function
T.disp_x; % m, horizontal dispersion
T.disp_xp; % rad, horizontal dispersion prime
T.disp_y; % m, vertical dispersion
T.disp_yp; % rad, vertical dispersion prime
T.disp_z; % m, longitudinal dispersion
```

2.2.2 Bunch6d_info

The user can inquire the Twiss parameters of a bunch, by calling the method:

```
I = B.get_info();
```

Two structures exist, one for Bunch6d and one for Bunch6dT respectively. If B is a Bunch6d:

```
I = B.get_info();
I.S; % m
I.mean_x; % mm, average H position
I.mean_y; % mm, average V position
I.mean_t; % mm/c, average arrival time
I.mean_xp; % mrad, average H angle
I.mean_yp; % mrad, average V angle
I.mean_P; % average momentum in MeV/c
I.mean_K; % average kinetic energy in MeV
I.mean_E; % average total energy in MeV
I.sigma_x; % mm
I.sigma_y; % mm
I.sigma_z; % mm
I.sigma_t; % mm/c
I.sigma_xp; % mrad
I.sigma_yp; % mrad
I.sigma_xxp; % mm*mrad
I.sigma_yyp; % mm*mrad
I.sigma_zP; % mm*MeV
I.sigma_E; % kinetic-energy spread in MeV
I.sigma_P; % momentum spread in MeV
I.emitt_x; % mm.mrad normalised emittance
I.emitt_y; % mm.mrad normalised emittance
I.emitt_z; % mm.keV that is, emitt_z / mm.keV = sigmaz / mm * sigma_P / keV
I.emitt_4d; % mm.mrad, 4d normalised emittance
I.alpha_x;
I.alpha_y;
I.alpha_z;
I.beta_x; % m
I.beta_y; % m
I.beta_z; % m
I.rmax; % mm, largest particle's xy distance from the origin
I.transmission; % percent
```

If B is a Bunch6dT:

```
I = B.get_info();
I.t; % mm/c
I.mean_X; % mm, average H position
I.mean_Y; % mm, average V position
```

```
I.mean_S; % mm, average L position
I.mean_Px; % MeV/c, average H momentum
I.mean_Py; % MeV/c, average V momentum
I.mean_Pz; % MeV/c, average L momentum
I.mean_K; % average kinetic energy in MeV
I.mean_E; % average total energy in MeV
I.sigma_X; % mm
I.sigma_Y; % mm
I.sigma_Z; % mm
I.sigma_Px; % MeV/c
I.sigma_Py; % MeV/c
I.sigma_Pz; % MeV/c
I.sigma_XPx; % mm*MeV/c
I.sigma_YPy; % mm*MeV/c
I.sigma_ZPz; % mm*MeV/c
I.sigma_E; % kinetic-energy spread in MeV
I.emitt_x; % mm.mrad normalised emittance
I.emitt_y; % mm.mrad normalised emittance
I.emitt_z; % mm.keV that is, emitt_z / mm.keV = sigmaz / mm * sigma_P / keV
I.emitt_4d; % mm.mrad, 4d normalised emittance
I.alpha_x;
I.alpha_y;
I.alpha_z;
I.beta_x; % m
I.beta_y; % m
I.beta_z; % m
I.rmax; % mm, largest particle's xy distance from the origin
I.transmission; % percent
```

2.3 Persistency

Both Bunch6d and Bunch6dT can be loaded and saved on disk, and exported, in multiple ways.

2.3.1 Saving and loading a beam

Two methods can be used to save and load the beam:

```
B.save(filename);
B.load(filename);
```

The beam is saved as a raw binary file. This guarantees that a beam loaded from disk is identical to the original. Notice that binary format is machine-dependent. It might happen that a file saved on one architecture cannot be read by another architecture, because their internal representation is different.

2.3.2 Exporting as DST file

One can save the beam in DST binary format:

```
B.save_as_dst_file(filename, frequency_in_MHz);
```

The RF frequency (expressed in MHz) is required.

2.3.3 Exporting as SDDS file

One can save the beam in binary format in SDDS files:

```
B.save_as_sdds_file(filename, description);
```

The string `description` is optional.

There exist also alternatives ways. For example, one can extract the phase space using `get_phase_space()` and then save the matrix using dedicated Octave or Python commands.

2.4 Example

For an example of a bunch created from the Twiss parameters, see Chapter 1. The following example creates a bunch from a matrix. The matrix has dimensions $N \times 6$, where 6 is the size of the phase space, and N is the number of macroparticles in the bunch.

```
% creates a bunch of 1e12 electrons, using 1000 macroparticles
O = zeros(1000,1); % a column vector of zeros
X = randn(1000,1); % mm, column vector of normally distributed positions
Y = randn(1000,1); % mm, column vector of normally distributed positions
P = 100*ones(1000,1); % MeV/c, column vector of particles momenta
M = [ X O Y O O P ]; % the beam matrix, %x %xp %y %yp %t %P

% create a bunch
B0 = Bunch6d(RF_Track.electronmass, 1e12, -1, M);

% retrieve the phase space following MAD-X convention
T0 = B0.get_phase_space("%x %px %y %py %Z %pt");

% retrieve the phase space following the TRANSPORT convention
T0 = B0.get_phase_space("%x %xp %y %yp %dt %d");

% retrieve the phase space following PLACET convention
T0 = B0.get_phase_space("%E %x %y %dt %xp %yp");

% save on disk
B0.save('my_bunch.dat'); % RF-Track binary format
B0.save_as_dst_file('my_bunch.dst', 750.0); % save as DST
B0.save_as_sdds_file('my_bunch.sdds', 'this is my bunch'); % save as SDDS
```

■

3. Beam lines

3.1 Environments

RF-Track offers two different environments to track particles: the `Lattice` and the `Volume`. The environment `Lattice` represents the accelerator as a plain list of elements. `Lattice` works with `Bunch6d` and transports the beam, element by element, from the entrance to the exit plane of each element sequentially.

The environment `Volume` provides more flexibility than `Lattice`: it can simulate elements with arbitrary position and orientation in the three-dimensional space, and allows elements to overlap. `Volume` works with `Bunch6dT`, which is more suitable for space-charge calculations. The possibility for `Bunch6dT` to handle particle creation at any time and location, also allows the simulation of cathodes, field emission, and dark currents. In a `Volume`, particles can propagate in any direction (even backwards). So that particles of the same bunch can happen to occupy different elements simultaneously.

Several "special" elements, unavailable to `Lattice`, allow taking full advantage of the flexibility of `Volume`: e.g., analytic coils and solenoids, where the magnetic field is computed from analytic formulæand permeates the whole 3D space, allowing realistic fringe fields. The possibility to overlap elements make `Volume` perfect for the simulation of injectors, where a solenoidal magnetic field usually surrounds the accelerating field of the gun, and the effects of space-charge are critical.

Notice that also the environment `Lattice` can take into account space-charge effects. However, since `Bunch6d` maintains the distribution of the particles on the same longitudinal plane, the calculation of space-charge forces needs an on-the-fly extrapolation of each particle's longitudinal position, using the arrival time and the velocity, to reconstruct the three-dimensional distribution. Because of this somehow artificial manipulation of the phase space, we judge `Lattice` as most suitable for space-charge-free regions of the accelerator. Whereas `Volume`, which through `Bunch6dT` maintains the full spatial distribution of the beam, is the preferred choice for space-charge-dominated regimes.

`Lattice` is straightforward and fast, but better limited to space-charge-free regions. `Volume` is extremely flexible and handles space-charge without extrapolations, but this additional flexibility comes at the cost of being computationally more expensive. This chapter describes these two environments, as well as all the elements available to the user.

3.1.1 The Lattice

Constructor

In order to setup a lattice, it is sufficient to create an object of type `Lattice`:

```
L = Lattice();
```

There are no input options.

Adding an element

To append elements to a lattice, you can use the method `append`, Which accepts elements as well as lattices:

```
L.append(element);
L.append(lattice);
```

Notice that this method appends *a copy* of the new element, at the end of the lattice. In C++ jargon, one would say that the elements are passed by value, not by reference.

Tracking the beam

To track a beam, e.g., `B0` through a lattice, one can call the method `track`:

```
B1 = L.track(B0);
```

where the input argument is the beam to be tracked, and the return value is the beam at the exit of the lattice.

3.1.2 The Volume

Constructor

In order to setup a volume, it is sufficient to create an object of type `Volume`:

```
V = Volume();
```

There are no input options.

Adding an element

To place elements into volume, you can use the method `add`, which comes in many flavors:

```
V.add(element, Xpos, Ypos, Zpos, reference='entrance');
V.add(element, Xpos, Ypos, Zpos, roll, pitch, yaw, reference='entrance');
V.add(lattice, Xpos, Ypos, Zpos, reference='entrance');
V.add(lattice, Xpos, Ypos, Zpos, roll, pitch, yaw, reference='entrance');
```

```
V.add(volume, Xpos, Ypos, Zpos, reference='entrance');
V.add(volume, Xpos, Ypos, Zpos, roll, pitch, yaw, reference='entrance');
```

The possible arguments are:

`element`	the element to be added	
`lattice`	the lattice to be added	
`volume`	the volume to be added	
`Xpos`	the horizontal position of the new element	[m]
`Ypos`	the vertical position of the new element	[m]
`Zpos`	the longitudinal position of the new element	[m]
`roll`	the rotation angle around the Z axis	[rad]
`pitch`	the rotation angle around the X axis	[rad]
`yaw`	the rotation angle around the Y axis	[rad]
`reference`	reference point for position and angles: it can be either `'entrance'`, `'center'`, or `'exit'`	

The angles follow the Tait–Bryan formalism. Like in `Lattice`, this method adds *a copy* of the new element to the `Volume`.

Tracking the beam

To track a beam, e.g., `B0` through a lattice, one can call the method `track`:

```
B1 = V.track(B0, options);
```

The possible arguments are:

`B0`	the beam to be tracked
`options`	an instance of the object `TrackingOptions`. See next chapter for the details.

The return value is `B1`, that is, the beam right after *all particles* leave the volume.

3.2 Elements

RF-Track provides with a large number of elements.

3.2.1 Drift

```
D = Drift(L=0); % L [m]
```

The only parameter, `L`, is the length of the drift space in meters.

3.2.2 Quadrupole

```
Q = Quadrupole(L, strength); % L [m] strength [MV/c/m]
Q = Quadrupole(L, P_Q, k1); % L [m] P_Q [MV/c] k1 [1/m^2]
```

The possible arguments are:

`L`	the quadrupole length	[m]
`strength`	the integrated focusing strength, S	[MV/c/m]
`P_Q`	the beam's magnetic rigidity, P/q	[MV/c]
`k1`	the focusing strength, k_1	[1/m^2]

Here follow a few formulæ to disentangle the relations between: the integrated strength, S; the focusing strength, k_1; the quadrupole gradient, G; and the beam rigidity, P/q or $B\rho$, with their units:

$$\begin{aligned} S &= P/q \cdot k_1 \cdot L = \left(\frac{P/q}{\mathrm{MV}/c}\right)\left(\frac{k_1}{1/\mathrm{m}^2}\right)\left(\frac{L}{\mathrm{m}}\right) && [\mathrm{MV}/c/\mathrm{m}] \\ &= G \cdot L = 299.792458\left(\frac{G}{\mathrm{T/m}}\right)\left(\frac{L}{\mathrm{m}}\right) \end{aligned}$$

$$\begin{aligned} k_1 &= \frac{G}{B\rho} = \left(\frac{G}{\mathrm{T/m}}\right)\left(\frac{\mathrm{T\,m}}{B\rho}\right) && [1/\mathrm{m}^2] \\ &= \frac{G}{P/q} = 299.792458\left(\frac{G}{\mathrm{T/m}}\right)\left(\frac{\mathrm{MV}/c}{P/q}\right) \end{aligned}$$

$$P/q = 299.792458\left(\frac{B\rho}{\mathrm{T\,m}}\right) \qquad [\mathrm{MV}/c]$$

Examples

If one has a 20 cm long quadrupole, with $k_1 = 0.1$ 1/m^2, for a proton beam with $P_{\mathrm{ref}} = 200$ MeV/c, one can use the following lines:

```
Lquad = 0.2; % m, quadrupole length
P = 200; % MeV/c, reference momentum
k1 = 0.1; % 1/m^2, focusing strength
Q = Quadrupole(Lquad, P, k1);
```

Or, given the quadrupole gradient, G:

```
Lquad = 0.2; % m, quadrupole length
G = 1.2; % T/m, quadrupole gradient
strength = 299.792458 * G * Lquad; % quadrupole integrated strength
Q = Quadrupole(Lquad, strength);
```

3.2.3 Sector bending dipole

3.2.4 Transfer line

3.2.5 Field maps

One of the strengths of RF-Track is the tracking in field maps. One-, two-, or three- dimensional field maps are accepted, both real and complex, to simulate static, standing-wave, or traveling-wave electromagnetic fields, both backward and forward traveling.

One-dimensional field maps allow the simulations of three-dimensional fields, just proving the field's longitudinal component along the symmetry axis. RF-Track provides to expand the field off-axis, in full respect of Maxwell's equations. Two-dimensional field maps allow the simulation of cylindrical-symmetric fields from just the field over one plane. Three-dimensional field maps allow tracking in the most generic electromagnetic fields. Two- and three- dimensional field maps accept Cartesian 3D mesh grids with regular mesh spacing.

Structure walls in field maps

Three-dimensional field maps can carry additional information than the field itself. They can contain information about the structure's walls embedding the field, using the special numeric quantity "Not-a-number" (NaN). In computing, NaN is a member of a numeric data type that can be interpreted as a value that is undefined or unrepresentable, especially in floating-point arithmetic. In RF-Track, the presence of NaNs in a field map is interpreted as walls and permits the detection of looses when the particles hit the walls during tracking. This is an extremely useful feature because it allows the computation of losses even in complicated geometries and detects the exact location where particles impact the walls in three dimensions.

Interpolation methods

RF-Track offers two interpolation methods

- Linear interpolation (LINT)
- Cubing interpolation (CINT) - which is the default

In the case of LINT, the interpolation uses the 8 closest vertexes of the 3D mesh cell enclosing the point of interest. Notice that in this case, the granularity of the loss detection coincides with the mesh cell size. In the case of CINT, the interpolation uses the 64 closest vertexes, as it considers the 3x3x3 mesh cells surrounding the point of interest. This means that the granularity of the loss detection is effectively 3 times the size of a mesh cell.

Cubic interpolation is in general better than linear, because it provides a smooth field over the entire volume, therefore more complying with the Maxwell's equations. The price to pay for this smoothness, is computational time.

3.2.6 Coil

3.2.7 Electron cooler

3.2.8 Standing wave structure

3.2.9 Traveling wave structure

3.2.10 Adiabatic Matching Device

3.3 Imperfections

3.3.1 Element misalignment

4. Particle Tracking

4.1 Tracking options

4.1.1 Integration algorithms

4.1.2 Transport table

Both `Lattice()` and `Volume()` offer the possibility to track average beam quantities such as beam size, emittance, energy spread, etc., during tracking, and store them in a "transport table". Such a transport table can be retrieved using the method `get_transport_table()`.

Like `get_phase_space()` for the beam, `get_transport_table()` allows the user to inquire about specific quantities. Table 4.1 lists all identifiers that are accepted.

To enable a transport table in `Volume()` it is sufficient to specify the option `tt_dt_mm` in the tracking options, specifying the time interval, expressed in mm/c, between two consecutive samplings.

4.2 Space-charge and beam-beam effects

4.2.1 Particle-to-particle algorithm

4.2.2 3D FFT algorithm

4.2.3 Mirror charges

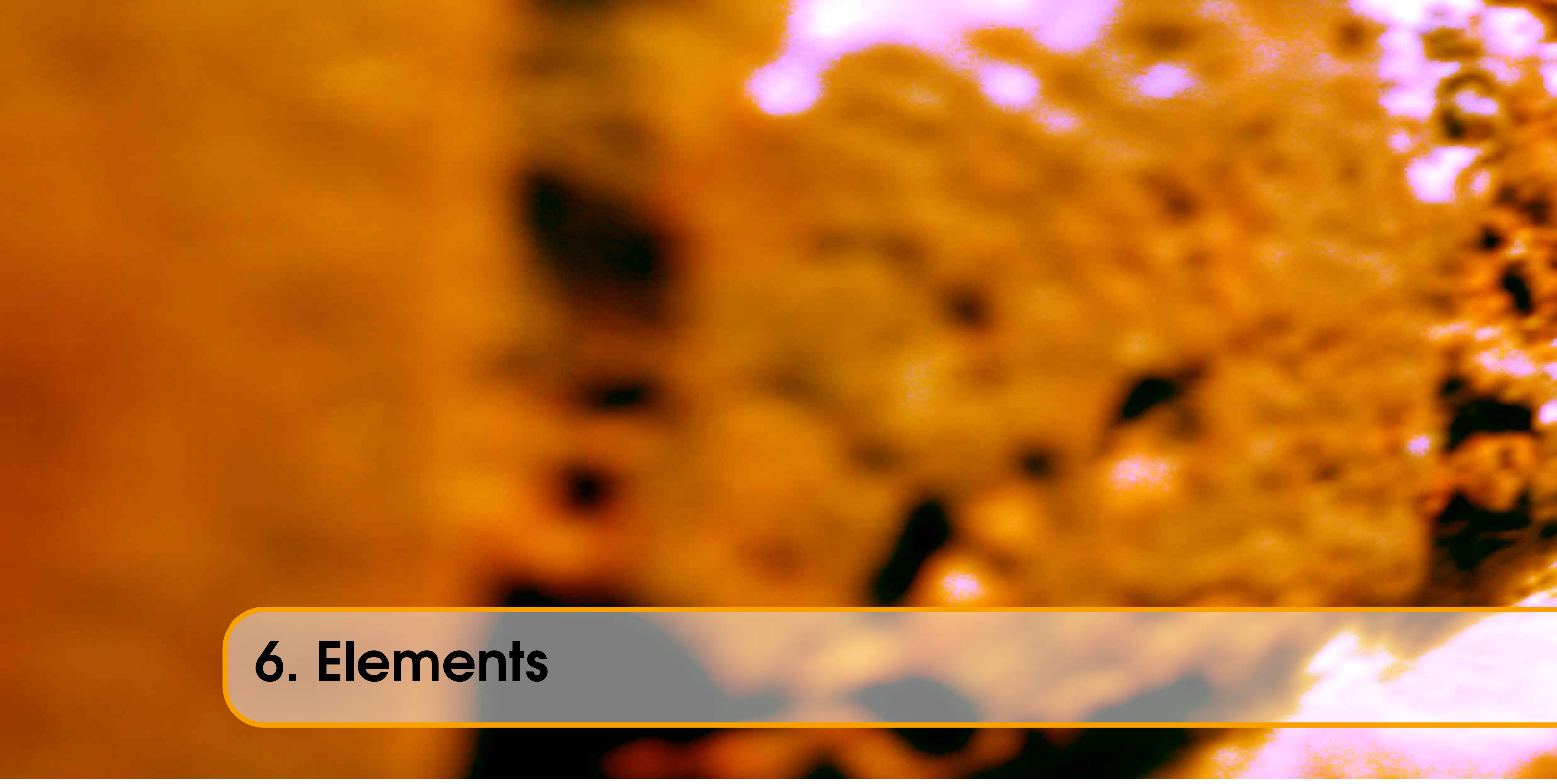
`%mean_X`	average horizontal position	[mm]
`%mean_Y`	average vertical position	[mm]
`%mean_S`	average longitudinal coordinate	[mm]
`%mean_K`	average kinetic energy	[MeV]
`%mean_E`	average total energy	[MeV]
`%t`	time	[mm/c]
`%emitt_x`	normalized horizontal emittance	[mm.mrad]
`%emitt_y`	normalized vertical emittance	[mm.mrad]
`%emitt_z`	normalized longitudinal emittance	[mm.permille]
`%beta_x`	horizontal beta function	[m]
`%beta_y`	vertical beta function	[m]
`%beta_z`	longitudinal beta function	[m]
`%alpha_x`	horizontal alpha function	[-]
`%alpha_y`	vertical alpha function	[-]
`%alpha_z`	longitudinal alpha function	[-]
`%sigma_X`	horizontal spread	[mm]
`%sigma_Y`	vertical spread	[mm]
`%sigma_Z`	longitudinal spread	[mm]
`%sigma_Px`	horizontal momentum spread	[MeV/c]
`%sigma_Py`	vertical momentum spread	[MeV/c]
`%sigma_Pz`	longitudinal momentum spread	[MeV/c]
`%N`	transmission	[#]

Table 4.1: List of identifiers accepted by `Bunch6dT::get_phase_space()`.

II

Physics Manual

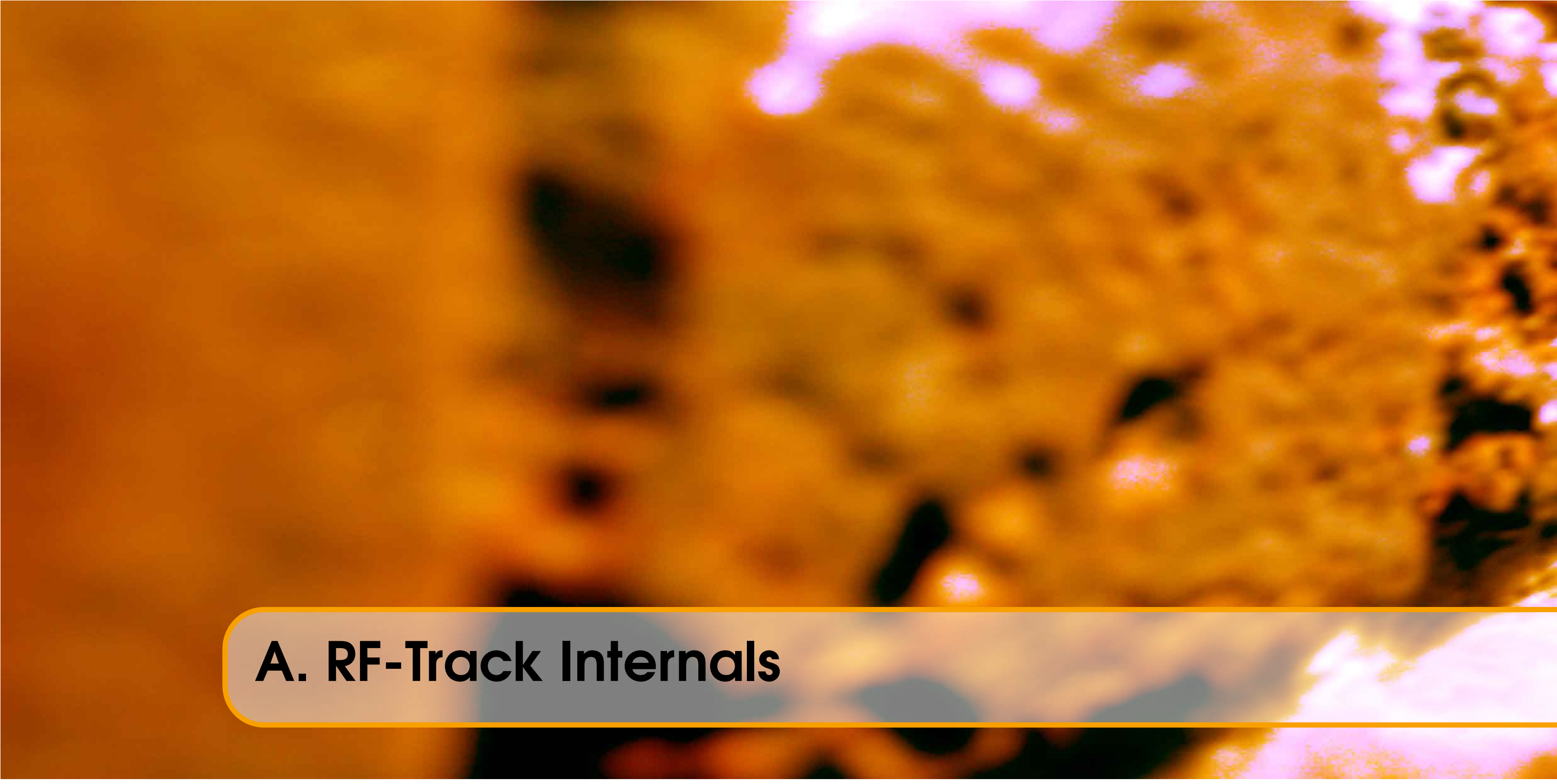
5. Particle Tracking



6. Elements

III

Appendix



A. RF-Track Internals

A.1 Bean dynamics

A.1.1 Dynamical variables

Table A.1 lists the single-particle dynamical variables, their formal definition, and the suggested variable name, often used in this manual.

Table A.1: RF-Track dynamical variables.

Name	Symbol	Definition	Variable name
Reference momentum	P_0		`P0`
Momenta	P_x, P_y, P_z		`Px`, `Py`, `Pz`
Normalized momenta	p_x, p_y, p_z	$p_x = P_x/P_0, \ldots$	`px`, `py`, `pz`
Transverse angles	x', y'	$x' = P_x/P_z, \ldots$	`xp`, `yp`

A.1.2 Equations of motion

A.2 Space charge

A.2.1 Integrated Green's functions

A.3 Numerical computing

A.3.1 Interpolation

Linear

Cubic

A.3.2 Discrete maps of vector fields

Divergence-Free

Curl-free

A.3.3 3D rotations

Quaternions

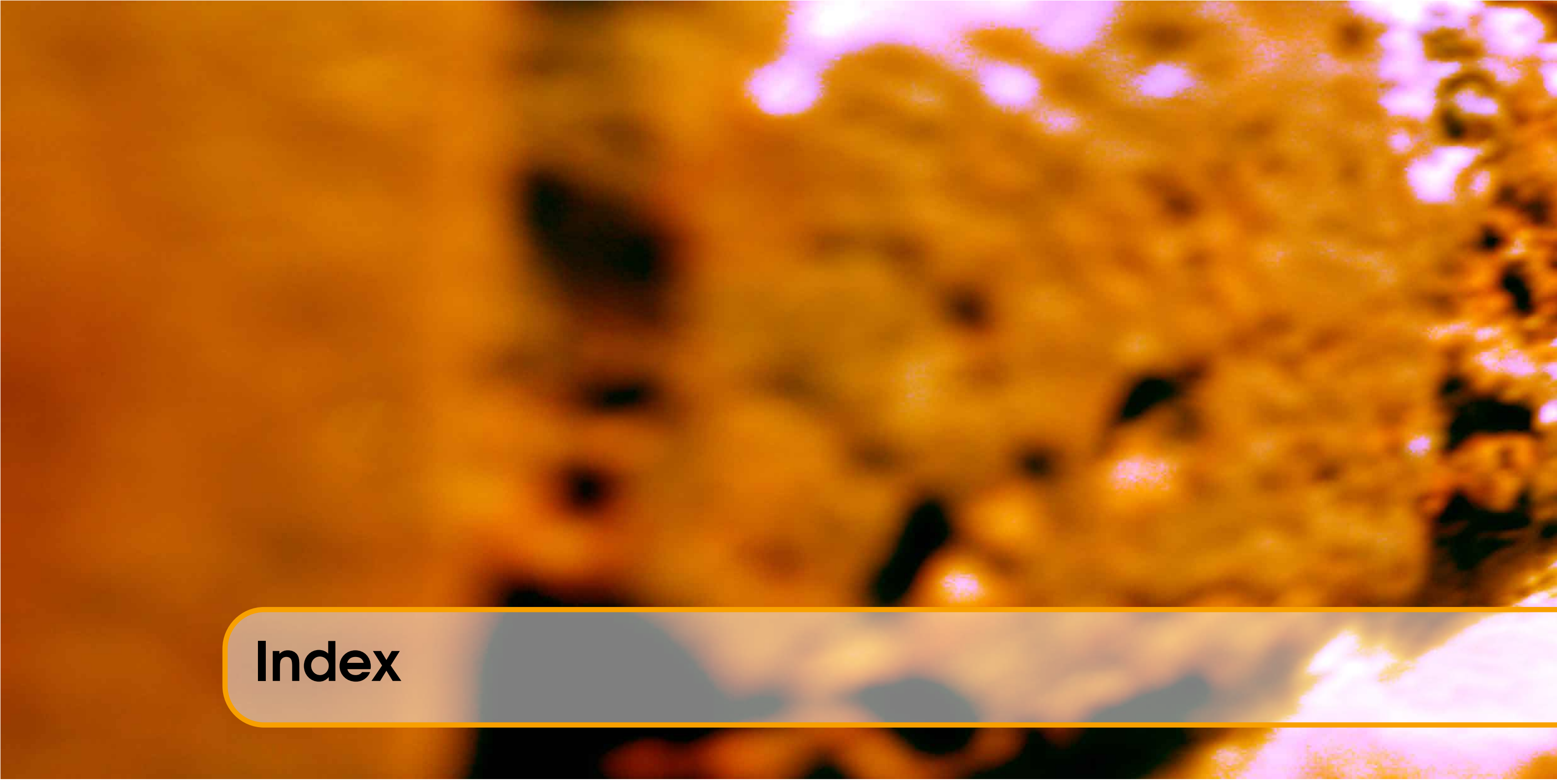
A.3.4 Random number generators

Table A.2 shows a list of the random number generators available to RF-Track. Consult [3] for a detailed description of each of them. The default is `mt19937`, which is among the fastest high-quality generators available.

Name	Description
`taus2`	Maximally equidistributed combined Tausworthe generator by L'Ecuyer
`mt19937`	Makoto Matsumoto and Takuji Nishimura generator
`gfsr4`	Lagged-fibonacci generator
`ranlxs0`	Second-generation version of the RANLUX algorithm of Luscher
`ranlxs1`	Like the previous by with increased order of strength
`ranlxs2`	Like the previous by with increased order of strength
`mrg`	Fifth-order multiple recursive generator by L'Ecuyer, Blouin and Coutre
`ranlux`	Implementation of the original algorithm developed by Luscher
`ranlux389`	Like the previous but gives the highest level of randomness
`ranlxd1`	Double precision output (48 bits) from the RANLXS generator
`ranlxd2`	Like the previous by with increased order of strength

Table A.2: List of random number generators available to RF-Track

Bibliography



Index