RF-Track Reference Manual

Version 2.5.5

Andrea Latina

AUTHOR AND CONTACT:

Andrea Latina
Beams Department
Accelerator and Beam Physics Group
CERN
CH-1211 GENEVA 23
SWITZERLAND

andrea.latina@cern.ch

Started in April 2020

Contents

User Manual

1. Introduction

RF-Track is a tracking code developed at CERN for optimising particle accelerators, offering 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. Fast, optimised, parallel algorithms allow element overlap and direct- and indirect-space-charge calculations.

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, underlying physical models, and 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 two powerful tools.

1.1 Getting started

RF-Track was developed on macOS in C++17 but runs on GNU/Linux and other POSIX-compliant systems. This manual section will describe preparing a suitable environment for compiling and running RF-Track.

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

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

1.1.1 Conventions used in this manual

This manual contains numerous examples which can be typed on the keyboard. Some examples 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

This manual uses a colour code to distinguish between shell, Octave, or Python commands. Shell commands entered at the terminal are shown in light grey:

```
$ 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 generally 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`, ...).

1.2 Installing RF-Track

1.2.1 On macOS

If you intend to use RF-Track on macOS with Python, it is sufficient to give the command

```
$ pip install RF_Track
```

If you intend to use RF-Track with Octave on macOS, you need to install MacPorts. If you don't have MacPorts installed, you will need to install it. If you already have it, it is good practice to update and upgrade your packages before you proceed:

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sudo port upgrade outdated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

Then, you need to issue the following command to install the required packages:

```
$ sudo port install fftw-3 gsl octave octave-optim
```

Now, you should download the file `RF_Track.oct` from the RF-Track page and put it in the Octave's path to use it.

1.2.2 On Linux

If you intend to use RF-Track in Linux with Python, it is sufficient to give the command

```
$ pip install RF_Track
```

If you intend to use RF-Track with Octave, the binaries provided on the RF-Track page require Ubuntu (or package-compatible distributions). 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. On Ubuntu 24.0.4, you can give the following command:

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

1.2.3 On Windows

All Ubuntu binaries will also work on Windows under the WSL environment (`https://learn.microsoft.com/en-us/windows/wsl/install`). To install RF-Track on your Windows, open PowerShell or the Windows command prompt in administrator mode by right-clicking and selecting "Run as administrator", then issue the following command:

```
wsl --install --web-download -d Ubuntu
```

You can then follow the steps for Ubuntu.

1.3 Running RF-Track

When using RF-Track with Octave, it is enough for your Octave to know where the RF-Track binary file is located. You can start your Octave scripts with, e.g., the command:

```
octave:1> addpath('/Users/<USERNAME>/Codes/rf-track-2.3.0');
```

Or you can add this command to the file `.octaverc` in your home directory to run it automatically whenever Octave is launched. Then, to load RF-Track, it is sufficient to give the command:

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

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

```
>>> import RF_Track
```

1.3.1 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 a 90-degree phase advance, and the beam consists of 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.

```
1 %% Load RF-Track
2 RF_Track;
```

```
3
4  %% Beam parameters
5  mass = RF_Track.electronmass;       % particle mass in MeV/c^2
6  population = 1e10;                  % number of particles per bunch
7  Q = -1;                             % particle charge in units of e
8  Pref = 5;                           % reference momentum in MeV/c
9  B_rho = Pref / Q;                   % beam magnetic rigidity in MV/c
10
11 %% FODO cell parameters
12 Lcell = 2; % cell length in m
13 Lquad = 0.0; % m, zero-length quadrupole (thin quadrupole)
14 Ldrift = Lcell/2 - Lquad; % drift space between the two quadrupoles
15
16 mu = 90; % phase advance per cell in deg
17 k1L = sind(mu/2) / (Lcell/4);       % 1/m, quadrupole focusing strength
18 strength = k1L * B_rho;             % MeV/m, quadrupole strength
19
20 %% Create the elements
21 Qf = Quadrupole(Lquad/2,  strength/2);  % half focusing quadrupole
22 Qd = Quadrupole(Lquad, -strength);      % defocusing quadrupole
23 Dr  = Drift(Ldrift); % drift space
24 Dr.set_tt_nsteps(100); % number of steps for the transport table
25
26 %% Create the lattice
27 FODO = Lattice();           % Create a new object Lattice() called FODO
28 FODO.append(Qf);            % 1/2 F
29 FODO.append(Dr);            %     O
30 FODO.append(Qd);            %     D
31 FODO.append(Dr);            %     O
32 FODO.append(Qf);            % 1/2 F
33
34 %% Define Twiss parameters
35 Twiss = Bunch6d_twiss();
36 Twiss.beta_x = Lcell * (1 + sind(mu/2)) / sind(mu); % m
37 Twiss.beta_y = Lcell * (1 - sind(mu/2)) / sind(mu); % m
38 Twiss.alpha_x = 0.0;
39 Twiss.alpha_y = 0.0;
40 Twiss.emitt_x = 1; % mm.mrad, normalized emittances
41 Twiss.emitt_y = 1; % mm.mrad
42
43 %% Create the bunch
44 B0 = Bunch6d(mass, population, Q, Pref, Twiss, 10000);
45
46 %% Perform tracking
47 B1 = FODO.track(B0);
48
```

```
49 %% Retrieve the Twiss plot and the phase space
50 T = FODO.get_transport_table('%S %beta_x %beta_y');
51 M = B1.get_phase_space('%x %xp %y %yp');
52
53 %% Make plots
54 figure(1)
55 hold on
56 plot(T(:,1), T(:,2), 'b-')
57 plot(T(:,1), T(:,3), 'r-')
58 legend({ '\beta_x ', '\beta_y ' })
59 xlabel('S [m]')
60 ylabel('\beta [m]')
61
62 figure(2)
63 scatter(M(:,1), M(:,2), '*')
64 xlabel('x [mm]')
65 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 makes the entire set of RF-Track commands and their predefined constants available to Octave (see the following section for details about them).

Lines 5-18 declare some Octave variables related to our problem that will be useful later. They include bunch parameters and FODO cell parameters. 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. Other constants exist and are listed in the following sections.

Lines 21-24 define the elements constituting 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 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 24 uses a *method* of the object Drift, which specifies that the drift must be divided into 100 steps. Dividing a drift into steps doesn't help the tracking itself, but here, it tells RF-Track that we want to sample average quantities like the emittances, beam size and Twiss parameters 100 times along the drift. This will produce a nice Twiss plot. 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 those beam quantities along the lattice.

Lines 27-32 define the FODO lattice itself. An object of type `Lattice()` is created with the name FODO. A `Lattice()` is an empty sequence at creation. Lines 27 through 31 append to `FODO` the elements that compose the cell. It is important to understand that internally, RF-Track stores in `FODO` a *copy* of these elements, not the elements themselves. Therefore, modifying `Qf`, `Qd`, or `Dr` after appending them will not affect `FODO`.

Lines 35-44 define the Twiss parameters and create the bunch `B0`, which is an instance of the object `Bunch6d()` using the defined Twiss parameters. The Twiss parameters are assigned to `Twiss`, an instance of `Bunch6d_twiss()`. `B0` is here created with 10'000 macro-particles.

Line 47 tracks the bunch `B0` through the FODO cell and stores the outcoming bunch in `B1`, another instance of type `Bunch6d()`.

Lines 50-51 retrieve the Twiss parameters and the phase space and store them in two Octave variables `T` and `M`.

Lines 54-65 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.

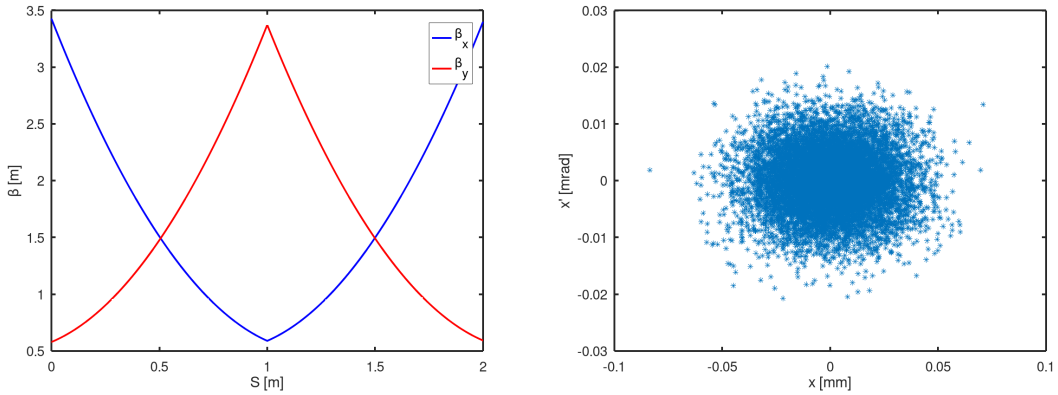


Figure 1.1: The output of the example.

1.3.2 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.

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

1.3.3 Predefined constants

Several constants are predefined within RF-Track for the user's convenience:

```
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
muonlifetime            % muon lifetime, in mm/c
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 write:

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

1.4 Run-time parameters and options

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

1.4.1 Parallelism

RF-Track is a parallel application. By default, RF-Track uses the maximum number of parallel threads available on your machine, which can be inquired via the RF-Track's variable "`max_number_of_threads`". The user can change this number (typically reducing it to avoid overloading the CPU) by setting the variable `number_of_threads`.
In Octave,

```
% The number of threads that RF-Track must use
RF_Track.number_of_threads = <YOUCHOOSE>;

% The max number of threads supported by your CPU [read-only]
RF_Track.max_number_of_threads
```

In Python,

```
% The number of threads that RF-Track must use
RF_Track.cvar.number_of_threads = <YOUCHOOSE>

% The max number of threads supported by your CPU [read-only]
RF_Track.max_number_of_threads
```

The desired number of threads that RF-Track should use (in this example as `<YOUCHOOSE>`) should not exceed `RF_Track.max_number_of_threads`. Note that the fastest RF-Track setup isn't necessarily achieved by using all the threads available on your system. Sometimes, using less can

lead to faster simulations. This depends on the CPU you are using, the number of particles you are tracking, and the simulation setup in terms of collective effects, transport table, and screens, i.e., the points where RF-Track needs to break parallel tracking to compute average or global bunch quantities. Each simulation case should be studied individually and a preliminary test with different numbers of threads is recommended to find the optimum number of threads.

1.4.2 Environment variables

A number of environment variables exist to customize RF-Track's runtime behaviour:

RF_TRACK_NUMBER_OF_THREADS

The number of threads that RF-Track uses can also be specified via the environment variable, `RF_TRACK_NUMBER_OF_THREADS`. This allows the user to specify the number of threads at runtime without modifying the input script. In this example:

```
$ RF_TRACK_NUMBER_OF_THREADS=8 octave-cli my_rft_script.m
```

Octave will execute the RF-Track script `my_rft_script.m` using a maximum of eight parallel threads. Similarly, with Python:

```
$ RF_TRACK_NUMBER_OF_THREADS=8 python my_rft_script.py
```

RF_TRACK_NO_UPDATE_CHECK

At start-up, RF-Track checks if a newer version is available. To turn off the automatic check, set this environment variable to '1'. Example:

```
$ export RF_TRACK_NO_UPDATE_CHECK=1
$ python my_rft_script.py
```

1.4.3 Random Number Generator

RF-Track uses high-quality random number generators. The user can use the following commands to set the desired random number generator and its initial seed.

```
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
```

The default random number generator is "mt19937" of Makoto Matsumoto and Takuji Nishimura, known as the "Mersenne Twister" generator, which is among the fastest high-quality generators available. Table 1.2 shows a list of the random number generators available to RF-Track. Consult [3] for a detailed description of each of them.

1.4.4 Version number

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

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 1.2: List of random number generators available to RF-Track

Returns the RF-Track version number.

1.5 Notes for Python users

Three important points for Python users:

1. In RF-Track, the input arguments of each function are *positional*, not *keyword* arguments as Python users may be used to and expect. This means that you cannot change their order.
2. In this manual, the syntax `function(arg1=var1, arg2=var2)` is purely illustrative, as it is intended to help the reader immediately know the meaning of each input parameter and the default value if the parameter is omitted. It does not indicate the use of *keyword* arguments.
3. It is also important to know that whenever RF-Track's inputs are vectors or matrices, RF-Track expects `numpy` arrays. Therefore, Octave lines like

```
Quad = Multipole (length, [0, k1L])
Sext = Multipole (length, [0, 0, k2L])
```

must be written in Python as

```
import RF_Track as rft
import numpy as np

Quad = rft.Multipole(length, np.array([0, k1L]))
Sext = rft.Multipole(length, np.array([0, 0, k2L]))
```

1.6 Further information

1.6.1 Acknowledgments

The author gratefully acknowledges the many colleagues who contributed to the development and validation of RF-Track. Early users — Stefano Benedetti, Costanza Agazzi, Yanliang Han, Yongke

Zhao, Luca Garolfi, Elena Fol, Alexander Malyzhenkov, and Vlad Musat — provided essential feedback that strengthened the code and guided its adaptation to real-life applications.

Feature contributions came from Mohsen Dayyani Kelisani (analytic RF structures), Bernd Michael Stechauner (multiple Coulomb scattering models), Avni Aksoy (ASTRA-like particle generator), and Riccardo De Maria (TPSA clarifications). PhD students Javier Olivares Herrador and Paula Desiré Valdor contributed the beam-loading and intra-beam scattering models, while Summer Student Lale Balat helped build a robust test suite. Stewart Boogert interfaced BDSIM to RF-Track.

The author also acknowledges collaborations with the CLEAR facility at CERN, the Oncology Department of Lausanne University Hospital (CHUV), and the CLIC and FCC study groups, which provided valuable opportunities for benchmarking and applications, as well as the support of the Accelerator and Beam Physics Group, Beams Department, CERN.

1.6.2 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, 2024
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 = {2024}
}
```

2. Beam Models

2.1 Tracking in time or in space

RF-Track implements two particle tracking methods: tracking *in time* and tracking *in space*. The tracking *in time* should be preferred in space-charge-dominated regimes, where the relative positions of the particles in space matter. 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 each of these two beam types: `Lattice` for `Bunch6d`, and `Volume` `Bunch6dT`. This chapter will describe them in detail.

2.2 Single-bunch beams

2.2.1 Bunch6d

When `Bunch6d` is used, the tracking is performed using the accelerator longitudinal coordinate S as the integration variable. This corresponds to what is described in accelerator physics textbooks, which is at the basis of matrix-based beam optics. In this model, all particles are on the same plane at a given longitudinal coordinate S at the beginning of an element and are then transported to the end of the element, updating 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, along with the units used by RF-Track internally, follows:

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 along the Lattice, which can be accessed like this:

```
current_S = B.S; % m
```

Emittance calculation

When using a `Bunch6d`, the normalised emittances are computed as:

$$\begin{aligned}
\varepsilon_x &= \beta_{\text{avg}}\,\gamma_{\text{avg}}\det\left(\text{cov}\left\{x, x'\right\}\right)^{1/2} \\
\varepsilon_y &= \beta_{\text{avg}}\,\gamma_{\text{avg}}\det\left(\text{cov}\left\{y, y'\right\}\right)^{1/2} \\
\varepsilon_z &= \frac{1}{m}\det\left(\text{cov}\left\{t, E\right\}\right)^{1/2} \\
\varepsilon_{4\text{D}} &= \beta_{\text{avg}}\,\gamma_{\text{avg}}\det\left(\text{cov}\left\{x, x', y, y'\right\}\right)^{1/4} \\
\varepsilon_{6\text{D}} &= \beta_{\text{avg}}\,\gamma_{\text{avg}}\det\left(\text{cov}\left\{x, x', y, y', t, \frac{E}{P_{\text{avg}}}\right\}\right)^{1/6}
\end{aligned}$$

Constructors

A new `Bunch6d` can be created in multiple ways. The most common ones are two: from a set of Twiss parameters or directly from a beam matrix with the phase space. Multi-species bunches can be created using a beam matrix of the extended phase space. Here is 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 ID ] );
B = Bunch6d(mass, population, charge, [ X XP Y YP T P ] );
B = Bunch6d( [ X XP Y YP T P MASS Q N ID ] );
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 2.4.1)	
`nParticles`	number of macro-particles in bunch	[#]
`sigmaCut`	if > 0 cuts the distributions at `sigma_cut` sigmas	[#]
`X`	column vector of the horizontal coordinates	[mm]
`Y`	column vector 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 vector of the x' angles	[mrad]
`YP`	column vector 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	[#]
`ID`	column vector of particle ID's	[INTEGER]

With no arguments, the default constructor allows the creation of an empty beam. The return value is an object of type `Bunch6d`.

Accessing the phase space

The particle coordinates can be inquired using the method `get_phase_space()`. This method allows retrieving the bunch's particle distribution in multiple ways.

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

The two arguments are:

`format_fmt`	This parameter allows the user to specify and format what phase space representation should be returned
`which`	`'all'|'good'`. This parameter specified whether all particles should be returned, including the lost ones, or just the good ones (default=`'good'`)

Table 2.1 shows all possible identifiers accepted by `get_phase_space()`. The return value of this function, `M`, is a matrix with the requested phase space.

Modifying the phase space

The particle coordinates can be changed by the user using the method `set_phase_space()`:

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

The only argument accepted by this method is a 6-column matrix containing the phase space in `Bunch6d` format, that is, the phase space columns are x, y, the particle's transverse position in mm, x', y' the angles in mrad, t the arrival time in mm/c, and P the total momentum in MeV/c, in the

`%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 a 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]
`%tp`	inverse of V_z	[mrad/c]
`%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, $p_t = (E - E_0)/P_0 c$	[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	[#]
`%id`	particle's id	[#]

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

order given above (x, x', y, y', t, P).

2.2.2 Bunch6dT

`Bunch6dT` allows tracking "in time". When `Bunch6dT` is used, the time, t, is the integration variable. The 6D phase-space coordinates of each particle are (X, Y, Z, P_x, P_y, P_z). `Bunch6dT` maintains, t, the clock common to all particles, updated at each integration step.

In `Bunch6dT`, the beam is represented by a set of macro-particles whose state vector is an extended phase space:

$$\begin{pmatrix} X, & P_x, & Y, & P_y, & Z, & P_z, & m, & Q, & N, & t_0 \end{pmatrix}$$

The positions and momenta of each particle are updated as they are transported through the accelerator. The meaning of each symbol, along with the units used to store the information internally, follows:

X, Y, Z	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]

In `Bunch6dT`, a major difference from `Bunch6d` is the presence of t_0, the creation time of each particle. This allows, for example, the simulation of cathodes and particle emission. `Bunch6dT` also stores the time t at which the bunch is taken, which can be accessed (read and write) like this:

```
current_time = B.t; % mm/c
```

Emittance calculation

When using a `Bunch6dT`, the normalised emittances are computed as:

$$\begin{aligned}
\varepsilon_x &= \frac{1}{m}\det\left(\mathrm{cov}\left\{x, P_x\right\}\right)^{\frac{1}{2}} \\
\varepsilon_y &= \frac{1}{m}\det\left(\mathrm{cov}\left\{y, P_y\right\}\right)^{\frac{1}{2}} \\
\varepsilon_z &= \frac{1}{m}\det\left(\mathrm{cov}\left\{z, P_z\right\}\right)^{\frac{1}{2}} \\
\varepsilon_{4\mathrm{D}} &= \frac{1}{m}\det\left(\mathrm{cov}\left\{x, P_x, y, P_y\right\}\right)^{\frac{1}{4}} \\
\varepsilon_{6\mathrm{D}} &= \frac{1}{m}\det\left(\mathrm{cov}\left\{x, P_x, y, P_y, z, P_z\right\}\right)^{\frac{1}{6}}
\end{aligned}$$

Constructors

A new `Bunch6dT` can be created in multiple ways. The most common ones are two: 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 Z Pz ID ] );
B = Bunch6dT(mass, population, charge, [ X Px Y Py Z Pz ] );
B = Bunch6dT( [ X Px Y Py Z Pz MASS Q N T0 ID ] );
B = Bunch6dT( [ X Px Y Py Z Pz MASS Q N T0 ] );
B = Bunch6dT( [ X Px Y Py Z Pz MASS Q N ] );
B = Bunch6dT( [ X Px Y Py Z 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 2.4.1)	
`nParticles`	number of macro-particles in bunch	[#]
`sigmaCut`	if > 0 cuts the distributions at `sigma_cut` sigmas	[#]
`X`	column vector of the horizontal coordinates	[mm]
`Y`	column vector of the vertical coordinates	[mm]
`Z`	column vector of the longitudinal coordinates	[mm]
`Px`	column vector of the horizontal momenta	[MeV/c]
`Py`	column vector of the vertical momenta	[MeV/c]
`Pz`	column vector 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]
`ID`	column vector of particle ID's	[INTEGER]

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

Accessing the phase space

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

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

The two arguments are:

`format`	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` is a matrix with the requested phase

`%X`	horizontal coordinate	[mm]
`%Y`	vertical coordinate	[mm]
`%Z`	longitudinal coordinate	[mm]
`%deg@MHz`	longitudinal coordinate in degrees at a 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]
`%tp`	inverse of V_z	[mrad/c]
`%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, $p_t = (E - E_0)/P_0 c$	[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	[#]
`%id`	particle's id	[#]

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

space.

Modifying the phase space

The user can change particle coordinates using the method `set_phase_space()`.

```
B.set_phase_space( [ X Px Y Py Z Pz ] );
```

The only argument is a 6-column matrix containing the phase space. The input phase space's columns are X, Y, Z, the absolute position in mm, and P_x, P_y, P_z, the absolute momentum in MeV/c, in the order given in the above (X, P_x, Y, P_y, Z, P_z).

2.2.3 Conversion between Bunch6d and Bunch6dT

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

Since a `Bunch6dT` represents a particle distribution in the 3D space, a bunch of type `Bunch6dT` cannot be converted into a bunch of type `Bunch6d`, where all the particles are sampled at the same longitudinal location.

2.2.4 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 beam property affects the space charge calculation and has no other effect on the tracking.

2.2.5 Particles lifetime

The particle lifetime can be set using the method `set_lifetime(T)` of both the objects `Bunch6d` and `Bunch6dT`. Example:

```
B.set_lifetime(T);
```

The only argument of this method is:

`T`	the lifetime	[mm/c]

For convenience, the muon lifetime is predefined so that one can use it directly:

```
B.set_lifetime (RF_Track.muonlifetime );
```

For now, a particle that reaches the end of its lifetime is flagged as lost. No new particles are created.

2.2.6 Offsetting the beam

In some studies, it is useful to evaluate the accelerator performance in the presence of beam misalignments, either transverse, longitudinal, or both. RF-Track provides a convenient way to introduce such offsets directly at the bunch level via the `displaced` method, available for both `Bunch6d` and `Bunch6dT` objects.

The following example illustrates its usage:

```
B_offset = B.displaced (dx, dy, dz, dt, roll, pitch, yaw);
B_offset = B.displaced (dx, dy, dz, dt);
B_offset = B.displaced (dx, dy, dz);
```

In all cases, `B_offset` represents a *displaced copy* of the original bunch `B`; the original bunch remains unchanged.

The input arguments have the following meaning:

`dx`	Horizontal displacement	[mm]
`dy`	Vertical displacement	[mm]
`dz`	Longitudinal displacement	[mm]
`dt`	Time offset	[mm/c]
`roll`	Rotation around the Z axis	[mrad]
`pitch`	Rotation around the X axis	[mrad]
`yaw`	Rotation around the Y axis	[mrad]

Conceptually, this operation places the bunch on a plane whose origin is located at $(dx,\ dy,\ dz)$ and whose orientation is defined by the rotations `roll`, `pitch`, and `yaw` with respect to the accelerator reference frame. The particles are then drifted to the entrance plane of the next element. Then, the time offset `dt` is added to each particle.

This approach allows one to model beam offsets and angular misalignments in a physically consistent way, making it particularly suitable for tolerance studies, injection error analyses, and sensitivity evaluations.

2.3 Multi-bunch beams

RF-Track can simulate multi-bunch beams. In RF-Track, a `Beam` is a set of single bunches that the user can specify individually at arbitrary distances from each other.

This model allows a specialised implementation of long-range collective effects for multi-bunch beams. For example, specific single-bunch effects such as space charge are applied to each bunch individually, while bunch-to-bunch effects such as long-range wakefields can be computed considering the large spacing between bunches using a dedicated long-range algorithm. In addition, accessing information about each bunch and analysing the results after tracking is much easier.

The “Beam” model provides high flexibility, for example, by allowing the user to simulate bunch trains where some bunches are multi-particles and others single particles, thus speeding up simulations where one needs to focus on a specific bunch. Moreover, the bunches can be arbitrarily and irregularly spaced, have different charges and even be of different species.

2.3.1 Example of Beam Definition

The following lines provide an example declaration of a train consisting of 30 equally spaced bunches:

```
% Define a bunch
bunch = Bunch6d(mass, charge, q, phase_space);

% Define the train structure
n_bunches = 30; % number of bunches in the train
bunch_spacing = 1/3 * RF_Track.ns; % mm/c, bunch spacing

% Define a beam
B0 = Beam(n_bunches, bunch, bunch_spacing);
```

Bunches can also be added to a `Beam` by using the method `append()`, that comes in three forms:

```
B0.append (bunch );
B0.append (bunch, spacing );
B0.append (n_bunches, bunch, spacing );
```

In the first form, a bunch is added to the Beam with a specified `spacing`. Note that if the Beam is empty, the spacing parameter is ignored. The second form adds `n_bunches` to the beam; like in the previous form, if the Beam is empty, the spacing parameter is ignored for the first bunch. In the third form, the `bunch` is added to the beam as it is; no particular spacing is imposed.
The input arguments are:

`bunch`	a `Bunch6d` (or `Bunch6dT` with `BeamT`)	[`Bunch6d` | `Bunch6dT`]
`spacing`	the bunch spacing	[mm/c | mm]
`n_bunches`	the number of bunches to be added	[INTEGER]

Follows an example,

```
B0 = Bunch6d([...]); % create a bunch

Beam0 = Beam(); % create an empty beam

Beam0.append( B0 ); % add a first bunch
Beam0.append( B0, 10 * RF_Track.ps ); % add a bunch, 10 ps from the 1st
Beam0.append( B0, 20 * RF_Track.ps ); % add a bunch, 20 ps from the 2nd
```

It is important to mention that the first `Bunch6d` added to a `Beam` is added "as it is". The arrival time of the following bunches is adjusted to maintain the user-requested bunch spacing.

Once a `Beam` is initialized, it can be used for tracking. If "`Line`" is an arbitrary beamline, then `B1 = Line.track(B0);` is the outcoming train, where `B1{1}` is the first bunch, `B1{2}` is the second, etc.

```
B1 = Line.track(B0);
B1_0 = B1{1}; % The first bunch
B1_1 = B1{2}; % The second bunch
B1_2 = B1{3}; % The third bunch
```

In Python,

```
B1 = Line.track(B0)
B1_0 = B1[0]  # The first bunch
B1_1 = B1[1]  # The second bunch
B1_2 = B1[2]  # The third bunch
```

2.4 Twiss parameters

The structures `Bunch6d_twiss` and `Bunch6dT_twiss` gather all the information to generate a beam from the Twiss parameters.

2.4.1 Bunch6d_twiss

```
T = Bunch6d_twiss();
T.emitt_x; % mm.mrad, normalised horizontal emittance x.px
T.emitt_y; % mm.mrad, normalised vertical emittance y.py
T.emitt_z; % mm.permille, normalised longitudinal emittance t.pt
T.sigma_t; % mm/c, rms bunch duration
T.sigma_pt; % permille, normalised energy spread std(E-E_ref)/P_ref
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_px; % rad, horizontal dispersion prime
T.disp_y;  % m, vertical dispersion
T.disp_py; % rad, vertical dispersion prime
T.disp_z;  % m, longitudinal dispersion
```

Notice that the longitudinal phase space can be specified in three alternative ways:

1. Giving both the normalised longitudinal emittance `emitt_z` and the Twiss parameter β_z.
2. Giving the normalised longitudinal emittance `emitt_z` and either `sigma_t` or `sigma_pt`.
3. Giving both `sigma_t` and `sigma_pt`.

2.4.2 Bunch6dT_twiss

```
T = Bunch6dT_twiss();
T.emitt_x; % mm.mrad, normalised horizontal emittance x.px
T.emitt_y; % mm.mrad, normalised vertical emittance y.py
T.emitt_z; % mm.permille, normalised longitudinal emittance z.pz
T.sigma_z; % mm, rms bunch length
T.sigma_pz; % permille, norm. long. momentum spread std(Pz-P_ref)/P_ref
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_px; % rad, horizontal dispersion prime
T.disp_y;  % m, vertical dispersion
T.disp_py; % rad, vertical dispersion prime
T.disp_z;  % m, longitudinal dispersion
```

Similarly to `Bunch6d_Twiss`, the longitudinal phase space can be specified in three alternative ways:

1. Giving both the normalised longitudinal emittance `emitt_z` and the Twiss parameter β_z.
2. Giving the normalised longitudinal emittance `emitt_z` and either `sigma_z` or `sigma_pz`.
3. Giving both `sigma_z` and `sigma_pz`.

2.5 Inquiring the bunch properties

The user can inquire about a bunch's statistical quantities, such as the average value or the standard deviation of the phase space variables, the emittances, or the Twiss parameters, by calling the method:

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

Two versions exist, for `Bunch6d` and for `Bunch6dT`.

2.5.1 Bunch6d_info

If `B` is a bunch of type `Bunch6d`, `B.get_info()` returns:

```
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_Px; % average Px in MeV/c
```

```
I.mean_Py; % average Py in MeV/c
I.mean_Pz; % average Pz in MeV/c
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_t; % mm/c
I.sigma_xp; % mrad
I.sigma_yp; % mrad
I.sigma_xpx; % mm*mrad
I.sigma_ypy; % mm*mrad
I.sigma_tpt; % mm/c*permille
I.sigma_E; % energy spread in MeV
I.sigma_P; % momentum spread in MeV/c
I.emitt_x; % mm.mrad, normalised emittance
I.emitt_y; % mm.mrad, normalised emittance
I.emitt_z; % mm.permille, normalised emittance (sigma_t*sigma_d)
I.emitt_4d; % mm.mrad, 4d normalised emittance
I.emitt_6d; % mm.mrad, 6d 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; % the total number of particles in the bunch
```

2.5.2 Bunch6dT_info

If `B` is a bunch of type `Bunch6dT`, `B.get_info()` returns:

```
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; % energy spread in MeV
I.emitt_x; % mm.mrad normalised emittance
I.emitt_y; % mm.mrad normalised emittance
I.emitt_z; % mm.permille, normalised emittance (sigma_z*sigma_pt)
I.emitt_4d; % mm.mrad, 4d normalised emittance
I.emitt_6d; % mm.mrad, 6d 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; % the total number of particles in the bunch
```

2.6 Bunch persistency

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

2.6.1 Saving and loading a beam on disk

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

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

The beam is saved as a binary file. This ensures that the saved and loaded beams are bit-wise identical. Note that the binary format used by each architecture to store double-precision numbers is hardware- and architecture-dependent and may vary from computer to computer. For this reason, a particular architecture may be unable to read a file saved on another hardware architecture because their internal representation of double precision numbers is different.

2.6.2 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.

2.6.3 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.

2.6.4 Saving the phase space

There are also alternative ways to save a beam as a file. For example, one can extract the phase space using `get_phase_space()` and then save the phase-space matrix using appropriate Octave or Python commands.

2.7 Spin Polarization Tracking

Since version 2.5.0, RF-Track can track spin polarization. Each particle has an associated anomalous magnetic moment G and a spin vector $\mathbf{S}$, a unit 3D vector representing the direction of the particle's spin polarization in its rest frame. The evolution of the spin vector $\mathbf{S}$ follows the Thomas–BMT equation:

$$\frac{d\mathbf{S}}{dt} = \Omega \times \mathbf{S},$$

where Ω is the spin precession vector

$$\Omega = -\frac{q}{m}\left[\left(G+\frac{1}{\gamma}\right)\mathbf{B} - \frac{G\gamma}{\gamma+1}(\beta\cdot\mathbf{B})\beta - \left(G+\frac{1}{\gamma+1}\right)\beta\times\mathbf{E}\right]$$

and

- q is the particle charge,
- m is the particle mass,
- γ is the Lorentz factor,
- $G = \frac{g-2}{2}$ is the particle anomalous magnetic moment,
- $\beta = \mathbf{v}/c$ is the normalized velocity,
- $\mathbf{B}, \mathbf{E}$ are the magnetic and electric fields in the lab frame (in T and V/m).

Notice that RF-Track considers the evolution of the spin vector due to both electric $\mathbf{E}$ and magnetic $\mathbf{B}$ fields.

2.7.1 Setting the polarization

Once a bunch `B0` is created, the spin can be set using one of the two methods:

```
B0.set_polarization (anomalous_magnetic_moment, P );
B0.set_polarization (anomalous_magnetic_moment, P, Sx, Sy, Sz);
```

In these methods, the input arguments are:

`anomalous_magnetic_moment`	the anomalous magnetic moment G
`P`	the beam polarization, as a number or as a matrix
`Sx`, `Sy`, `Sz`	the spin polarization vector

In the first form, `P` can be a matrix with three columns and as many rows as particles in the beam. In this case, each row indicates the polarization vector of each particle, and the method sets the spin of each particle to the input. If `P` is a real scalar number between 0 and 1, then the particles of the bunch are initialized with a degree of *vertical* polarization equal to `P`.

In the second form, `P` is the degree of polarization, whereas S_x, S_y, and S_z indicate an arbitrary spin polarization vector. For example, with `P` $= 0.8$ and $S_x = S_y = 0$, and $S_z = 1$, one creates a beam with 80% *longitudinal* polarization:

```
% Define the parameters
A = RF_Track.electron_anomalous_magnetic_moment; % A = (g-2)/2
P = 0.8; % 80% polarization
Sx = 0;
```

```
Sy = 0;
Sz = 1;

% Set the polarization of B0
B0.set_polarization (A, P, Sx, Sy, Sz);
```

The `anomalous_magnetic_moment` is a user-defined quantity. Predefined values are listed in Table 2.3.

Variable Name	Particle	Value
`RF_Track.electron_anomalous_magnetic_moment`	electron	0.00115
`RF_Track.proton_anomalous_magnetic_moment`	proton	1.792
`RF_Track.muon_anomalous_magnetic_moment`	muon	0.00116

Table 2.3: Pre-defined anomalous magnetic moments G of the electron, proton, and muon.

Important Remark

In RF-Track v2.4.1, the polarization vector of each particle is automatically normalized to 1. Starting with v2.4.2, however, this automatic normalization has been removed. Now, the input polarization of a particle can be any number between 0 and 1. The physical implications of this modification are as follows:

- RF-Track v2.4.1 assumes that the spin vector of each macroparticle has a magnitude of 1. This means that each macroparticle is fully polarized in a specific direction. In other words, all of the real particles within the macroparticle have the same identical spin orientation.
- RF-Track v2.4.2 and later versions allow the polarization vector of each macroparticle to be less than 1 (still within the range of 0 to 1). This means that the spin vector of each macroparticle is the average polarization of the real particles it represents.

Note that any input polarization larger than 1 will still be reduced to 1.

2.7.2 Inquiring the polarization

The polarization vector of each particle in a bunch can be accessed using `get_phase_space()` and the special identifiers `%Sx`, `%Sy`, and `%Sz`. The average polarization of a bunch can be tracked through a beamline using the `transport_table` with the special identifiers `%mean_Sx`, `%mean_Sy`, and `%mean_Sz`.

```
% Polarization vector of each particle
S1 = B1.get_phase_space ('%Sx %Sy %Sz');

% After tracking in a lattice L,
% get the average vertical polarization along S
T = L.get_transport_table ('%S %mean_Sy');
```

2.7.3 Example of spin polarization tracking

This example tracks the polarization of a bunch through a solenoid magnet designed to rotate the spin from vertical to horizontal. These solenoid and beam parameters are taken from the design of the Spin Rotator described in the ILC Technical Design Report.

```
% Initial phase space, all particles at Z = -5m with momentum 5 GeV/c
B = zeros(1000, 6); % 1000 rows x 6 columns
B(:,5) = -5000; % [mm]    Z = -5m
B(:,6) = 5000;  % [MeV/c] P = 5 GeV/c

% Create a Bunch6dT for Volume
B0 = Bunch6dT (RF_Track.electronmass, 0.0, -1, B);

% Set the polarization to 0.8 vertical
G = RF_Track.electron_anomalous_magnetic_moment;
P = 0.8;
Sx = 0;
Sy = 1;
Sz = 0;
B0.set_polarization(G, P, Sx, Sy, Sz);

% ILC Spin Rotator Solenoid
Lsol = 8.32; % m, solenoid length
Bsol = 3.15; % T, solenoid on-axis field
Rsol = 0.1;  % m, solenoid aperture radius
Sol = Solenoid (Lsol, Bsol, Rsol);

% Create a Volume with the ILC Spin Rotator Solenoid
V = Volume();
V.set_s0 (-5)  % Left boundary
V.set_s1 (+5)  % Right boundary
V.add (Sol, 0, 0, 0, 'center') % Solenoid centered at 0,0,0

% Integration parameters
V.dt_mm = 10;
V.odeint_algorithm = 'rk2';
V.tt_dt_mm = 50;

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

% Make plots
figure(1)
hold on
S = V.get_transport_table('%mean_S %mean_Sx %mean_Sy %mean_Sz');
plot(S(:,1) / 1e3, S(:,2), 'displayname', 'S_x', 'linewidth', 2)
plot(S(:,1) / 1e3, S(:,3), 'displayname', 'S_y', 'linewidth', 2)
plot(S(:,1) / 1e3, S(:,4), 'displayname', 'S_z', 'linewidth', 2)
legend('box', 'off')
xlabel('S [m]')
```

```
46  ylabel('Polarization')
```

The result of this example is visible in Fig. 2.1.

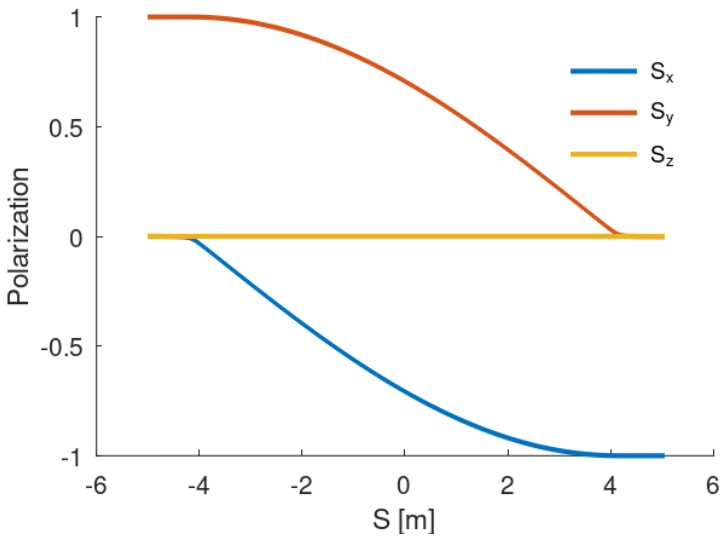


Figure 2.1: Example of Spin Tracking. The polarization is moved from vertical to horizontal.

3. Beam Tracking

3.1 Tracking environments

RF-Track offers two distinct environments to track particles: the `Lattice` and the `Volume`.

The environment `Lattice` represents the accelerator as a list of consecutive elements. `Lattice` works with `Bunch6d` and transports the beam, element by element, from the entrance to the exit plane of each element.

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). 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 formulæ and permeates the whole 3D space, allowing for the simulation of realistic fringe fields.

It must be mentioned that the environment `Lattice` can also apply space-charge effects to the beam. However, since `Bunch6d` maintains the distribution of the particles on the same longitudinal plane, the calculation of the space-charge force needs an on-the-fly extrapolation of each particle's longitudinal position, using the arrival time and the velocity to reconstruct the three-dimensional spatial distribution. Because of this somehow nonphysical manipulation of the phase space, we deem `Lattice` more suitable for sections of the accelerator free of space-charge effects. On the contrary, `Volume` –which through `Bunch6dT` maintains the full spatial distribution of the beam– should be the preferred choice in space-charge-dominated regimes. These considerations and the capability to superimpose elements make `Volume` perfect for the simulation of injectors, where a solenoidal magnetic field typically surrounds the gun's acceleration field, and space charge effects are critical. The additional flexibility of `Volume` comes at the cost of being computationally more expensive.

Summarising, `Lattice` is straightforward and fast, and its use is recommended in space-charge-

free regions. `Volume` is more flexible and can handle space-charge effects, but it's CPU-consuming.

This chapter describes these two environments and all the elements available to the user.

3.2 Lattice

Before setting up a lattice, one needs to create an object of type `Lattice`:

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

All options regarding the integration algorithms or the presence of collective effects are element-dependent and must be specified element by element.

3.2.1 Adding elements

One can use the methods `append` or `insert` to add elements to a Lattice. With both methods, one adds elements at the end of the existing Lattice. The method `append` accepts individual elements or lattices. When a Lattice is appended to another Lattice, it is seen by the enclosing Lattice as a single element that embeds other elements. This can be useful to simulate "girders" for example, rigid supports holding several elements together. Given that each element can be misaligned independently, one can have misaligned "girders" holding misaligned elements.

When a Lattice is added to another Lattice using the method `insert`, its single elements are appended one by one to the new Lattice, and the notion of the initial Lattice as a container is lost.

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

It is important to understand that these methods add a *copy* of the element to a Lattice, not the element itself. In C++ jargon, one would say that the elements are added "by value" – and not "by reference". This means that, after appending or inserting an element to a Lattice, whatever modification one applies to the original element will not affect the *copy* added to a Lattice.

If one wants to modify one element *after* the element has been added to a Lattice, and avoid wasting memory when appending several instances of the same element, one can pass the element "by reference". This can be done using the following methods:

```
L.append_ref(element);
L.insert_ref(lattice);
```

For example, this could be useful when dealing with large field maps: adding multiple identical copies of the same field map to a Lattice would be just memory-consuming and redundant.

3.2.2 Importing a MAD-X lattice

A dedicated constructor allows RF-Track to convert an entire MAD-X Twiss file into an RF-Track Lattice:

```
L = Lattice ('twiss_file.tws');
```

In this form, `Lattice` imports the entire Twiss file `'twiss_file.tws'` from MAD-X and creates a Lattice containing the corresponding elements. The Twiss file must be saved in MAD-X using the commands:

```
select, flag=twiss, full;
twiss, file=twiss_file.tws;
```

Note that if the lattice being converted is a transfer line —that is, a line that does not feature a periodic solution— you will need to specify the initial Twiss parameters. For example:

```
call, file="linac4.seq";
use, sequence=linac4;
select, flag=twiss, full;
twiss, betx=0.451, alfx=-3.599, bety=2.236, alfy=-10.419, file=linac4.tws;
```

3.2.3 Element misalignment

Elements can be misaligned by specifying their installation offsets when added to the Lattice or randomly scattering them. The following variants of the method `append` allow an element to be added to a Lattice with specific installation offsets:

```
L.append (element, dX, dY, dZ, reference = "entrance");
L.append (element, dX, dY, dZ, roll, pitch, yaw, reference = "entrance");
L.append_ref (element, dX, dY, dZ, reference = "entrance");
L.append_ref (element, dX, dY, dZ, roll, pitch, yaw, reference = "entrance");
```

The arguments are:

`element`	the element to be added	
`lattice`	the lattice to be added	
`dX`	the horizontal position of the new element	[m]
`dY`	the vertical position of the new element	[m]
`dZ`	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 definition: yaw, pitch, and roll. Note that no approximations are made, such as small angles or offsets.

Random misalignment

Elements can also be scattered randomly using the following Lattice methods:

```
L.scatter_elements(type, dX, dY, dZ, roll, pitch, yaw, reference='entrance');
L.scatter_elements(dX, dY, dZ, roll, pitch, yaw, reference='entrance');
```

The elements will be scattered according to normal distributions whose sigma are the parameters specified in these commands. The arguments are:

`type`	the type of element to be scattered; it can be one among:
	`'bpm'`
	`'sbend'`
	`'lattice'`
	`'absorber'`
	`'solenoid'`
	`'sextupole'`
	`'multipole'`
	`'corrector'`
	`'rf_elememt'`	

`dX`	the rms horizontal scattering	[mm]
`dY`	the rms vertical scattering	[mm]
`dZ`	the rms longitudinal scattering	[mm]
`roll`	the rms rotation angle around the Z axis	[mrad]
`pitch`	the rms rotation angle around the X axis	[mrad]
`yaw`	the rms rotation angle around the Y axis	[mrad]
`reference`	reference point for position and angles: it can be either `'entrance'`, `'center'`, or `'exit'`	

In the second form, where the element type to be scattered is not specified, all elements in the lattice will be scattered, including the elements in nested Lattices.

3.2.4 Tracking the beam

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

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

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

3.2.5 Accessing and modifying the elements

Once the elements have been added to a Lattice, they can be accessed in two ways: by index or by name.
In Octave:

```
L{1} % return the first element
L{'NAME'} % return all the elements called 'NAME'
```

In Python:

```
L[0] # return the first element
L['NAME'] # return all the elements called 'NAME'
```

In both Octave and Python, the string 'NAME' accepts wildcards. Note that the attributes of each element can be changed dynamically. For example,

```
% Define the elements
F = Quadrupole (0.2, +1); % focusing quad
D = Quadrupole (0.2, -1); % defocuding quad
O = Drift(1); % drift

% Define the lattice
L = Lattice();
L.append (F);
L.append (O);
L.append (D);
L.append (O);

% Modify the strength of the first quadrupole
L{1}.set_strength (2); % Now the first quadrupole has strength 2
```

3.3 Volume

To set up a Volume, it is sufficient to create an object of type `Volume`:

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

There are no input options.

3.3.1 Adding elements

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

```
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');
V.add_ref( ... );
```

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 element in the Volume	[m]
`Ypos`	the vertical position of the element in the Volume	[m]
`Zpos`	the longitudinal position of the element in the Volume	[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'`. This means that the specified position and angles denote the entrance, the exit, or the center of the element.	

The angles follow the Tait-Bryan definition of yaw, pitch and roll. No approximations such as "small angles" or "small offsets" are made. When a `Lattice` or `Volume` is added, its individual elements are added one by one.

As with `Lattice`, the methods `add(...)` add a *copy* of the new element to the `Volume`. If you want to add a *reference* to an element rather than a copy, i.e. in such a way that modifying the element after it has been added will directly affect the tracking in `Volume`, one can use the `add_ref(...)` methods.

3.3.2 Accessing and modifying the elements

Once the elements have been added to a Volume, they can be accessed in two ways: by index or by name.
In Octave:

```
V{1} % return the first element
V{'NAME'} % return all the elements called 'NAME'
```

In Python:

```
V[0] # return the first element
V['NAME'] # return all the elements called 'NAME'
```

In both Octave and Python, the string 'NAME' accepts wildcards. Like in a Lattice, each element can be changed dynamically. See the previous section about Lattice for an example.

Tracking the beam

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

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

The possible arguments are:

`B0`	the beam to be tracked, a `Bunch6dT`
`options`	a set of `TrackingOptions`. See the next subsection for details.

The tracking continues until the slowest particle has left the Volume. The return value, `B1`, is the `Bunch6dT` at that moment. See page 50 to learn more about the "End-of-tracking" condition.

When integrating the equations of motion, RF-Track distributes the particles among all available CPU threads and performs parallel tracking. When a collective effect is due to be considered, RF-Track must retrieve all particles from each thread, and only then can it compute the collective effect. This mechanism necessarily breaks the parallel tracking, even though the calculation of the collective effects is performed in parallel.

Breaking the parallelism of tracking certainly slows it down. For this reason, it is usually desirable to apply collective effects kicks at a larger time step than the step used to integrate the equations of motion. A convergence study is recommended to find the optimal compromise between the tracking speed and the accuracy of the results.

Similar slowdowns occur in the periodic evaluation of the transport table entries or while saving the beam to disk when using watchpoints.

Tracking Options

In Volume, RF-Track performs tracking by numerically integrating the equations of motion over time. This numerical integration is staggered: larger time steps are used for collective effects, while smaller steps are used to evolve the beam through the elements' electromagnetic fields accurately.

RF-Track allows the user to specify these time steps and choose the algorithm to integrate the equations of motion through the so-called `TrackingOptions`. A description of the `TrackingOptions` is available in Table 3.1.

Integration algorithms

Among the integration algorithms, '`leapfrog`' (which, more precisely, implements the Verlet integration method) is the fastest. The greater speed of Leapfrog comes at the expense of accuracy. Excellent accuracy at a reasonable computational cost is offered by '`rk2`', the recommended alternative to the leapfrog method. Until RF-Track version 2.4.2, `leapfrog` was the default. To improve accuracy, since version 2.5.0, the default is `rk2`.

As already mentioned, we recommend performing a convergence study to obtain stable and accurate results. This study should test different integration algorithms and time steps until a reasonable compromise between simulation speed and accuracy of the results is achieved.

Follows a list of available integration algorithms:

Table 3.1: A list of all the `TrackingOptions`

`odeint_algorithm`	The integration algorithm. See the following paragraphs for a list of the available algorithms. Default = '`rk2`'	[STRING]
`odeint_epsabs`	Absolute error tolerance ('`rk`' algorithms only). Default = 0.001	[REAL]
`odeint_epsrel`	Relative error tolerance ('`rk`' algorithms only). Default = 0.0	[REAL]
`dt_mm`	The integration step, the suffix `mm` stresses the units	[mm/c]
`t_max_mm`	Sets the max tracking time. Default = $+\infty$	[mm/c]
`t_min_mm`	Sets the min tracking time (used for backtracking). Default = $-\infty$	[mm/c]
`sc_dt_mm`	Apply a space-charge kick every `sc_dt_mm` mm/c	[mm/c]
`cfx_dt_mm`	Apply the other collective effects every `cfx_dt_mm` mm/c	[mm/c]
`tt_dt_mm`	Add an entry of the Transport table every `tt_dt_mm` mm/c	[mm/c]
`tt_select`	Select what particles should be included in the transport table computation. It can be one among: '`all`', '`active`', '`all_in_volume`', or '`active_in_volume`'. '`all`' also includes particles that haven't yet come into existence, for instance, particles at the cathode in a photoinjector simulation. Default = '`all`'	[STRING]
`emission_nsteps`	If `sc_dt_mm` is set, applies `emission_nsteps` space-charge kicks during bunch emission Default = 10	[INTEGER]
`emission_range`	Maintain an emission tracking mode for as much time as `emission_range` times the emission time Default = 10.0	[REAL]
`wp_dt_mm`	"wp" stands for "watch point". Save the beam on disk every `wp_dt_mm` mm/c	[mm/c]
`wp_basename`	Base name for the files on disk. Default = "`watch_beam`". The files will have names: "`watch_beam.XXXXXXX.txt`" and contain 10 columns:	[STRING]
	1. X horizontal position	[mm]
	2. P_x horizontal momentum	[MeV/c]
	3. Y vertical position	[mm]
	4. P_y vertical momentum	[MeV/c]
	5. Z longitudinal position	[mm]
	6. P_z longitudinal momentum	[MeV/c]
	7. m mass	[MeV/c^2]
	8. Q single-particle charge	[e]
	9. N number of real particles per macroparticle	[REAL]
	10. particle id	[INTEGER]
`wp_gzip`	Automatically compress the watch-point files. Default = false	[BOOLEAN]
`verbosity`	Verbosity level during tracking. 0 = silent; 1 = talkative; 2 = more talkative. Default = 0.	[INTEGER]

Algorithm	Description
`'analytic'`	This algorithm solves analytically the equations of motion assuming a *locally constant* field. In truly constant fields, this algorithm gives a solution that is *exact*. In non-constant fields, its accuracy depends on the size of the integration step. This algorithm is symplectic only in truly constant fields.
`'leapfrog'`	This algorithm updates positions and momenta at different interleaved time points. It's a second-order symplectic algorithm.
`'rk2'`	Explicit embedded Runge-Kutta (2, 3) method.
`'rk4'`	Explicit 4th order (classical) Runge-Kutta. Error estimation is performed using the step-doubling method.
`'rkf45'`	Explicit embedded Runge-Kutta-Fehlberg (4, 5) method. This method is a good general-purpose integrator.
`'rkck'`	Explicit embedded Runge-Kutta Cash-Karp (4, 5) method.
`'k8pd'`	Explicit embedded Runge-Kutta Prince-Dormand (8, 9) method.
`'msadams'`	A variable-coefficient linear multistep Adams method in Nordsieck form. This stepper uses explicit Adams-Bashforth (predictor) and implicit Adams-Moulton (corrector) methods in $P(EC)^m$ functional iteration mode. Method order varies dynamically between 1 and 12.

The first three are sufficient in most cases; however, we recommend conducting convergence studies and speed tests to select the most suitable algorithm and step size for your specific case.

Time limits of the integration

In Volume, tracking continues until the absolute time, t, is included between `t_min_mm` and `t_max_mm`. The default value for `t_max_mm` is $+\infty$, for `t_min_mm` is $-\infty$, which means that tracking (or backtracking) continues until all particles have left the Volume (or have reached their creation time, if backtracking). When `t_max_mm` or `t_min_mm` is set by the user to a finite number, tracking will stop when t reaches it. See the next paragraphs for more details on the stop criterion in Volume.

Setting the tracking options

A Volume itself is an instance of the structure `TrackingOptions`; this allows one to set the volume's tracking options in this possibly clearer way:

```
% Create Volume
V = Volume();

% Set the tracking options
V.odeint_algorithm = 'rkf45';
V.dt_mm = 0.1; % mm/c
V.verbosity = 2;

% Add a few elements
V.add (ELEMENT1, 0, 0, 0);
V.add (ELEMENT2, 0, 0, 0);
V.add (ELEMENT3, 0, 0, 0);
```

```
% Perform tracking
V.track(B0);
```

The planes s_0 and s_1

Two planes, called s_0 and s_1, orthogonal to the longitudinal axis Z, define the longitudinal extension of a Volume. When one adds elements to a Volume, RF-Track automatically moves s_0 backwards and s_1 forward to accommodate the new element.

The planes s_0 or s_1 can have any position and orientation in space. One can arbitrarily place these planes to achieve specific configurations, or embed a Volume into a Lattice (see next subsection), in multiple ways:

```
% Setting S0
V.set_s0 (z)
V.set_s0 (P0, t);
V.set_s0 (x, y, z, roll=0, pitch=0, yaw=0);

% Setting S1
V.set_s1 (z);
V.set_s1 (P0, t);
V.set_s1 (x, y, z, roll=0, pitch=0, yaw=0);

% Setting one w.r.t. the other
V.set_s0_from_s1 (P0, l);
V.set_s1_from_s0 (P0, l);
```

In these commands, the arguments are:

`x, y, z`	coordinates	[m]
`t`	the tracking time	[mm/c]
`l`	a distance	[m]
`P0`	a reference particle	[`Bunch6dT`]
`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]

Notice that adding new elements to a Volume automatically updates s_1. So, if you add new elements after `set_s1()`, s_1 will likely be changed to include the new elements.

End-of-tracking condition

Tracking in Volume continues as long as particles exist between these two planes, or until the maximum tracking time (tracking option `t_max_mm`) is reached. By default, the maximum tracking time `t_max_mm` is set to $+\infty$ (`Inf` in Octave, or `numpy.inf` in Python). Beware that if a particle gets trapped in the Volume, tracking will continue indefinitely unless the max tracking time is reached `t_max_mm`.

3.3.3 Volume as a Lattice element

Once a Volume has been initialized, adding all its elements and setting its planes s_0 and s_1, one can insert it into a Lattice and treat it as a standard Lattice element. Figure 3.1 gives a pictorial repre-

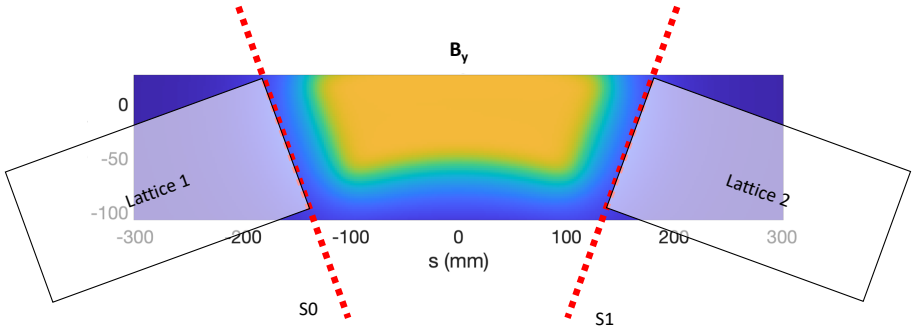


Figure 3.1: A Volume can be sandwiched between two Lattices.

sentation of a potential simulation scenario. An illustrative example script follows:

```
% A reference particle placed in (0, 0, 0) with Pz = 100 MeV/c
P0 = Bunch6dT (mass, 0.0, +1, [ 0 0 0 0 0 100 ]);

% A Volume containing the field map in the figure
Dipole = Volume();
Dipole.dt_mm = 1.0;
Dipole.odeint_algorithm = 'rk2';
Dipole.add (DIPOLE_MAP, 0, 0, 0, 'center');
Dipole.set_00 (P0, -150); % track backward P0 by 150 mm/c to set s0
Dipole.set_s1 (P0, +150); % track forward P0 by 150 mm/c to set s1
Dipole.set_tt_nsteps(20);

% A Lattice
L = Lattice();
L.append (Lattice1)
L.append (Dipole)
L.append (Lattice2)
```

When tracking through `L` is performed, the particles coming from `Lattice1` are distributed over the plane s_0 of `Dipole`, tracked through the Volume `Dipole`, and collected at s_1 to form a `Bunch6d` suitable to continue tracking through `Lattice2`.

3.3.4 Transport table and Screens

For a detailed tracking of the transport table quantities, a Volume can be sliced into slices using the method `set_tt_nsteps()` like any Lattice element. Continuing with the example shown in Fig. 3.1, and described conceptually in the previous script, the line

```
Dipole.set_tt_nsteps(20);
```

tells RF-Track that we want to track the lattice transport table in 20 steps. To calculate the Lattice transport table, RF-Track will then distribute 20 screens orthogonal to the curved trajectory between s_0 and s_1.

3.4 Particle losses

Particles that are lost during tracking can be retrieved from the Lattice or Volume in which they were lost. This can be done using the following methods:

```
M = V.get_lost_particles();
M = L.get_lost_particles();
```

Both methods return an 11-column matrix with the information on where and when each particle was lost. The information is given in the reference frame of the element itself.

In the case of Lattice, the 11 columns are:

1. `X` in mm
2. `XP` in mrad
3. `Y` in mm
4. `YP` in mrad
5. `T` in mm/c
6. `P` in MeV/c
7. `S` in m, the longitudinal position at which the particle was lost
8. `MASS`, in MeV/c^2
9. `Q` in e
10. `N` the macro-particle charge
11. `ID` the particle ID

In the case of Volume, the 11 columns are:

1. `X` in mm
2. `Px` in MeV/c
3. `Y` in mm
4. `Py` in MeV/c
5. `Z` in mm
6. `Pz` in MeV/c
7. `T` in mm/c, the time at which the particle was lost
8. `MASS` in MeV/c^2
9. `Q` in e
10. `N` the macro-particle charge
11. `ID` the particle ID

3.5 Synchronization of time-dependent elements with the beam

Time-dependent elements -such as RF accelerator structures (whether field maps, travelling-wave, or standing-wave elements), `Screens`, and the `LaserBeam` element used in Compton scattering simulation— require knowledge of the absolute arrival time of the beam in order to operate correctly

and synchronously with the beam. The method `set_t0()`, available for all time-dependent elements, allows the user to define this arrival time manually.

While determining the exact arrival time may seem straightforward for ultra-relativistic beams, where $\Delta z \approx c\,\Delta t$, it becomes considerably more complex for long beamlines or particles traveling at subluminal velocities ($v < c$), such as heavy ions, protons, or low-energy electrons. To simplify this task, RF-Track provides the `autophase()` method, common to both `Lattice` and `Volume`, which automatically sets the correct synchronization parameters for all time-dependent elements.

3.5.1 Autophasing

The `autophase()` method, available for both `Lattice` and `Volume`, synchronizes each time-dependent beamline element with the beam by propagating a test bunch through the structure and recording its arrival time, element by element.

Below is an example where `P0`, a `Bunch6d` representing an electron bunch with a single reference particle with total charge of 100 pC and average momentum P_{ref} (called `Pref` in the example), is used for autophasing a lattice `L`:

```
% Create a reference particle
P0 = Bunch6d(electronmass, 100*pC, -1, [ 0 0 0 0 0 Pref ]);

% Autophase the lattice
Pfinal = L.autophase(P0);
```

In this example, `autophase()` is given a single reference particle as input. If a full bunch is provided instead, the average particle will be used for autophasing.

After calling `autophase()`, all time-dependent elements are synchronized with the actual arrival time of `P0` at each location, and no manual `t0` assignment is necessary. Subsequent uses of the lattice `L` will retain these phases and arrival times. This function returns the final momentum of the reference particle after synchronization.

For RF elements, `autophase()` not only sets the arrival time but also determines the synchronous phase of the structure. Specifically, it adjusts the RF phase so that an input user phase value of `phid = 0` (which is the default, if unspecified) corresponds to on-crest acceleration. This feature relieves the user of the tedious task of manually determining the correct RF phase for each structure[1].

Three important remarks

Remark 1: `autophase()` should be considered an essential part of the initialization process for a `Lattice` or `Volume`. It must be called before any tracking is performed.

Remark 2: Any time-dependent element whose reference time is set using `set_t0()` *before* being added to a `Lattice` or `Volume` will be excluded from the autophasing process and will retain its manually defined reference time.

Remark 3: If the user does not explicitly call `autophase()`, RF-Track will automatically invoke it during the first call to the `track()` method.

[1] In fact, standard electromagnetic simulation codes provide field maps with an arbitrary RF phase.

3.5.2 Automatic setting of magnetic strengths

In RF-Track, magnetic fields are typically defined using either the magnetic strength or the gradient. This is the case, for instance, with the `Quadrupole` and `Multipole` elements discussed in the next chapter. However, when designing accelerator structures such as linacs composed of FODO cells interleaved with RF cavities, it is often challenging to specify the exact magnet gradients. This is because the magnetic strength depends on the beam momentum, which typically varies along the linac and is not known a priori. In such situations, it is more practical to define the magnets using their *normalised strengths* — quantities that are independent of energy and determined solely by the optical properties of the lattice. For example, the quadrupole's parameter k_1 [1/m^{-2}] describes the *normalised strength* of a quadrupole.

When k_1 is provided, RF-Track can defer the computation of the actual magnetic gradient until the beam energy is known. This is achieved by setting the reference momentum to `NaN` when creating the magnet. For example:

```
% We know the normalized focusing strength, k1,
% but the beam rigidity is not yet defined.

k1 = 0.12; % m^-2, normalised strength
L = 0.1; % m, quadrupole length
P_Q = NaN; % Beam momentum is unknown at this stage

% Define a quadrupole with deferred gradient computation
Q = Quadrupole (L, P_Q, k1);
```

The quadrupole gradient will then be automatically computed and assigned during the call to `autophase()`. If the quadrupole length is set to zero, then `k1` is interpreted as the integrated strength $k_1 L$ [1/m], allowing the simulation of thin-lens quadrupoles.

3.6 Backtracking

RF-Track supports beam backtracking. During backtracking, element misalignments and all deterministic collective effects are fully accounted for. This includes effects such as space charge or wakefields, but excludes stochastic processes like multiple Coulomb scattering or quantum incoherent synchrotron radiation emission.

Backtracking can be performed using the `btrack()` method, which is available for both `Lattice` and `Volume` objects. For example:

```
% B1 is the desired final phase-space distribution
B1 = Bunch6d(...);

% Backtrack to find B0, the initial distribution that evolves into B1
B0 = L.btrack(B1);
```

This feature is particularly useful when the goal is to determine the initial beam distribution that yields a specified final distribution at the end of a beamline.

4. Beamline Elements

4.1 Introduction

RF-Track offers a comprehensive set of elements, ranging from conventional matrix-based quadrupoles and sector bends to more sophisticated options, such as complex field maps and analytic fields that permeate the entire 3D space. This chapter gives a short description of the constructors for each element.

In some elements, specific attributes cannot be directly set via the constructor but can be set using appropriate "set" methods. The most relevant methods are given for each element type. The user can access a list of all methods using the interactive prompts of Octave and Python.

4.1.1 Methods available to all elements

Many practical methods are available to all elements to achieve specific setups.

Setting the element name

The element can have an optional name that can be used to look up the elements in the tracking environments. Two methods allow to set and get an element's name:

```
E.set_name (STRING);
E.get_name ();
```

Setting the aperture

All elements are equipped with an aperture, which, by default, is not considered. The aperture can be set and inquired using the following set of methods:

```
E.set_aperture_x (Rx); % m
E.set_aperture_y (Ry); % m
E.set_aperture_shape (SHAPE); % m
E.set_aperture (Rx, Ry, SHAPE); % m
E.get_aperture_x (); % return m
```

```
E.get_aperture_y (); % return m
E.get_aperture_shape (); % return a string
```

The arguments are:

Rx, Ry	the aperture radii	[m]
SHAPE	a string among 'none', 'rectangular', and 'circular'	[STRING]

The 'circular' aperture becomes elliptical when Rx $\neq$ Ry. The aperture is checked at each integration step.

Adding constant magnetic and electric fields

Available to most element types, two methods allow embedding an element in a constant magnetic or electric field with any orientation in space:

```
E.set_static_Bfield (Bx, By, Bz); % T
E.set_static_Efield (Ex, Ey, Ez); % V/m
```

Where E is the instance of an element.

Activating collective effects

Collective effects can be added to *any* element using the following method:

```
E.add_collective_effect (CFX);
```

See the dedicated chapter for a list of the collective effects implemented in RF-Track.

4.1.2 Inquiring and plotting the electromagnetic field

Each RF-Track element can be queried for the electromagnetic field it represents at any point. The method `get_field()` can be used to do this. See the following example:

```
Q = Quadrupole (1.0, 100.0); % Or any other element

% Inquire the % field at point (x,y,z,t)
x = 0;  % mm
y = 10; % mm
z = 20; % mm
t = 30; % mm/c

[E,B] = Q.get_field (x, y, z, t); % Returns E in [V/m] and B in [T].
```

The input arguments of `get_field()` can be either scalars or one-dimensional vectors to query a sequence of points at once. E and B are three-element vectors containing E_x, E_y, E_z and B_x, B_y, B_z respectively in the case of a single point, or three-column matrices with as many rows as there are points queried.

Compound elements such as `Lattice` and `Volume` can also be queried for the field. This makes it possible, for example, to plot the total electromagnetic field of all elements in a given environment at any point.

4.1.3 Tracking the beam properties

Both `Lattice()` and `Volume()` offer the possibility of storing average beam quantities, such as beam size, emittance, energy spread, dispersion, etc., during tracking into a "transport table". After

tracking, such a transport table can be retrieved using the method `get_transport_table()` available to both `Lattice` and `Volume`.

Like `get_phase_space()` for the beam, `get_transport_table()` allows the user to inquire about specific quantities.

In `Lattice()`, the user can choose the number of points at which the phase space is sampled using the element's method `set_tt_nsteps()`. For example:

```
D = Drift (1); % 1-meter-long drift
D.set_tt_nsteps (100);
```

which will track the beam through the drift `D` in 100 steps, sampling the phase space in that many points. Table 4.1 lists all the accepted identifiers.

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, in mm/c, between two consecutive samplings. Table 4.2 lists all the accepted identifiers.

Table 4.1: List of identifiers accepted by `Lattice::get_transport_table()`

`%S`	longitudinal position	[m]
`%mean_x`	average horizontal position	[mm]
`%mean_y`	average vertical position	[mm]
`%mean_t`	average arrival time	[mm/c]
`%mean_xp`	average horizontal angle, x'	[mrad]
`%mean_yp`	average vertical angle, y'	[mrad]
`%mean_Px`	average horizontal momentum	[MeV/c]
`%mean_Py`	average vertical momentum	[MeV/c]
`%mean_Pz`	average longitudinal momentum	[MeV/c]
`%mean_P`	average total momentum	[MeV/c]
`%mean_K`	average kinetic energy	[MeV]
`%mean_E`	average total energy	[MeV]
`%emitt_x`	normalized horizontal emittance	[mm.mrad]
`%emitt_y`	normalized vertical emittance	[mm.mrad]
`%emitt_z`	normalized longitudinal emittance	[mm.permille]
`%emitt_4d`	normalized transverse emittance	[mm.mrad]
`%emitt_6d`	normalized 6d emittance	[mm.mrad]
`%disp_x`	horizontal dispersion function	[m]
`%disp_y`	vertical dispersion function	[m]
`%disp_z`	longitudinal dispersion function	[m]
`%disp_px`	horizontal dispersion-prime function	[rad]
`%disp_py`	vertical dispersion-prime function	[rad]
`%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_t`	longitudinal spread	[mm/c]
`%sigma_px`	normalized horizontal momentum spread	[mrad]
`%sigma_py`	normalized vertical momentum spread	[mrad]
`%sigma_pt`	normalized energy spread σ_E/P_{ref}	[permille]
`%sigma_P`	total momentum spread	[MeV/c]
`%rmax`	transverse envelope (all particles)	[mm]
`%rmax99.9`	transverse envelope (99.9% of the particles)	[mm]
`%rmax99`	transverse envelope (99% of the particles)	[mm]
`%rmax90`	transverse envelope (90% of the particles)	[mm]
`%N`	transmission, expressed in number of real particles	[#]

Table 4.2: List of identifiers accepted by `Volume::get_transport_table()`

`%t`	time	[mm/c]
`%mean_X`	average horizontal position	[mm]
`%mean_Y`	average vertical position	[mm]
`%mean_Z`	average longitudinal coordinate	[mm]
`%mean_K`	average kinetic energy	[MeV]
`%mean_E`	average total energy	[MeV]
`%mean_P`	average total momentum	[MeV/c]
`%mean_Px`	average horizontal momentum	[MeV/c]
`%mean_Py`	average vertical momentum	[MeV/c]
`%mean_Pz`	average longitudinal momentum	[MeV/c]
`%emitt_x`	normalized horizontal emittance	[mm.mrad]
`%emitt_y`	normalized vertical emittance	[mm.mrad]
`%emitt_z`	normalized longitudinal emittance	[mm.permille]
`%emitt_4d`	normalized transverse emittance	[mm.mrad]
`%emitt_6d`	normalized 6d emittance	[mm.mrad]
`%disp_x`	horizontal dispersion function	[m]
`%disp_y`	vertical dispersion function	[m]
`%disp_z`	longitudinal dispersion function	[m]
`%disp_px`	horizontal dispersion-prime function	[rad]
`%disp_py`	vertical dispersion-prime function	[rad]
`%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]
`%rmax`	transverse envelope (all particles)	[mm]
`%rmax99.9`	transverse envelope (99.9% of the particles)	[mm]
`%rmax99`	transverse envelope (99% of the particles)	[mm]
`%rmax90`	transverse envelope (90% of the particles)	[mm]
`%N`	transmission, expressed in number of real particles	[#]

4.2 Matrix-based elements

All matrix-based elements are, by definition, symplectic.

4.2.1 Drift

An empty region of space.

Constructor

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

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

Set methods

```
D.set_static_Bfield (Bx, By, Bz); % T
D.set_static_Efield (Ex, Ey, Ez); % V/m
D.add_collective_effect  (CFX );
```

These first two methods, used on a `Drift`, allow the simulation of localised regions of space where a static electromagnetic field is present. For example, one could use this to simulate a static accelerating field.

The method `add_collective_effect()` allows having localised regions of space where specific collective effects act on the beam. In combination with the `MultipleCoulombScattering` effect, one can, for instance, simulate regions in the air or other materials.

4.2.2 Quadrupole

A matrix-based quadrupole magnet.

Constructors

```
Q = Quadrupole (L=0, strength=0);
Q = Quadrupole (L, P_Q, k1);
```

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]
	If `NaN`, RF-Track will set it a tracking time using `autophase()`	
`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 $B\rho$:

$$S = \left(\frac{P/q}{\mathrm{MV}/c}\right)\left(\frac{k_1}{1/\mathrm{m}^2}\right)\left(\frac{L}{\mathrm{m}}\right) \qquad [\mathrm{MV}/c/\mathrm{m}]$$

$$k_1 = \left(\frac{G}{\mathrm{T/m}}\right)\left(\frac{\mathrm{T\,m}}{B\rho}\right) \qquad [1/\mathrm{m}^2]$$

Get methods

```
k1 = Q.get_K1(P_Q); % return 1/m^2
k1L = Q.get_K1L(P_Q); % return 1/m
G = Q.get_gradient(); % return T/m
strength = Q.get_strength(); % return MV/c/m
```

Set methods

```
Q.set_K1(P_Q, k1);
Q.set_K1L(P_Q, k1L);
Q.set_length(L);
Q.set_gradient(G);
Q.set_strength(strength);
```

Examples

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

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

Or, given the quadrupole gradient, G:

```
Lquad = 0.2; % m, quadrupole length
G = 1.2; % T/m, quadrupole gradient
Q = Quadrupole(Lquad);
Q.set_gradient(G);
```

4.2.3 Sector bending dipole

`SBend` is a sector-bending magnet on a curved reference system. Like MAD-X, a positive bend angle represents a bend to the right, i.e., towards negative x values. Figure 4.1 will help understand this convention.

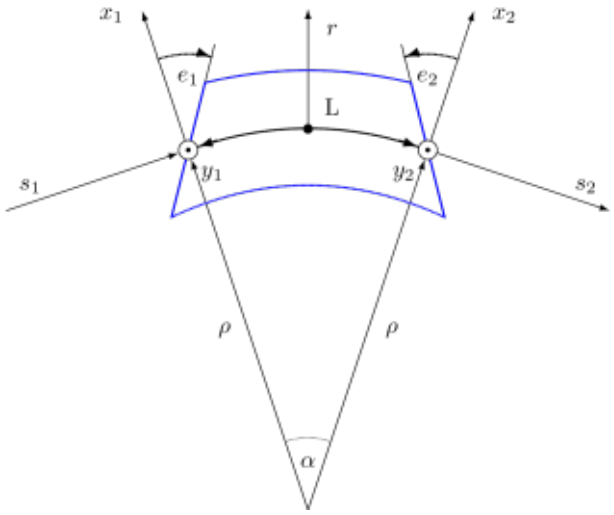


Figure 4.1: The reference system for a sector-bending magnet; the signs of pole-face rotations are positive, as shown. Figure retrieved from the MAD-X documentation.

Constructors

```
S = SBend (L, angle, P_Q, E1=0, E2=0, K1=0);
S = SBend (L=0);
```

The arguments are:

`L`	the bending magnet length	[m]
`angle`	the bending angle	[rad]
`P_Q`	the beam's magnetic rigidity, P/q	[MV/c]
`E1`, `E2`	the entrance and the exit angles	[rad]

Get methods

```
S.get_E1(); % the entrance edge angle in rad
S.get_E2(); % the exit edge angle in rad
S.get_E1d(); % the entrance edge angle in deg
S.get_E2d(); % the exit edge angle in deg
S.get_angle(); % the bending angle in rad
S.get_angled(); % the bending angle in deg
S.get_h(); % 1/m, the inverse of the bending radius
S.get_rho(); % m, the bending radius
S.get_K1(); % 1/m**2, normalzied quadrupole strength
S.get_Bfield(); % T, return the By field
```

Set methods

The obvious ones, plus:

```
S.set_h(H); % 1/m, set the curvature of the reference system
S.set_K1(K1); % 1/m^2, normalzied quadrupole strength
S.set_hgap(HGAP); % m, the half gap of the magnet
S.set_fint(FINT); % the fringe field integral
```

The parameter `FINT` follows the same definitions as in MAD-X:

$$\texttt{FINT} = \int_{-\infty}^{\infty} \frac{B_y(s)\,(B_0 - B_y(s))}{g\,B_0^2}\,\mathrm{d}s,$$

with $g = 2\,\texttt{HGAP}$.

The default value `FINT` of zero corresponds to the hard-edge approximation, i.e., a rectangular field distribution. For other approximations, one can refer to the following pre-computed values of `FINT`:

Linear field drop-off	$1/6$
Clamped “Rogowski” fringing field	0.4
Unclamped “Rogowski” fringing field	0.7
“Square-edged” non-saturating magnet	0.45

4.2.4 Rectangular bending dipole

`RBend` is a rectangular bend, a bending magnet whose entrance and exit windows are parallel. Its reference system is curved by default, and its length is intended to be the straight-line distance between the entry and exit points.

Constructor

```
R = RBend (L, angle, P_Q, E1=0, E2=0);
```

The arguments are:

`L`	the rectangular bending magnet length	[m]
`angle`	the bending angle	[rad]
`P_Q`	the beam's magnetic rigidity, P/q	[MV/c]
`E1`, `E2`	the entrance and the exit angles	[rad]

As the `RBend` is internally implemented as a `SBend` with appropriate entrance and exit edge angles, this element features the same "Set" and "Get" methods of a sector bend.

4.2.5 Corrector

The element `Corrector` creates a magnetic steerer for orbit correction.

Constructors

```
C = Corrector (L);
C = Corrector (L, Kx, Ky);
```

The input arguments describing the Corrector are:

L	the corrector length	[m]
Kx, Ky	the integrated horizontal and vertical corrector strengths	[T.mm]

4.3 Special elements

In RF-Track, special elements represent components or devices that cannot be implemented using simple linear transfer matrices but require numerical integration in non-linear fields. These include coils, solenoids, various RF fields, undulators, etc.

4.3.1 Coil

The element `Coil` creates the magnetic field generated by an electromagnetic coil.

Constructors

```
C = Coil (L, B0, R );
```

The input arguments describing the coil are:

`L`	the element length	[m]
`B0`	the peak on-axis field	[T]
`R`	the coil radius.	[m]

The coil is placed in the middle of the specified length `L`. When a `Coil` is placed in a Volume, its field permeates the whole 3D space. Warning: When used in Lattice, the field exists only within the extent of the specified element length, with the coil placed in the middle.

See also the element `Solenoid`.

4.3.2 Solenoid

The element `Solenoid` allows the insertion of a Solenoid magnet into a Lattice or a Volume. When a `Solenoid` is inserted into a Lattice, RF-Track will treat it as a matrix-based element. When `Solenoid` is inserted into a Volume, RF-Track computes the 3D magnetic field of the magnet using analytic equations. In this case, the field includes realistic fringe regions and permeates the whole 3D space, overlapping with other fields in the same Volume.

Constructor

```
S = Solenoid (L=0, B0=0, R=0);
S = Solenoid (L, B0, Rmin, Rmax, nsheets);
```

The arguments are:

`L`	the length of the solenoid	[m]
`B0`	the on-axis peak field,	[T]
`R`	the solenoid radius	[m]
`Rmin`	inner radius	[m]
`Rmax`	outer radius	[m]
`nsheets`	number of current sheets	[INTEGER]

In both Lattice and Volume, the argument radius, `R`, sets the aperture radius of the magnet. In Volume, its value is also used to compute the 3D field. Figure 4.2 shows the Solenoid field computed by RF-Track.

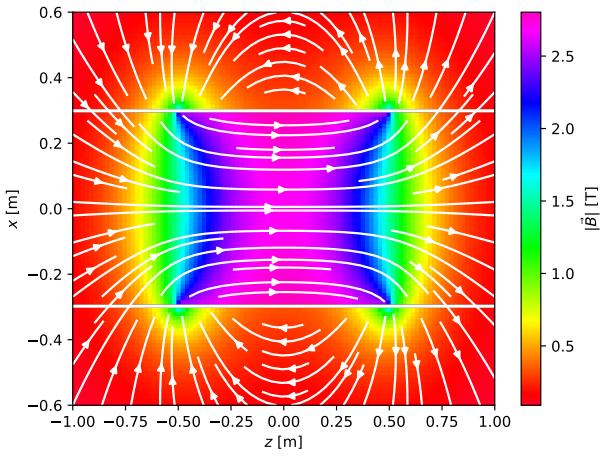


Figure 4.2: The Solenoid field computed by RF-Track.

4.3.3 Undulator

The element `Undulator` allows the insertion of a planar Undulator into a Lattice or a Volume. When particles travel through an `Undulator`, RF-Track computes the 3D magnetic field of the magnet using analytic equations and integrates the equations of motion using numerical integration.

Constructor

```
U = Undulator (lperiod, K, nperiods, kx2=0);
```

The arguments are:

`lperiod`	the length of an undulator period	[m]
`K`	the undulator K parameter in x direction	[-]
`nperiods`	the number of periods	[INTEGER]
`kx2`	the curvature of the pole surface, k_x^2 (default 0)	[1/m^2]

When `kx2` > 0, the poles are bent inwards; when `kx2` < 0, the poles are bent outwards. See Figure 4.3.

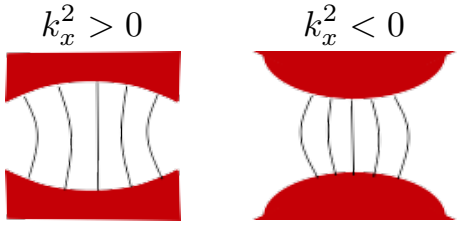


Figure 4.3: The parameter `kx2` describes the shape of the undulator's poles.

4.3.4 Transfer Line

The `TransferLine` element can be used to transport a beam through an entire beamline without having to specify each element individually, only by giving pairs of Twiss parameters. In the transverse plane, the element tracks the beam between each pair of Twiss parameters $\{\beta,\alpha\}_1$ and $\{\beta,\alpha\}_2$ using a transfer matrix in Twiss form:

$$M_{1\to 2} = \begin{pmatrix} \sqrt{\frac{\beta_2}{\beta_1}}(\cos\mu + \alpha_1 \sin\mu) & \sqrt{\beta_1\beta_2}\sin\mu \\ \frac{(\alpha_1-\alpha_2)\cos\mu-(1+\alpha_1\alpha_2)\sin\mu}{\sqrt{\beta_1\beta_2}} & \sqrt{\frac{\beta_1}{\beta_2}}(\cos\mu - \alpha_2\sin\mu) \end{pmatrix}$$

with phase advance μ. If horizontal and vertical chromaticities are provided, the phase advance in both planes is adjusted accordingly. In the longitudinal plane, a drift is applied with length $L = L_0\,(1+\alpha_C\delta)$, where L_0 is the length of the element's section, α_C is the momentum compaction factor, and δ is the relative momentum difference.
This element has three modes of operation:

- If one gives two sets of Twiss parameters, RF-Track will use a Twiss matrix.
- If one gives a Twiss table or a Twiss file from MAD-X, RF-Track will track through each consecutive pair of Twiss parameters using the Twiss transfer matrix.
- If one gives just one set of Twiss parameters, RF-Track will match *any* bunch being tracked to the desired set of Twiss parameters.

Like any other Lattice elements, a `TransferLine` can be segmented into steps, and collective effects can be applied through the TransferLine.

Constructors

```
T = TransferLine ("twiss_file.dat", Pref);
T = TransferLine (twiss_matrix, Pref, DQx, DQy, momentum_compaction);
T = TransferLine (twiss_matrix, Pref);
```

The arguments are:

`"twiss_file.dat"`	a Twiss file from MAD-X	
`twiss_matrix`	a matrix containing the Twiss parameters.	
	This matrix can be either a 7-column matrix, where each row contains: $[\,S,\ \beta_x,\ \alpha_x,\ \mu_x,\ \beta_y,\ \alpha_y,\ \mu_y\,]$	
	or an 11-column matrix, where each row contains: $[\,S,\ \beta_x,\ \alpha_x,\ \mu_x,\ \beta_y,\ \alpha_y,\ \mu_y,\ D_x, D_{px}, D_y, D_{py}\,]$	
	$S, \beta_x, \beta_y, D_x, D_y$ are expressed in	[m]
	μ_x, μ_y are expressed in	$[2\pi]$
`Pref`	the reference momentum	[MeV/c]
`DQx`, `DQy`	the horizontal and vertical chromaticities	$[2\pi]$
`momentum_compaction`	the momentum compaction	

All the input quantities follow their respective MAD-X definitions.

Note: This element works in `Lattice` only.

Example of matching section

We propose the example of a matching section implemented as a `TransferLine`, matching a set of Twiss parameters $[\ \beta_{0,x},\ \alpha_{0,x},\ \beta_{0,y},\ \alpha_{0,y}\ ]$ into $[\ \beta_{1,x},\ \alpha_{1,x},\ \beta_{1,y},\ \alpha_{1,y}\ ]$, with phase advance μ_x and μ_y over a length L:

```
% Input data
Pref = 100; % MeV/c, reference momentum
L = 5; % m, length of the matching section

% Initial Twiss parameters
b0_x = 1; % m, horizontal beta function
b0_y = 2; % m, vertical beta function
a0_x = a0_y = 0; % alpha

% Final Twiss parameters
b1_x = 5; % m, horizontal beta function
b1_y = 10; % m, vertical beta function
a1_x = a1_y = 0; % alpha

% Phase advance between initial and final
mu_x = pi/2; % phase advance in x
mu_y = pi/3; % phase advance in y

% Input 2-row matrix for the constructor
T = [ 0, b0_x, a0_x, 0.0,  b0_x, a0_y, 0.0;    % 1st row, initial Twiss
      L, b1_x, a1_x, mu_x, b1_x, a1_y, mu_y ]; % 2nd row, final Twiss

% Transfer matrix for Lattice
M = TransferLine(T, Pref);
```

4.3.5 Travelling-wave structure

The element `TW_Structure` allows the simulation of the TM_{01n} modes in a travelling-wave (TW) structure using an analytic description of the field, as in the equations below. A TW structure is a metallic enclosure where electromagnetic fields travelling with a specific phase advance along the structure can be excited. Figure 4.4 shows a model example of a travelling-wave structure.

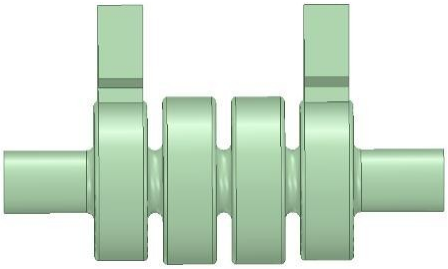


Figure 4.4: A model of a metallic travelling-wave structure.

Table 4.3: Fourier series expansions of the TM_{01n} electromagnetic fields in a TW structure

Quantity	Value
k_0	$\frac{\Delta\varphi}{L}$
k_n	$k_0 + \frac{2n\pi}{L}$
β_n	$\frac{\omega}{ck_n}$
q_n	$\sqrt{\left\|\left(\frac{\omega}{c}\right)^2 - (k_n)^2\right\|}$
$E_z(r,z,t)$	$\sum_{n=-\infty}^{\infty} a_n \sin[(\omega t+\phi_0) - k_n z] \times \begin{cases} J_0(q_n r) & \lvert\beta_n\rvert \geq 1 \\ I_0(q_n r) & \lvert\beta_n\rvert < 1 \end{cases}$
$E_r(r,z,t)$	$\sum_{n=-\infty}^{\infty} \frac{a_n k_n}{q_n} \cos[(\omega t+\phi_0) - k_n z] \times \begin{cases} J_1(q_n r) & \lvert\beta_n\rvert \geq 1 \\ I_1(q_n r) & \lvert\beta_n\rvert < 1 \end{cases}$
$B_\theta(r,z,t)$	$\sum_{n=-\infty}^{\infty} \frac{a_n (q_n^2+k_n^2)}{\omega q_n} \cos[(\omega t+\phi_0) - k_n z] \times \begin{cases} J_1(q_n r) & \lvert\beta_n\rvert \geq 1 \\ I_1(q_n r) & \lvert\beta_n\rvert < 1 \end{cases}$

For such structures, if L and $\Delta\varphi$ indicate the length and phase advance of one cavity cell, then the Fourier series expansions of the excited TM_{01n} modes in the cell are given by the expressions presented in Table 4.3.

Constructors

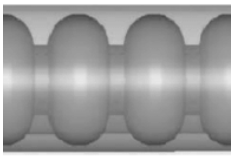
```
T = TW_Structure ([ an, ... ], n, freq, ph_advance, number_of_cells);
T = TW_Structure ();
```

The arguments are:

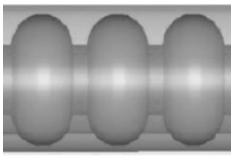
`[an, ... ]`	a vector with the Fourier coefficients (see Table 4.3)	[V/m]
`n`	the index of the first coefficient	[INTEGER]
`freq`	the rf frequency	[Hz]
`ph_advance`	the structure's phase advance per cell	[rad]
`number_of_cells`	the number of cells in the structure.	[REAL]

A positive number of cells indicates that the structure starts from the middle of the cell. A negative number indicates that the structure starts from the beginning of a cell. Follows an example with three cells:

- `number_of_cells = 3`



- `number_of_cells = -3`



Example of travelling-wave structure

We present an example of an ideal travelling-wave structure with three cells having a phase advance of $2\pi/3$, operating at 3 GHz with a gradient of 10 MV/m.

```
a0 = 10e6; % V/m, gradient
freq = 3e9; % Hz
ph_adv = 2*pi/3; % rad, phase advance
n_cells = 3; % number of cells
n = 0; % TM 01n mode

TW = TW_Structure(a0, n, freq, ph_adv, n_cells);
```

Figure 4.5 presents the longitudinal field profile when one changes the parameter `n` from 0 to 3 (top to bottom).

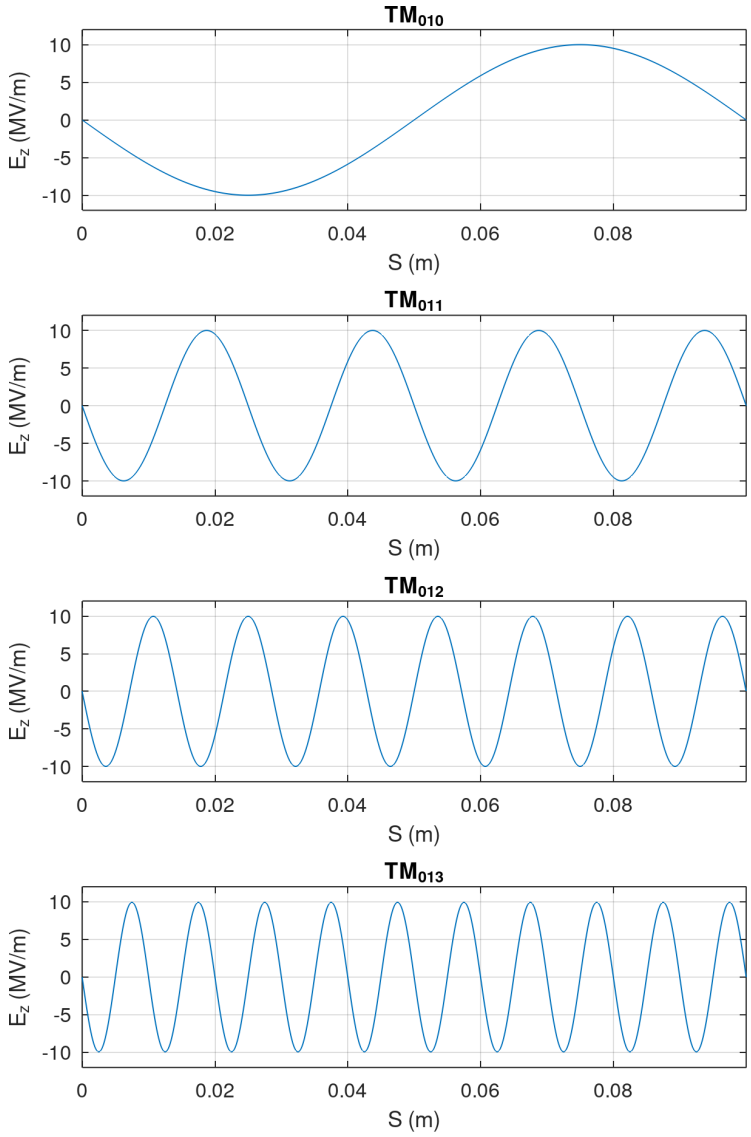


Figure 4.5: The longitudinal field at time $t = 0$, for a 3-cell TW structure with $f = 3$ GHz, phase advance $2\pi/3$, and peak field 10 MV/m. The four plots, top to bottom, have been obtained by setting the parameter `n` from 0 to 3.

4.3.6 Standing-wave structure

The element `SW_Structure` allows the simulation of the standing-wave (SW) structures connecting a beampipe with a TW structure. This enables the realistic simulation of a full TW structure, which must be preceded and followed by a SW structure. Figure 4.6 shows a model example of a standing-wave structure. In Table 4.4, J_n and I_n are the m^{th} ordinary Bessel function and m^{th} modified Bessel Function, respectively.

Constructors

```
S = SW_Structure ([ a1, ... ], frequency, cell_length, number_of_cells );
S = SW_Structure ();
```

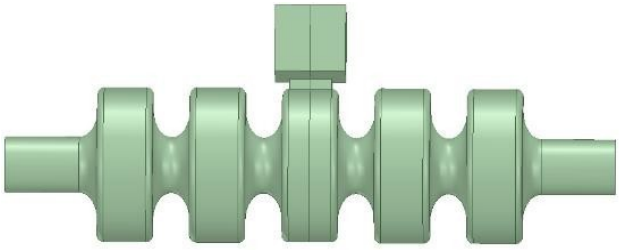


Figure 4.6: A model of a metallic standing-wave structure.

Table 4.4: Fourier series expansions of the TM_{01p} electromagnetic fields in a SW structure. In these equations, the flat ends of a cell are at $z = 0$ and $z = L$

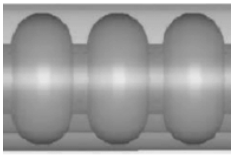
Quantity	**Value**
k_p	$\frac{p\pi}{L}$
q_p	$\sqrt{\left\|\left(\frac{\omega}{c}\right)^2 - k_p^2\right\|}$
$E_z(r,z,t)$	$\sum_{p=1}^{\infty} a_p \sin(k_p z)\cos(\omega t + \phi_0) \times \begin{cases} J_0(q_p r) & \frac{\omega}{c} \geq k_p \\ I_0(q_p r) & \frac{\omega}{c} < k_p \end{cases}$
$E_r(r,z,t)$	$\sum_{p=1}^{\infty} a_p \frac{k_p}{q_p}\cos(k_p z)\cos(\omega t + \phi_0) \times \begin{cases} J_1(q_p r) & \frac{\omega}{c} \geq k_p \\ I_1(q_p r) & \frac{\omega}{c} < k_p \end{cases}$
$B_\theta(r,z,t)$	$\sum_{p=1}^{\infty} a_p \frac{q_p^2 + k_p^2}{\omega q_p}\sin(k_p z)\sin(\omega t + \phi_0) \times \begin{cases} J_1(q_p r) & \frac{\omega}{c} \geq k_p \\ I_1(q_p r) & \frac{\omega}{c} < k_p \end{cases}$

The arguments are:

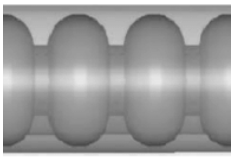
`[a1, ... ]`	a vector with the Fourier coefficients (see Table 4.4)	[V/m]
`frequency`	the rf frequency	[Hz]
`cell_length`	the length of each cell	[m]
`number_of_cells`	the number of cells in the structure.	[REAL]

As for the travelling-wave structure, a positive number of cells indicates that the structure starts from the start of the cell. A negative number indicates that the structure starts from the centre of a cell. Follows an example with three cells:

- `number_of_cells = 3`



- `number_of_cells = -3`



Example of a full structure

In this example, we propose an Octave function that builds a realistic 2.856 GHz structure consisting of (1) half a standing wave cell as the input coupler, (2) three cells of a travelling wave structure, and (3) half a standing wave cell as the output coupler. The Fourier coefficients were calculated using a fitting procedure over a 1D field map.

```
% Input arguments
% maxE is the rf gradient in V/m
% phid is the rf phase in deg

function TWS = make_tws (maxE, phid)
    RF_Track;
    n_cells = 3;
    freq = 2.856e9; % Hz
    ph_adv = 2*pi/3; % radian, phase advance per cell
    L_SW = 1.748602173202614e-01; % m, coupler length

    % Fourier modes SW
    A_SW = [  6.192020020706116e-01,  -6.953816923003616e-04, ... % k1, k2,
             -3.279125409035794e-01,   1.043714164141073e-04, ... % k3, k4,
              8.257347316504823e-02,  -1.849214043458953e-04, ... % ..
              8.669595750563971e-03,   2.327131602259179e-04, ...
             -1.597554108422184e-02,  -4.434041973047716e-04, ...
              5.789529950425815e-03 ];

    % Fourier modes TW
    A_TW = [  0.00132212262396783, -0.01924391586275388, ... % a_-3, a_-2,
              0.29326128415666941,  0.75002581163937343, ... % a_-1,  a_0,
             -0.02662679741546326,  0.00126167879112374, ... %   a1,   a2,
              8.3563612959562393e-06 ];                      %   a3

    % entrance coupler
    SWL = SW_Structure(maxE * A_SW, freq, L_SW, 0.5); % 1/2 SW
    SWL.set_t0(0.0);
    SWL.set_phid(phid);

    % travelling wave
    TW = TW_Structure(maxE * A_TW, -3, freq, ph_adv, n_cells);
    TW.set_t0(0.0);
    TW.set_phid(phid+90);
```

```
    % exit coupler
    SWR = SW_Structure(maxE * A_SW, freq, L_SW, -0.5); % 1/2 SW
    SWR.set_t0(0.0);
    SWR.set_phid(phid);

    % concatenate the three structures
    TWS = Lattice();
    TWS.append(SWL);
    TWS.append(TW);
    TWS.append(SWR);
end
```

Note the parameter `-3` as the second argument of the constructor of the element `TW_Structure`. It indicates the starting index of the Fourier modes provided:

```
A_TW = [   0.0013,   -0.0192,   0.2932,   0.7500,   -0.0266,   0.0012,   8.356e-06    ]
             a_-3       a_-2       a_-1       a_0        a_1        a_2        a_3
```

If, like in this example, n is -3, then RF-Track assumes that the input array `A_TW` contains 7 elements (that is, the number of elements in $n \in \{-3,-2,-1,0,1,2,3\}$).

4.3.7 Pillbox cavity

The element `Pillbox_Cavity` allows the simulation of the TM_{01n} modes in a standing-wave pillbox cavity. A pillbox cavity is a metallic enclosure where electromagnetic fields travelling with no phase advance can be excited. In this model, the flat ends of the cavity are at $z = 0$ and $z = L$. In Table 4.5, J_n and I_n are the m^{th} ordinary Bessel function and m^{th} modified Bessel Function, respectively.

Table 4.5: Fourier series expansions of the TM_{01p} electromagnetic fields in a pillbox cavity

Quantity	Value
k_p	$\frac{p\pi}{L}$
q_p	$\sqrt{\left\|\left(\frac{\omega}{c}\right)^2 - k_p^2\right\|}$
$E_z(r,z,t)$	$\sum_{p=0}^{\infty} a_p \cos(k_p z)\cos(\omega t + \phi_0) \times \begin{cases} J_0(q_p r) & \frac{\omega}{c} \geq k_p \\ I_0(q_p r) & \frac{\omega}{c} < k_p \end{cases}$
$E_r(r,z,t)$	$-\sum_{p=0}^{\infty} a_p \frac{k_p}{q_p} \sin(k_p z)\cos(\omega t + \phi_0) \times \begin{cases} J_1(q_p r) & \frac{\omega}{c} \geq k_p \\ I_1(q_p r) & \frac{\omega}{c} < k_p \end{cases}$
$B_\theta(r,z,t)$	$\sum_{p=0}^{\infty} a_p \frac{q_p^2 + k_p^2}{\omega q_p} \cos(k_p z)\sin(\omega t + \phi_0) \times \begin{cases} J_1(q_p r) & \frac{\omega}{c} \geq k_p \\ I_1(q_p r) & \frac{\omega}{c} < k_p \end{cases}$

Constructors

```
S = Pillbox_Cavity ([ a0, ... , ap ], frequency, cell_length, n_cells );
S = Pillbox_Cavity ();
```

The arguments are:

`[a0, ... , ap ]`	a vector with the Fourier coefficients (see Table 4.5)	[V/m]
`frequency`	the rf frequency	[Hz]
`cell_length`	the length of each cell	[m]
`n_cells`	the number of cells in the structure.	[REAL]

4.3.8 Multipole magnet

`Multipole` is a multipole magnet. The definition of the normal and skew coefficients is the same as that of MAD-X. In a `Multipole`, RF-Track uses numerical integration to track particles. The user can select the integration algorithm and number of steps using the set methods mentioned below (see Set and Get methods).

Constructors

```
M = Multipole (L, P_Q, [ K0L K1L K2L ... ]);
M = Multipole (L, [ S0 S1 S2 ... ]);
M = Multipole (L=0);
```

The arguments are:

`L`	the multipole magnet length	[m]
`P_Q`	the beam's magnetic rigidity, P/q	[MV/c]
`S0`, `S1`, ..., `Sn`	the (complex) multipole strengths	[MV/c/m^n]
`K0L`, `K1L`, ..., `KnL`	the (complex) multipole coefficients	[1/m^n]

Here follow a few formulæ to disentangle and clarify the relations between the integrated strengths, S; the integrated focusing strengths, k_nL; the field derivatives (gradients), B_n; and the beam rigidity, P/q:

$$k_nL = \left(\frac{\hat{k}_n}{1/\mathrm{m}^{n+1}}\right)\left(\frac{L}{\mathrm{m}}\right) \qquad [1/\mathrm{m}^n]$$

$$S_n = \left(\frac{P/q}{\mathrm{MV}/c}\right)\left(\frac{\hat{k}_n}{1/\mathrm{m}^{n+1}}\right)\left(\frac{L}{\mathrm{m}}\right) \qquad [\mathrm{MV}/c/\mathrm{m}^n]$$

$$B_n = \left(\frac{\partial^n \hat{B}}{\partial x^n}\frac{\mathrm{m}^n}{\mathrm{T}}\right) \qquad [T/\mathrm{m}^n]$$

The strengths, the multipole coefficients, and the field derivatives are vectors of complex numbers:

$$\hat{B} = B_y + iB_x,$$
$$\hat{k}_n = k_n^{(\mathrm{normal})} + ik_n^{(\mathrm{skew})}.$$

We recall the definition of "normal" and "skew" coefficients:

$$k_n^{(\mathrm{normal})} = \left(\frac{\mathrm{T{\cdot}m}}{B\rho}\right)\left(\frac{\partial^n B_y}{\partial x^n}\frac{\mathrm{m}^n}{\mathrm{T}}\right), \qquad [1/\mathrm{m}^{n+1}]$$

$$k_n^{(\mathrm{skew})} = \left(\frac{\mathrm{T{\cdot}m}}{B\rho}\right)\left(-\frac{\partial^n B_x}{\partial x^n}\frac{\mathrm{m}^n}{\mathrm{T}}\right). \qquad [1/\mathrm{m}^{n+1}]$$

Get methods

The obvious ones, plus:

```
S.get_Bn(); % the field derivatives in T/m^n
S.get_KnL(P_over_Q); % the integrated coefficient in 1/m^n
S.get_strengths(); % the integrated strength in MV/c/m^n
```

Set methods

The obvious ones, plus:

```
S.set_Bn([ B0 B1 B2 ... ]); % T/m^n, set the field derivatives
S.set_KnL(P_over_Q, [ K0L K1L K2L ... ]); % set the integrated coefficients
S.set_strengths([ S0 S1 S2 ... ]); %  MV/c/m^n, set the integrated strengths
```

Two methods allow setting the tracking options:

```
S.set_nsteps(N); % set the number of integration steps [default 10]
S.set_odeint_algorithm(name); % e.g. 'rk2', 'leapfrog', etc.
```

4.3.9 Absorber

The `Absorber` element is a block of matter where three collective effects affecting the beam are applied simultaneously: Multiple Coulomb scattering, Energy Straggling, and Stopping Power[1]. These enable the simulation of interactions between bunch particles and materials.

Optionally, `Absorber`s can have a circular or a rectangular shape. This is specified using the method:

```
A.set_shape (shape,  ax [m], ay [m] )
```

where shape can be either "`circular`", or "`rectangular`"; "`ax`" and "`ay`" are the radii in x and y.

Note that, when an `Absorber` is placed in a Volume, the computation of collective effects *must* be activated for it to be effective. This can be done by setting the TrackingOption `cfx_dt_mm`, see chapter 3. When an `Absorber` is placed in a Lattice, these collective effects are enabled by default.

Constructors

```
A = Absorber (L, material_name);
A = Absorber (L, X0, Z, A, density, I=-1);
```

The input arguments describing the material are:

`L`	absorber length	[m]
`material_name`	one among: '`air`', '`water`', '`beryllium`', '`lithium`', '`liquid_hydrogen`'	[STRING]
`X0`	radiation length	[cm]
`Z`	atomic number of absorber	[INTEGER]
`A`	atomic mass of absorber	[g mol^{-1}]
`density`	density of the absorber	[g cm^{-3}]
`I`	mean excitation energy. If unspecified or -1, an empirical approximation is used.	[eV]

Main methods

A set of '`enable`' and '`disable`' methods allows the full customization of this element.

```
A.enable_log_term();
A.enable_fruehwirth_model(); % DEFAULT
A.enable_wentzel_model(); % DEFAULT
A.enable_stopping_power(); % DEFAULT
A.enable_energy_straggling(); % DEFAULT
A.enable_multiple_coulomb_scattering(); % DEFAULT

A.disable_log_term(); % DEFAULT
A.disable_fruehwirth_model();
A.disable_wentzel_model();
A.disable_stopping_power();
A.disable_energy_straggling();
A.disable_multiple_coulomb_scattering();
```

[1] For a detailed description of the three effects, see the dedicated chapter.

4.3.10 Electron cooler

RF-Track can simulate electron cooling using a sophisticated hybrid kinetic model in which the beam's macro-particles interact with a cold, magnetised electron plasma. The electron plasma is modelled with 3D meshes, allowing the user to specify arbitrary transverse and longitudinal variations of density and current. The following panel presents its constructor and the main methods to customize the electron cooler simulation fully:

```
EC = ElectronCooler (L, rx, ry, density, Vz);

% Plasma mesh
EC.set_temperature (Tr, Tl);
EC.set_electron_mesh (Nx, Ny, Nz, density, Vx, Vy, Vz); % uniform plasma
EC.set_electron_mesh (Nz, DENSITY2D, VX2d, VY2D, VZ2D); % 2D profile
EC.set_electron_mesh (DENSITY3D, VX3D, VY3D, VZ3D); % full 3D mesh

% Magnetic field
EC.set_static_Bfield (Bx, By, Bz);

% Plasma particle (default: electron)
EC.set_Q (Q=-1);
EC.set_mass (mass=electronmass);
```

The main input arguments are:

`L`	the electron cooler's length	[m]
`rx`, `ry`	the horizontal and vertical radius of the plasma	[m]
`Tr`, `Tl`	the radial and longitudinal temperature of the plasma	[eV]
`density`	the plasma density	$[m^{-3}]$
`DENSITY2D`	the plasma density as a `Nx`×`Ny` matrix	$[m^{-3}]$
`DENSITY3D`	the plasma density as a `Nx`×`Ny`×`Nz` matrix	$[m^{-3}]$
`Bx`, `By`, `Bz`	the constant magnetic field magnetizing the plasma	[T]
`Vx`, `Vy`, `Vz`	the velocity of the plasma beam	$[c]$
`VX2D`, `VY2D`, `VZ2D`	the velocity of the plasma beam as a `Nx`×`Ny` matrix	$[c]$
`VX3D`, `VY3D`, `VZ3D`	the velocity of the plasma beam as a `Nx`×`Ny`×`Nz` matrix	$[c]$

4.3.11 Adiabatic matching device

An `AdiabaticMatchingDevice` (AMD) is a magnetic element providing a strong tapered solenoid field, typically used in positron sources. This device, also known as a "Flux Concentrator", is essential to capture the positrons right after their creation. In RF-Track, this element is implemented as an analytic 3D field. The on-axis field provided by an AMD is purely longitudinal and is described as,

$$B_z(z) = \frac{B_0}{1+\alpha z}.$$

Figure 4.7 shows the on-axis field B_z as a function of z for an AMD with $B_0 = 7\,\mathrm{T}$ and $\alpha = 60\,\mathrm{m}^{-1}$.

Constructor

```
AMD = AdiabaticMatchingDevice (L, B0, ALPHA);
```

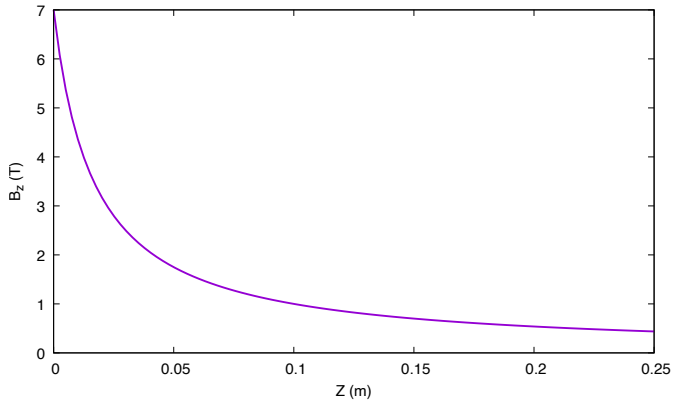


Figure 4.7: Magnetic profile in an AMD.

The arguments are:

`L`	the AMD length	[m]
`B0`	the peak on-axis field, B_0	[T]
`ALPHA`	the parameter α	[m^{-1}]

Main methods

The aperture of an Adiabatic Matching Device has the shape of a truncated cone. You can specify the aperture radius of each end of the AMD using the following two methods:

```
A.set_entrance_aperture (R1);
A.set_exit_aperture (R2);
```

Where `R1` and `R2` are the entrance and exit aperture radii in metres.

4.3.12 Space-charge Field

`SpaceCharge_Field` is a special element that allows you to know the electromagnetic field generated by an arbitrary distribution of particles at any point in space, using the method `get_field()`. It can be used to simulate weak-strong interactions in `Volume`.

In its constructor, `SpaceCharge_Field` accepts a particle distribution given as `Bunch6dT` as an input. Inside the rectangular bounding box enclosing all particles, the field is computed using the same 3D space-charge calculation routines used during tracking: a 3D PIC based on FFT-integrated retarded Green's functions in free space. Outside the bounding box (that is, at large distances), the field is computed using a Cartesian multipole expansion of the charge and current distribution in space to 5^{th} order.

Since `SpaceCharge_Field` creates an electromagnetic field that permeates the entire space, this element is intended for use in Volume only. Figure 4.8 provides a visual example of `SpaceCharge_Field` and the distribution used for its initialisation.

Constructor

```
SC = SpaceCharge_Field (B0T, Nx, Ny, Nz, Vz_slices=1);
```

The arguments are:

`B0T`	an arbitrary particle distribution	[`Bunch6dT`]
`Nx, Ny, Nz`	the number of mesh points for the 3D PIC solver	[INTEGER]
`Vz_slices`	the number of velocity slices accounting for relativistic effects	[INTEGER]

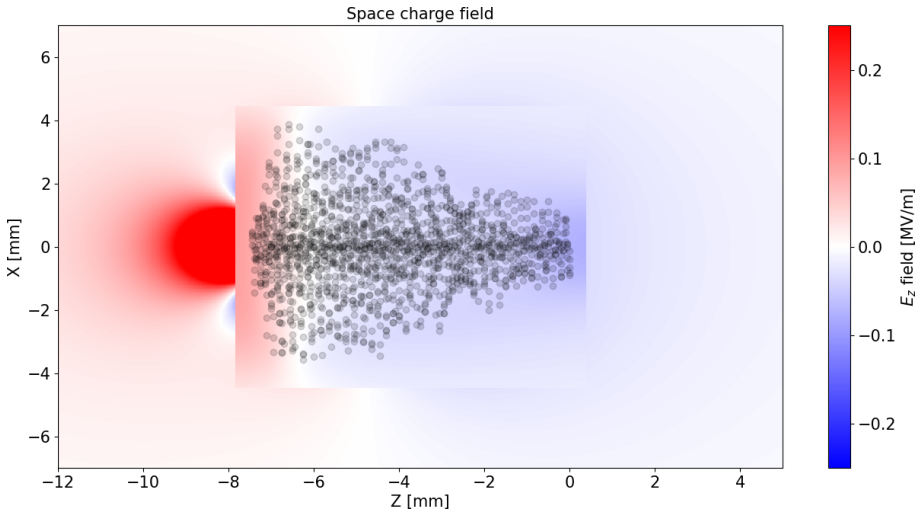


Figure 4.8: Electromagnetic field generated by a particle distribution, computed by the element `SpaceCharge_Field`. The field extends both internally and externally to the particle distribution generating it. A `SpaceCharge_Field` can be superimposed on external fields for the accurate simulation of sources. (Figure courtesy of Manon Boucard, PMB-Alcen, France).

4.3.13 Travelling-Wave Field

The element `TW_Field` allows the insertion of a travelling-wave structure directly from its shunt impedance, group velocity, and quality factor. With this element, explicit field-mapping fields for the electric and magnetic fields aren't necessary. In order to compute the field, RF-Track also needs to know the input power into the structure. The input power can be provided in two ways:

1. Constant P_{in} in W. When given a P_{in}, RF-Track computes the steady state of the field.
2. Dynamic P_{in}. The input power is provided as a 1D array of time-dependent values, and an injection time $t_t extinj$ is specified, determining the moment the bunch enters the structure along the power input.

Constructors

```
TW = TW_Field (P_in,            Q, r_Q, VG, freq, ph_adv, n_cells, z0_L=0 );
TW = TW_Field (P_in, dt, t_inj, Q, r_Q, VG, freq, ph_adv, n_cells );
```

The arguments are:

`P_in`	The input power in the structure. In the first form of the constructor, it is a scalar number. The assumption here is an infinitely long input power pulse. The field is computed after the filling is completed. In the second constructor, it is a vector of numbers sampling the input power pulse at regular time interval `dt`	[W]
`dt`	The time step of the input vector `P_in`	[mm/c]
`t_inj`	the time of injection of the beam in the structure, with respect to the input power vector `P_in`	[mm/c]
`Q`	The quality factor along the structure	
`r_Q`	Normalised shunt impedance per unit length array (r/Q)	[Ω/m]
`vg`	Group-velocity array	[c]
`ph_adv`	The phase advance per cell	[rad]
`n_cells`	The number of cells	[INTEGER]
`z0_L`	The length of the input and output couplers, expressed as a fraction of the cell length L_{cell}. If `z0_L` $\neq 0$, RF-Track will add a standing-wave coupler before and one after the travelling-wave body of the structure, with a length $$L_{\text{coupler}} = \texttt{z0\_L} \cdot L_{\text{cell}},$$ where L_{cell} is the length of the structure's cell	[REAL]

Notice that the key vectors describing the structure field, `Q`, `r_Q`, `VG`, must have the same number of elements.

4.3.14 LaserBeam

In Lattice, RF-Track can simulate inverse Compton scattering (ICS) between *any* charged particle and a laser beam in an element called `LaserBeam`. Chapter 5 illustrates this element and the ICS simulation in detail.

4.3.15 Volume as a Lattice element

The tracking environment Volume can be inserted into a Lattice like any standard element. When a `Bunch6d`, which is being tracked through a Lattice, enters a Volume that has been inserted into a Lattice as an element, its particles are placed over the are emitted one by one from the Volume's surface S_0, according to their time coordinate. Tracking is then performed in the Volume by time integration and stops when no more particles exist between S_0 and S_1.

During tracking, when a particle reaches S_1 and leaves the Volume, its position at S_1 and its arrival time are recorded, and a bunch of type `Bunch6d` is formed for propagation through the rest of the lattice. This way, a section where time integration is used with all the flexibility offered by Volume can be placed in a Lattice.

Note that the planes S_0 and S_1 can have any orientation in space. This allows for changes in the reference system, for example, when tracking in the field map of a bending magnet or when joining regions at an angle.

4.4 Field maps

One of RF-Track's strengths is tracking in field maps. Real and complex field maps, in one, two, or three dimensions, are accepted to simulate static fields, backward— or forward-travelling fields, or standing-wave radiofrequency fields.

Elements based on a one-dimensional on-axis field map still provide three-dimensional fields. By proving the longitudinal component of the field along the axis of symmetry, RF-Track extends the field off-axis according to Maxwell's equations, assuming cylindrical symmetry. Two-dimensional field maps allow the simulation of cylindrical-symmetric fields from the field over a plane. Three-dimensional field maps allow tracking in most generic electromagnetic fields. All field maps accept 3D Cartesian mesh grids with regular spacing.

Interpolation methods

RF-Track offers two interpolation methods

- Linear interpolation (LINT), the field is evaluated at any point by linearly interpolating with the eight nearest mesh points. This method is very fast and is the default method.
- Cubing interpolation (CINT), the field is evaluated in any point by cubically interpolating with the nearest 64 mesh points. This method is significantly slower but produces a smoother field.

Particle losses in field maps

In RF-Track, three-dimensional field maps can contain more information than the field itself. They can contain information about the walls of the element in which the field is embedded, using the special value "not-a-number" (NaN). In floating-point arithmetic, NaN is a numeric data type that can be interpreted as an undefined value. RF-Track interprets the presence of NaNs in a field map as a "wall". If a particle hits a NaN, RF-Track flags the particle as lost. This allows precise detection of losses in 3D space, even in complex geometries.

Interpolation method and losses detection

In the case of LINT, the interpolation uses the eight 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 cube with $3 \times 3 \times 3$ mesh cells surrounding the point of interest. This means that the granularity of the loss detection is effectively three times the size of a mesh cell.

CINT is generally smoother than LINT, as it considers four adjacent points instead of 2. The price for the increased smoothness is computational time.

4.4.1 1D RF field maps

An `RF_FieldMap_1d` provides an RF oscillating electromagnetic field based on the on-axis electric field. RF-Track assumes cylindrical symmetry around the structure's axis and reconstructs the off-axis transverse and longitudinal electromagnetic field, fulfilling Maxwell's equations. The user must provide a regular Cartesian 1D mesh of the longitudinal electric field E_z as complex (travelling waves) or real (standing waves) numbers.

Constructors

```
RF = RF_FieldMap_1d (Ez,
                     hz,
                     length,
                     frequency,
                     direction,
                     P_max = 1,
                     P_actual = 1);

RF = RF_FieldMap_1d_CINT ( " );
```

The default element `RF_FieldMap_1d` uses linear interpolation along the longitudinal axis and linear extrapolation in the radial direction. The variant, `RF_FieldMap_1d_CINT`, uses cubic interpolation along the longitudinal axis and cubic extrapolation in the radial direction.

The required input arguments are:

`Ez`	The on-axis electric field	[V/m]
`hz`	The mesh cell in the longitudinal direction	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]
`frequency`	The RF frequency (use 0 for static fields)	[Hz]
`direction`	0 : static field 1 : forward-travelling field −1 : backward-travelling field NOTE: standing-wave fields can have +1 or -1 identically	
`P_map`	The input power used to generate the input field map (default 1)	[W]
`P_actual`	The actual input power to operate the element (default 1)	[W]

If the user only needs to simulate an electric or magnetic field, the constant number zero, `0`, can be provided for the unnecessary field components.

Main methods

A set of dedicated methods allows the user to set the phase, the reference time, and the actual input power to the structure:

```
RF.set_t0(T0);
RF.unset_t0();
RF.set_phid(PHID);
RF.set_P_actual(P);
RF.set_smooth(N);
```

The input parameters are:

`T0`	the reference time	[mm/c]
`PHID`	the phase of the RF element	[deg]
`P`	the input power	[W]
`N`	number of points	[INTEGER]

The method `set_smooth(N)` applies a Gaussian filter to the field map to reduce numerical noise. The parameter `N` indicates the radius of the Gaussian function, kernel of the filter, in units of mesh points. Mathematically, a Gaussian filter corresponds to a convolution with a Gaussian function.

On the reference time

By default, the reference time of an RF element, t_0, is not set. The method `set_t0(T0)` allows the user to set the reference time of the RF element. When t_0 is not explicitly set, RF-Track sets it automatically with an "autophasing" (see dedicated section). This is the most common and straightforward way of setting it.

On the input power

Electromagnetic solvers generally use a pre-defined input power of 1 W to compute the RF electromagnetic field in the structure. If this is the case, one can directly give RF-Track the field as the solver computed it, $\mathbf{E}_{\text{map}}$, with the indication of the input power used for its computation, P_{map}. For tracking, though, one should provide the power at which the structure is operated to obtain the actual field acting on the particles. The relation between these quantities is the following:

$$\mathbf{E}_{\text{actual}} = \mathbf{E}_{\text{map}} \sqrt{\frac{P_{\text{actual}}}{P_{\text{map}}}}.$$

If the input field map already provides the nominal field, one can accept the default values $P_{\text{map}} = 1$ and $P_{\text{actual}} = 1$ to provide an unscaled field.

4.4.2 2D RF field maps

An `RF_FieldMap_2d` provides an RF oscillating electromagnetic field based on a field map defined over a 2D plane. RF-Track assumes cylindrical symmetry around the structure axis. The user must provide a regular Cartesian 2D mesh per each field component: E_r, E_z, and B_t, B_z. The four 2D meshes can be either complex numbers (travelling waves) or real numbers (standing waves). Their elements,

$$E_{r_{ij}},\ E_{z_{ij}},, \text{ and } B_{t_{ij}},\ B_{z_{ij}},$$

have two indexes, ij: the first, i, runs along the longitudinal direction, and the second, j, along the radial direction.

Constructors

```
RF = RF_FieldMap_2d (Er, Ez,
                     Bt, Bz,
                     hr, hz,
                     length,
                     frequency,
                     direction,
                     P_max = 1,
                     P_actual = 1);

RF = RF_FieldMap_2d_CINT ( " );
```

The default element `RF_FieldMap_2d` uses linear interpolation.

The variant, `RF_FieldMap_2d_CINT`, provides cubic interpolation.

The required input arguments are:

`Er`, `Ez`	2D mesh of the radial and longitudinal electric field, or zero `0` if the electric field is absent	[V/m]
`Bt`, `Bz`	2D mesh of the azimuthal and longitudinal magnetic field, or zero `0` if the magnetic field is absent	[T]
`hr`, `hz`	The sizes of a mesh cell in the radial and longitudinal directions	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]
`frequency`	The RF frequency (use 0 for static fields)	[Hz]
`direction`	0 : static field 1 : forward-travelling field −1 : backward-travelling field NOTE: standing-wave fields can have +1 or -1 identically	
`P_map`	The input power used to generate the input field map (default 1)	[W]
`P_actual`	The actual input power to operate the element (default 1)	[W]

If the user only needs to simulate an electric or magnetic field, the constant number zero, `0`, can be provided for the unnecessary field components.

Main methods

A set of dedicated methods allows the user to set the phase, the reference time, and the actual input power to the structure:

```
RF.set_t0(T0);
RF.unset_t0();
RF.set_phid(PHID);
RF.set_P_actual(P);
RF.set_smooth(N);
```

The input parameters are:

`T0`	the reference time	[mm/c]
`PHID`	the phase of the RF element	[deg]
`P`	the input power	[W]
`N`	number of points	[INTEGER]

The method `set_smooth(N)` applies a Gaussian filter to the field map to reduce numerical noise. The parameter `N` indicates the radius of the Gaussian function, kernel of the filter, in units of mesh points. Mathematically, a Gaussian filter corresponds to a convolution with a Gaussian function.

On the reference time

By default, the reference time of an RF element, t_0, is not set. The method `set_t0(T0)` allows the user to set the reference time of the RF element. When t_0 is not explicitly set, RF-Track sets it automatically with an "autophasing" (see dedicated section). This is the most common and simple situation.

On the input power

Electromagnetic solvers generally use a pre-defined input power of 1 W to compute the RF electromagnetic field in the structure. If this is the case, one can directly give RF-Track the field as the solver computed it, $\mathbf{E}_{\text{map}}$, with the indication of the input power used for its computation, P_{map}. For tracking, though, one should provide the power at which the structure is operated to obtain the actual field acting on the particles. The relation between these quantities is the following:

$$\mathbf{E}_{\text{actual}} = \mathbf{E}_{\text{map}} \sqrt{\frac{P_{\text{actual}}}{P_{\text{map}}}}.$$

If the input field map already provides the nominal field, one can accept the default values $P_{\text{map}} = 1$ and $P_{\text{actual}} = 1$ to provide an unscaled field.

4.4.3 3D RF field maps

An `RF_FieldMap` provides an RF oscillating electromagnetic field. The user must provide a regular Cartesian 3D mesh per each field component: E_x, E_y, E_z, and B_x, B_y, B_z, or alternatively the constant 0 when the field is unknown or null.

The 3D meshes can be either complex numbers (travelling waves) or real numbers (standing waves). Their elements,

$$E_{x_{ijk}},\ E_{y_{ijk}},\ E_{z_{ijk}},\ \text{and}\ B_{x_{ijk}},\ B_{y_{ijk}},\ B_{z_{ijk}},$$

have three indexes, ijk: the first, i, runs along the horizontal direction, x; the second, j, runs along the vertical direction, y; and the third, k, runs along the longitudinal direction z.

Constructors

```
RF = RF_FieldMap (Ex, Ey, Ez,
                  Bx, By, Bz,
                  x0, y0,
                  hx, hy, hz,
                  length,
                  frequency,
                  direction,
                  P_max = 1,
                  P_actual = 1);
RF = RF_FieldMap_CINT ( " );
```

The default element `RF_FieldMap` uses linear interpolation.

The variant `RF_FieldMap_CINT` provides cubic interpolation.

The required input arguments are:

`Ex`, `Ey`, `Ez`	3D mesh of the electric field, or zero, `0`	[V/m]
`Bx`, `By`, `Bz`	3D mesh of the magnetic field, or zero, `0`	[T]
`x0`, `y0`	The position of the bottom-left starting point of the field map in the $x-y$ plane	[m]
`hx`, `hy`, `hz`	The sizes of a mesh cell in the x, y, and z directions	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]
`frequency`	The RF frequency (use 0 for static fields)	[Hz]
`direction`	0 : static field 1 : forward-travelling field -1 : backward-travelling field NOTE: standing-wave fields can have +1 or -1 identically	
`P_map`	The input power used to generate the input field map (default 1)	[W]
`P_actual`	The actual input power to operate the element (default 1)	[W]

If the user only needs to simulate an electric or magnetic field, the constant number zero, `0`, can be provided for the unnecessary field components.

Main methods

A set of dedicated methods allows the user to set the phase, the reference time, and the actual input power to the structure:

```
RF.set_t0(T0);
RF.unset_t0();
RF.set_phid(PHID);
RF.set_P_actual(P);
RF.set_smooth(N);
```

The input parameters are:

`T0`	the reference time	[mm/c]
`PHID`	the phase of the RF element	[deg]
`P`	the input power	[W]
`N`	number of points	[INTEGER]

The method `set_smooth(N)` applies a Gaussian filter to the field map to reduce numerical noise. The parameter `N` indicates the radius of the Gaussian function, kernel of the filter, in units of mesh points. Mathematically, a Gaussian filter corresponds to a convolution with a Gaussian function.

On the reference time

By default, the reference time of an RF element, t_0, is not set. The method `set_t0(T0)` allows the user to set the reference time of the RF element. When t_0 is not explicitly set, RF-Track sets it automatically with an "autophasing" (see dedicated section). This is the most common and simple situation.

On the input power

Electromagnetic solvers generally use a pre-defined input power of 1 W to compute the RF electromagnetic field in the structure. If this is the case, one can directly give RF-Track the field as the solver computed it, $\mathbf{E}_{\text{map}}$, with the indication of the input power used for its computation, P_{map}. For tracking, though, one should provide the power at which the structure is operated to obtain the actual field acting on the particles. The relation between these quantities is the following:

$$\mathbf{E}_{\text{actual}} = \mathbf{E}_{\text{map}} \sqrt{\frac{P_{\text{actual}}}{P_{\text{map}}}}.$$

If the input field map already provides the nominal field, one can accept the default values $P_{\text{map}} = 1$ and $P_{\text{actual}} = 1$ to provide an unscaled field.

4.4.4 1D static magnetic field maps

An `Static_Magnetic_FieldMap_1d` provides a static magnetic field based on the on-axis field. RF-Track assumes cylindrical symmetry around the structure's axis and reconstructs the field's off-axis transverse and longitudinal components, fulfilling Maxwell's equations. The user must provide a regular Cartesian 1D mesh of the longitudinal magnetic field B_z.

Constructors

```
M = Static_Magnetic_FieldMap_1d (Bz, hz, length=-1 );
M = Static_Magnetic_FieldMap_1d_CINT ( " );
```

The element `Static_Magnetic_FieldMap_1d` uses linear interpolation along the longitudinal axis and linear extrapolation in the radial direction. `Static_Magnetic_FieldMap_1d_CINT` uses cubic interpolation along the longitudinal axis and cubic extrapolation in the radial direction.

The required input arguments are:

`Bz`	The on-axis magnetic field	[T]
`hz`	The mesh cell length in the longitudinal direction	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]

4.4.5 2D static magnetic field maps

An element of type `Static_Magnetic_FieldMap_2d` provides a magnetic field based on a field map defined over a 2D plane. RF-Track assumes cylindrical symmetry around the element's axis. The user must provide a regular Cartesian 2D mesh per each field component: B_r and B_z. The two input 2D meshes,

$$B_{r_{ij}},\ B_{z_{ij}}$$

have two indexes, ij: the first, i, runs along the longitudinal direction, and the second, j, along the radial direction.

Constructors

```
S = Static_Magnetic_FieldMap_2d (Br, Bz, hr, hr, length=-1);
S = Static_Magnetic_FieldMap_2d_CINT ( " );
```

The element `Static_Magnetic_FieldMap_2d` uses linear interpolation.

The variant, `Static_Magnetic_FieldMap_2d_CINT`, provides cubic interpolation.

The required input arguments are:

`Br`, `Bz`	2D mesh of the radial and longitudinal magnetic field	[T]
`hr`, `hz`	The sizes of a mesh cell in the radial and longitudinal directions	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]

4.4.6 3D static magnetic field maps

An element of type `Static_Magnetic_FieldMap` provides a magnetic field based on a 3D field map. It accepts three input 3D meshes or four if one specifies the field using its scalar and vector potentials.

Constructors

```
S = Static_Magnetic_FieldMap (Bx, By, Bz,
                              x0, y0,
                              hx, hy, hz,
                              length);

S = Static_Magnetic_FieldMap (Ax, Ay, Az, PhiM,
                              x0, y0,
                              hx, hy, hz,
                              length);
```

The required input arguments are:

`Bx`, `By`, `Bz`	3D mesh of the magnetic field	[T]
`Ax`, `Ay`, `Az`	3D mesh of the vector magnetic potential `0`	[T m]
`PhiM`	3D mesh of the scalar magnetic potential `0`	[T m]
`x0`, `y0`	The position of the bottom-left starting point of the field map in the $x-y$ plane	[m]
`hx`, `hy`, `hz`	The sizes of a mesh cell in the x, y, and z directions	[m]
`length`	the total length of the element (if -1 take the field map length)	[m]

4.5 Beam diagnostics

4.5.1 Beam position monitor

The `Bpm` element adds a beam position monitor to a Lattice. The Bpm can be thick and have a user-defined resolution. The scaling error can also be applied. Reading a Bpm can be done after tracking a beam. Note that subsequent readings for the same Bpms will result in different readings, as each measurement is affected by the resolution error.

The Bpm element is currently only implemented in Lattice.

Constructor

```
B = Bpm (L, resolution );
```

The input arguments describing the material are:

`L`	the element length. The reading occurs in the middle of the specified length.	[m]
`resolution`	the resolution	[mm]

After tracking, to get all Bpms' readings at once, one can call the method `get_bpm_readings()` of the Lattice object.

Get methods

```
[X,Y] = B.get_reading(); % return mm
res = B.get_resolution(); % return mm
```

Set methods

```
B.set_resolution(res); % accept mm
B.set_scaling_factor(X_scaling, Y_scaling = X_scaling);
```

4.5.2 Screens

The `Screen` element adds a Screen to a Lattice or Volume. A screen is a thin element that captures a snapshot of the phase space of a Beam or just a Bunch6d when this is traversing the screen itself.

Screens can have a finite extension and a specific time window; see the methods below. When a time window is specified, the screen is automatically synchronized to the first bunch traversing the screen unless the user selects a specific activation time.

In a Lattice, Screens can be placed between elements with any arbitrary offset. In Volume, Screens can be placed in any position with any orientation in space. When a bunch traverses a screen, RF-Track retains the arrival time of each particle at the screen and creates a Bunch6d with the bunch's phase space in the screen's reference system.

After tracking, the phase space of the bunch (or of the beam) can be retrieved from Volume and Lattice using the methods `get_bunch_at_screens()` or `get_beam_at_screens()`, depending on whether one is tracking a single or multi-bunch beam. These methods return a list of Bunch6d's or Beams objects, one per screen.

Constructor

```
S = Screen ();
```

Get methods

```
B = S.get_bunch(); % return a Bunch6d
B = S.get_beam(); % return a Beam
t0 = S.get_t0(); % returns the reference time in mm/c
```

Set methods

```
S.set_width (W); % mm, screen width W
S.set_height (H); % mm, screen height H
S.set_time_window (T); % mm/c, screen time window T
S.set_t0 (t0); % set reference time [default: unset]
S.unset_t0 (); % unset the reference time
```

The input arguments are:

`W`	Width. Particles hit the screen if $-\mathtt{W}/2 \leq x \leq \mathtt{W}/2$.	[mm]
`H`	Height. Particles hit the screen if $-\mathtt{H}/2 \leq y \leq \mathtt{H}/2$.	[mm]
`T`	Time window. Particles are stored if $-\mathtt{T}/2 \leq (t - \mathtt{t0}) \leq \mathtt{T}/2$.	[mm/c]
`t0`	the reference time [default unset]	

By default, a new Screen has infinite extension and a boundless time window.

5. Collective Effects

In RF-Track, multiple single-particle and collective effects can be added to each element and overlapped. If, for example, one needs to simulate two collective effects, say `EFFECT1` and `EFFECT2`, one can do:

```
E = Drift (1.0); % 1m long drift, or whichever other element
E.add_collective_effect (EFFECT1);
E.add_collective_effect (EFFECT2);
```

to have both effects act on the beam when the beam travels through the element `E`.

Collective effects in Lattice

In Lattice, the effects are uniformly distributed along the element over a user-defined number of steps set through the method `set_cfx_nsteps()`:

```
E.set_cfx_nsteps (NUMBER_OF_STEPS);
```

The kicks due to collective effects are applied using a *velocity Verlet* algorithm, which is similar to the leapfrog method. This means that, for example, if only one integration step is used, the algorithm will apply a half-kick at the entrance of the element, perform the tracking through the entire element in one step, and then apply the second half-kick at the exit.

Collective effects in Volume

Within a Volume, the user must specify how frequently collective effects should be computed by setting the tracking parameter `cfx_dt_mm`, which defines a time step expressed in mm/c. Typically, `cfx_dt_mm` is larger than the integration step `dt_mm` used to solve the equations of motion. Here is an example:

```
V = Volume();
V.dt_mm = 0.1; % mm/c, fine integration step
V.cfx_dt_mm = 10; % mm/c, apply collective effects every 10 mm/c
```

Collective effects in RF-Track are computed using parallel algorithms. However, the degree of parallelism achievable when tracking without collective effects is inherently superior, as particles are fully independent and there is no interaction between them. For this reason, simulations run faster when `cfx_dt_mm` is large compared to `dt_mm`. Of course, there is a trade-off between simulation speed and the accuracy of the results. A convergence study is therefore always recommended.

5.1 Space charge

Space charge effects are handled differently from other collective effects. While space charge is inherently associated with the bunch itself, other collective effects typically depend on specific components of the beamline. For instance, wakefields arise from accelerating structures (e.g., short-range wakes) or from particular sections of the beamline (e.g., beam pipes producing resistive-wall wakes). Likewise, spurious magnetic multipole fields are confined to specific magnetic elements.

5.1.1 Space-charge in Volume

As discussed in Chapter 2 (Beam Models), the `Volume` (time integration) environment is the most appropriate for space-charge-dominated regimes, such as injectors and other low-energy systems. In `Volume`, space charge effects are modeled as acting continuously throughout the entire 3D space.

Space charge can be activated by setting the tracking parameter `sc_dt_mm`, which specifies the kick interval, i.e., how often a space-charge kick is applied during tracking. The unit is mm/c:

```
V = Volume()
V.sc_dt_mm = 10; # mm/c
```

As with `cfx_dt_mm`, it is generally advantageous to choose a space-charge kick interval `sc_dt_mm` that is larger than the integration step `dt_mm`. This improves computational efficiency by enabling highly parallel tracking, particularly in situations where the bunch charge distribution remains nearly constant between successive kicks.

5.1.2 Space-charge in Lattice

Space charge effects can also be simulated in the `Lattice` (space integration) environment, but additional considerations are required due to the nature of space-based tracking. When using space integration, i.e., with a `Bunch6d`, all particles are located on the same longitudinal plane at each step, and their arrival time is represented by the fifth phase space coordinate.

However, the evaluation of the space charge force requires the spatial configuration of particles at the same time, rather than their time of arrival at a common longitudinal location. To address this, RF-Track performs an on-the-fly transformation of the bunch: the particles are virtually drifted so that they become simultaneous in time, with their average longitudinal position aligned to the current integration position. This transformation is used solely for the computation of the space charge kicks.

A drift step is applied to each particle to achieve this time alignment. Once the space charge kicks are computed and applied, tracking proceeds as usual. This technique ensures that space charge effects can be accurately modeled within the `Lattice` framework, despite its inherently spatial nature.

In the `Lattice` environment, space charge effects are enabled by specifying the number of space charge kicks to be applied within each element. This is done on an element-by-element basis:

```
E = Element(); % any element
E.set_sc_nsteps(NUMBER_OF_STEPS);
```

Here, NUMBER_OF_STEPS is a positive integer indicating how many evenly spaced space charge kicks should be applied across the element's length. A larger number of steps increases accuracy but may increase computation time.

5.1.3 Space Charge Models

RF-Track provides two algorithms for modeling space charge effects:

```
SC = SpaceCharge_PIC_FreeSpace(Nx=16, Ny=16, Nz=16);
SC = SpaceCharge_P2P();
```

SpaceCharge_PIC_FreeSpace() implements a free-space fast Fourier solver based on 3D retarded integrated Green's functions. This is the default and recommended space charge algorithm. The parameters Nx, Ny, and Nz define the number of mesh points used along each spatial dimension. These values can be any positive integers—odd or even—and do not need to be powers of two. The space charge field is recomputed on the updated grid at every kick.

SpaceCharge_P2P() implements a particle-to-particle algorithm that computes the electromagnetic interaction between all pairs of macroparticles. This method is significantly slower and more susceptible to unphysical artifacts, especially when particles with large charge are too close to one another. It was originally implemented for academic purposes and retained for consistency checks. Use of SpaceCharge_PIC_FreeSpace() is strongly recommended in all practical scenarios.

Importantly, unlike many other simulation codes, RF-Track accounts for both electric and magnetic contributions to the space charge force. As a result, beam–beam interactions are naturally included in the simulation.

Setting the space-charge algorithm

The space charge computing engine can be changed by setting the RF-Track settings' variable SC_engine. In Octave, this is done like in the following example:

```
SC = SpaceCharge_PIC_FreeSpace(40, 40, 40);
RF_Track.SC_engine = SC;
```

In Python,

```
SC = SpaceCharge_PIC_FreeSpace(40, 40, 40)
RF_Track.cvars.SC_engine = SC
```

This is usually done at the beginning of a script, as it affects all subsequent space charge calculations. The default engine is SpaceCharge_PIC_FreeSpace(32, 32, 32).

Space-charge force

The space charge routines can also compute the space-charge force outside of tracking using the method compute_force(). For example,

```
B0 = Bunch6d ( ... );    % A bunch, or a Bunch6dT
SC = SpaceCharge_PIC_FreeSpace (40, 40, 40); % SC
```

```
F = SC.compute_force (B0); % returns the force in MeV/m
```

`F` is a 3-column matrix, with as many rows as particles in the bunch `B0`. The three columns are F_x, F_y, and F_z, the three components of the force exerted on each particle, expressed in MeV/m.

5.2 Intra-Beam Scattering

Small-angle Coulumb collisions within the bunch, also known as Intra-Beam Scattering, can be computed as a collective effect named `IntraBeamScattering`. The input arguments are four:

```
IBS = IntraBeamScattering (Nx = 16, Ny = 16, Nz = 16,  Ncol = -1);
```

where `Nx`, `Ny`, and `Nz` are integer numbers that set the number of mesh points for the calculation. Their default value is 16. To ensure full physics consistency, it is recommended that the same number of 3D mesh points be used in both the space-charge and intra-beam-scattering modules.

The parameter `Ncol` specifies the maximum number of intra-beam scattering (IBS) collisions simulated per kick. By default, `Ncol` is set to -1, indicating that all collisions expected within the integration step need to be computed explicitly. To reduce computational time, the user may set `Ncol` to a small positive value, in which case only a subset of collisions is simulated directly; the remaining collisions are approximated using an analytical model.

5.3 Incoherent synchrotron radiation

In RF-Track, the emission of synchrotron radiation is generated by *any fields* acting on the particle, both electric and magnetic. The impact of synchrotron radiation emission can be considered using the collective effect `IncoherentSynchrotronRadiation`, which accepts just one optional, boolean argument:

```
I = IncoherentSynchrotronRadiation ( quantum=false, track_photons=false );
```

If `quantum` is set to `true`, the synchrotron radiation emission will be a stochastic Monte Carlo process; otherwise, the effect will be an average energy loss.

If `track_photons` is set to `true`, the synchrotron radiation photons will be added to the bunch and tracked with it throughout the accelerator.

5.4 Magnetic multipolar errors

`MultipoleKick` allows the attachment of multipole magnetic errors to *any* element. It accepts a list of complex gradients:

```
M = MultipoleKick ([ B0, B1, ... Bn ]);
```

The argument is:

`B0`, `B1`, ..., `Bn` the (complex) gradients [T/m^n]

with

$$B_n = \left(\frac{\partial^n \hat{B}}{\partial x^n} \frac{\mathrm{m}^n}{\mathrm{T}} \right) \qquad [T/\mathrm{m}^n]$$

The real part of each B_n represents the field's "normal" component, while the imaginary part represents the "skew" component.

5.5 Beam loading

The beam loading effect is the gradient reduction in the accelerating structure due to the beam's excitation of the fundamental accelerating mode. This translates into a transient effect for long trains where the late bunches will be affected by the EM field left by earlier bunches. RF-Track allows the consideration of this effect both in travelling-wave and standing-wave structures.

5.5.1 Beam loading in ultrarelativistic scenarios

The beam loading effect in ultrarelativistic scenarios ($v \simeq c$) is considered for both travelling-wave structures and standing-wave structures using the collective effect `BeamLoading()`. The following constructors distinguish between transient and steady-state:

Constructors

```
% Steady state beam loading - For trains entering after tfill
% Fixed charge per bunch
BL = BeamLoading(Ncells, freq, ph_ad, Q, r_Q, vg, q, particles_bunch, ...
  fb, nbins);
% Transient beam loading - For bunches entering before tfill.
% No assumption on the train distribution
BL = BeamLoading(Ncells, freq, ph_ad, Q, r_Q, vg, nbins);
```

The possible arguments are:

`Ncells`	Number of cells of the TW structure	
`freq`	RF-Frequency	[Hz]
`ph_ad`	Phase advance	[rad]
`Q`	Quality factor array	
`r_Q`	Normalised shunt impedance per unit length array (r/Q)	[Ω/m]
`vg`	Group-velocity array	[c]
`nbins`	Number of bins for single-bunch longitudinal slicing	
`q`	Single particle charge	[e]
`particles_bunch`	Number of particles per bunch	
`fb`	Bunch injection frequency	[Hz]

If no `nbins` value is provided, the default value is `nbins=16`.

Solve methods

For trains with bunches of equal charge and fixed spacing, the beam-induced gradient exhibits a transient behaviour which, after a certain time (t_{fill} for TW structures or 5τ for SW structures), stabilises at the so-called steady state. In this case, the beam-induced gradient be retrieved with the following constructors:

```
BL.solve_pde_steady(q_bunch, fb, ffactor);
BL.solve_pde_transient(q_bunch, fb, Nbunches, ffactor);
```

The possible arguments are:

`q_bunch`	Total charge per bunch	[e]
`Nbunches`	Number of bunches of the train	
`ffactor`	Bunch form factor	

The most common bunch distributions and associated form factors are given below:

GAUSSIAN: $$F(\omega) = \exp -\frac{\omega^2 \sigma_z^2}{2c^2}$$

UNIFORM: $$F(\omega) = \frac{\sin x}{x}, \text{ with } x = \frac{1}{2}\omega t_b$$

Here, σ_z refers to the longitudinal single-bunch energy spread, and t_b is the time-range of it. If no `ffactor` is given, the default value is `ffactor = 1`.

Set methods

Two common strategies exist for beam loading compensation: (1) Injecting bunches early in the structure before its RF-filling and (2) optimising the input-power pulse shape in TW structures. Bunch injection and input-power profile can be arbitrarily defined in RF-Track before tracking using the following constructors:

```
% Set unloaded gradient according to Pinput to compensate BL in TW structures
BL.set_unloaded_gradient(Pinput, dt_Pinput, t_inj, RF_Structure );
% Set early injection time to compensate BL in SW structures
BL.set_early_injection(t_inj, RF_Structure );
```

The required arguments are:

`Pinput`	Input power spline	[W]
`dt_Pinput`	Time step associated to the `Pinput` spline	[mm/c]
`t_inj`	Injection time with respect to the origin of `Pinput`	[mm/c]
`RF_Structure`	RF-Element with the TW/SW field map	

Get methods

Some structure-relevant magnitudes, as well as the beam-induced and unloaded gradients, can be retrieved from the constructors as follows:

```
% Structure information
BL.get_Lcell(); % Cell length, m
BL.get_tfill(); % Filling time, mm/c
BL.get_z0(); % Starting longitudinal coordinate, m
BL.get_z1(); % Ending longitudinal coordinate, m
BL.get_wake_function(); % Longitudinal wakefield, V/pC/m

% Interpolated splines of figures of merit along the structure
BL.get_vg(); % Group velocity, c
BL.get_dvg(); % Group velocity spatial derivative, c/mm
BL.get_Q(); % Quality factor
BL.get_rho(); % Normalized shunt impedance per unit length, Ohm/m
BL.get_drho(); % Normalized shunt impedance p.u.l derivative, Ohm/m/mm
```

```
% Structure gradient - After solve methods
BL.get_G(); % Transient beam-induced gradient, V/m
BL.get_G_steady(); % Steady beam-induced gradient, V/m

% Structure gradient - After Set methods
BL.get_G_unloaded(); % Transient unloaded gradient, V/m
BL.get_G_unloaded_steady(); % Steady unloaded gradient, V/m
BL.get_dt(); % Time-step in transient gradient matrices, mm/c
```

The function `BL.get_wake_function()` retrieves the longitudinal wakefield associated with the excitation of the fundamental mode, which is the one on which the implementation of the BL kick is based, as discussed in [LINAC24].

Further longitudinal short-range effects arising from higher-order modes can be simulated with `ShortRangeWakefield`, as presented in 5.6.1. To avoid overlap of the effects, the following constructors allow to enable/disable the short-range BL force:

```
BL.disable_short_range();
BL.enable_short_range();
```

5.5.2 Beam loading in standing-wave structures

The beam loading effect in standing-wave structures for non-ultrarelativistic scenarios is taken into account using the collective effect `BeamLoadingSW()`, which accepts the following arguments:

```
% Transient BL effect in SW structures
BL = BeamLoadingSW(SWS, Q, r_Q, Ncells, mass, q, tinj);
```

In this case, the arguments are:

`SWS`	RF-Element with the SW field map	
`Q`	Loaded quality factor array	
`r_Q`	Normalized shunt impedance per unit length array (r/Q)	[Ω/m]
`Ncells`	Number of cells of the TW structure	
`mass`	Particle mass	[MeV/c^2]
`q`	Single particle charge	[e]
`tinj`	Injection time of the bunch	[τ]

For SW structures, $\tau = \frac{2Q}{\omega}$.

Get methods

The following quantities can be obtained from the `BeamLoadingSW()` collective effect:

```
BL.get_Lcell(); % Cell length, m
BL.get_tfill(); % Filling time, mm/c
BL.get_tinj();  % Injection time, mm/c
```

```
BL.get_TT1(); % Time-transit factor array for the first cell
BL.get_TT2(); % Time-transit factor array for any complete cell
```

5.6 Wakefields

RF-Track implements three wakefield models: `ShortRangeWakefield`, which implements K. Bane's approximation described in [SLAC-PUB-9663, 2003], `LondRangeWakefield` for multi-bunch simulations, and `Wakefield_1d` for user-defines short- and long-range wakefield function provided by the user.

5.6.1 Short-range wakefield

`ShortRangeWakefield` implements the transverse and longitudinal effect of short-range wakefields.

```
SRWF = ShortRangeWakefield (a, g, l);
SRWF = ShortRangeWakefield ([ai af], [gi gf], [li lf], Ncells);
```

In its second form, this effect can simulate cell-to-cell variations such as the tapering of iris apertures. The required input parameters describe the accelerating structure's cell geometry:

a	the average iris aperture radius	[m]
g	the average gap length	[m]
l	the average cell length	[m]
`[ ai af ]`	a 2-element vector with the initial and final iris apertures	[m]
`[ gi gf ]`	a 2-element vector with the initial and final gap lengths	[m]
`[ li lf ]`	a 2-element vector with the initial and final cell lengths	[m]
`Ncells`	The number of cells in the structure	[INTEGER]

The meaning of these quantities is shown in Figure 5.1.

In evaluating the wakefield effect, $W_\perp(s)$ and $W_{||}(s)$ are calculated using the analytic approximation of the normalized wake potential presented in [K.Bane, SLAC-PUB-9663]:

$$w_\perp(s) = \frac{4Z_0 c s_{\perp 0}}{\pi a^4}\left[1-\left(1+\sqrt{\frac{s}{s_{\perp 0}}}\right)\exp\left(-\sqrt{\frac{s}{s_{\perp 0}}}\right)\right] \qquad [\mathrm{V/pC/m/mm}]$$

$$w_{||}(s) = \frac{Z_0 c}{\pi a^2}\exp\left(-\sqrt{\frac{s}{s_{||0}}}\right) \qquad [\mathrm{V/pC/m}]$$

where Z_0 is the impedance of free space, a is the average aperture radius of the structure, g is the gap length, d is the length of the cell, $s_{||0}$ and $s_{\perp 0}$ are

$$s_{||0} = 0.41\,\frac{a^{1.8} g^{1.6}}{d^{2.5}},$$

$$s_{\perp 0} = 1.69\,\frac{a^{1.79} g^{0.38}}{d^{1.17}},$$

respectively. The range of validity of these formulægenerally covers most of the cases.

Inquiring the wake function

The user can inquire RF-Track about the single-particle Wake function using the method:

```
Wl = SRWF.w_long ( S ); % returns V/pC/m
Wt = SRWF.w_transv ( S ); % returns V/pC/m/mm
```

where the input parameter:

S	is the longitudinal distance in meters from the source particle. Since the Wakefield can only affect the trailing particles, `S<0`.	[m]

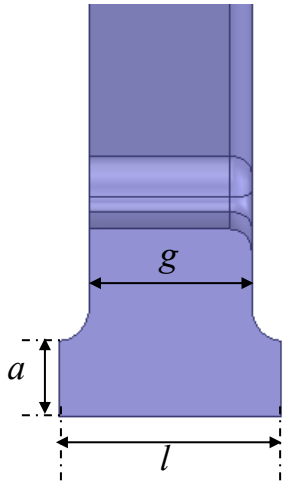


Figure 5.1: The geometric parameters a, g, and l used to describe the short-range wakefield.

Cell-to-cell misalignment

To study the impact of cell-to-cell misalignment, RF-Track can randomly scatter each cell using the method:

```
SRWF.scatter_cells (RANGEX, RANGEY ); % mm
```

The parameters `RANGEX` and `RANGEY` determine the amplitude of the offset in mm. Each call to this function offsets each of the `Ncells` cells of the structure within a range $-$`RANGE`$/2$ and $+$`RANGE`$/2$ according to a uniform random distribution.

5.6.2 Long-range wakefield

In the case of multi-bunch operation, long-range wakefield effects can become significant. In RF-Track, the command `LongRangeWakefield` allows the simulation of long-range wakefield effects. Its input arguments are:

```
LRWF = LongRangeWakefield (A, freq, Q);
LRWF = LongRangeWakefield (A, freq, Q, angle);
```

The required input parameters describe the transverse high-order modes:

`A`	A vector or a matrix containing the modes' amplitudes	[V/pC/mm/m]
`freq`	A vector or a matrix containing the modes' frequencies	[GHz]
`Q`	A vector or a matrix containing the modes' Q factors	[unit less]
`angle`	A vector or a matrix containing the modes' polarization angles. If `NaN`, no polarization is considered, and the kick is given in the x and y directions. (DEFAULT: `NaN`)	[deg]

When `A`, `freq`, and `Q` are matrices, e.g.

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ \vdots & & & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \tag{5.1}$$

the first index, n, runs over the longitudinal axis of the structure, whereas the second index, m, runs over the different modes. Thus, one can simulate a set of modes that varies along the structure.

The transverse dipole-mode wake function implemented is:

$$w_\perp(s) = \sum_n A_n \cdot \exp\left(\frac{\pi s}{Q_n \lambda_n}\right) \sin\left(\frac{2\pi s}{\lambda_n}\right), \qquad [\mathrm{V/pC/m/mm}]$$

with λ_n being the wavelength corresponding to the frequency f_n. The polarization angle indicates the mode's polarization. If `angle=0`, for example, the wakefield will only affect the x axis. If `angle=90`, for instance, the wakefield will only affect the y axis. If `angle=NaN`, the wakefield will equally affect the x and y axes.

Referring to the notation in literature, one can write the coefficients A_n as follows:

$$A_n = 2\left(\frac{R'_x}{Q}\right)_n \frac{4\pi^2 f_n}{4c} = \left(\frac{R'_x}{Q}\right)_n \frac{2\pi^2 c}{\lambda_n^2} \qquad [\mathrm{V/pC/m/mm}]$$

with R'_x is the transverse shunt impedance in Ohm/m. A_n is the wake-function amplitude per unit of length.

Inquiring the wake function

The user can inquire RF-Track about the wake function using the methods:

```
Wx = LRWF.w_transv_x ( S ); % returns V/pC/m/mm
Wy = LRWF.w_transv_y ( S ); % returns V/pC/m/mm
```

where the input parameter S is the longitudinal distance in meters from the source particle. Since the Wakefield can only affect the trailing particles, that is, for $S < 0$. This function returns the sum of all modes averaged over the number of cells.

If no polarization angle is specified, `w_transv_x(s)` and `w_transv_y(s)` return the same value.

5.6.3 Generic wakefield

The collective effect `Wakefield_1d` allows the user to provide an arbitrary wake function, or Green's function, using two 1d vectors sampling the function.

```
WF = Wakefield_1d (Wt, Wl, hz);
```

The required input parameters to describe the wakefield are:

`Wt`	A 1d vector with the transverse component of the wake function	[V/pC/mm/m]
`Wl`	A 1d vector with the longitudinal component of the wake function	[V/pC/m]
`hz`	the 1d mesh spacing of `Wt` and `Wl`	[m]

Inquiring the wake function

The user can inquire RF-Track about the single-particle Wake function using the methods:

```
Wl = WF.w_long ( S ); % returns V/pC/m
Wt = WF.w_transv ( S ); % returns V/pC/m/mm
```

where the input parameter:

S	is the longitudinal distance in meters from the source particle. Since the Wakefield can only affect the trailing particles, `S<0`.	[m]

5.7 Passage of particles through matter

Three complementary effects can be turned on or off independently to simulate the passage of charged particles through matter. The first is multiple Coulomb scattering; the second is stopping power, which accounts for the energy loss described by the Bethe-Bloch formula; and finally, energy straggling, which is the fact that the energy loss isn't the same for all particles.

5.7.1 Multiple Coulomb scattering

The Multiple Coulomb Scattering module allows the simulation of the interaction between the bunch particles and a material. This effect can be added to *any* elements to simulate beam tracking through matter. Follows a list of the constructor and the main methods:

```
M = MultipleCoulombScattering( material );
M = MultipleCoulombScattering( X0, Z, A, density, I=-1 );

M.enable_log_term();
M.enable_fruehwirth_model(); % DEFAULT
M.enable_wentzel_model(); % DEFAULT

M.disable_log_term(); % DEFAULT
M.disable_fruehwirth_model();
M.disable_wentzel_model();
```

The input arguments describing the material are:

`material`	a pre-defined material among: 'air', 'water', 'beryllium', 'lithium', 'liquid_hydrogen', 'titanium', 'tungsten'	[STRING]
`X0`	the radiation length	[cm]
`Z`	the atomic number of absorber	[INTEGER]
`A`	the atomic mass of absorber	[g mol^{-1}]
`density`	the material density	[g cm^{-3}]
`I`	the mean excitation energy. If not provided, RF-Track will compute it using the approximation proposed by Bloch.	[eV]

5.7.2 Stopping power

Charged particles lose energy as they pass through matter, governed by the Bethe-Bloch equation, which describes how the energy loss depends on the properties of the material and the particle. The energy loss per unit path length is defined as

$$-\left\langle \frac{dE}{ds} \right\rangle = 4\pi N_{\mathrm{A}} \frac{\alpha^2}{m_e} \frac{\rho Z}{A} \frac{1}{\beta^2} \left[\frac{1}{2} \ln \frac{2m_e \gamma^2 \beta^2 T_{\max}}{I^2} - \beta^2 \right]. \tag{5.2}$$

In RF-Track, Eq. (5.2) is automatically applied as the default energy loss model when the command `MultipleCoulombScattering(material)` is used. For electrons in water and electrons in air, RF-Track uses the data tabulated by the National Institute of Standards and Technology, `https://physics.nist.gov/PhysRefData/Star/Text/ESTAR.html`, for muons in liquid

hydrogen, RF-Track uses the data tabulated by the Particle Data Group `https://pdg.lbl.gov/2022/AtomicNuclearProperties/MUE/muE_hydrogen_liquid.txt`.

5.7.3 Energy straggling

While a bunch traverses a material, the energy loss is not the same for all the particles. Hence, particles incident on the foil with the same energy emerge with different energies. This energy distribution caused by the scattering material is known as the particles' 'straggling".

6. Inverse Compton Scattering

6.1 Introduction

RF-Track can simulate laser beams. The laser can interact with electrons, positrons, or any other charged particle through Thompson and Compton scattering. The element `LaserBeam` allows the laser beam's initialisation.

RF-Track provides several functions that can fine-tune the laser-beam interaction conditions. Position and angle offsets can be applied to the laser beam, enabling the study of misalignments and other imperfections.

6.1.1 General parameters

Structure `LaserBeam` contains all information required to define a laser beam from its general parameters. The possible arguments are:

```
LB = LaserBeam();
LB.pulse_energy; % mJ, laser pulse energy
LB.pulse_length; % ps, laser pulse length
LB.wavelength; % nm, laser wavelength
LB.length; % m, length of the interaction region
LB.P; % laser polarization, abs(P)<=1 or NaN for unpolarized laser beams
LB.R; % mm, laser sigma spot size, can be Gaussian or tophat profile
LB.Rx; % mm, horizontal sigma spot size
LB.Ry; % mm, vertical sigma spot size
LB.M2 = 1; % laser beam quality factor
LB.rep_frequency = 0; % Hz, the laser repetition frequency
LB.number_of_pulses = 1; % number of pulses in the train
```

The laser beam size can be defined in two ways: for a symmetric laser beam profile, you can use

`LB.R`; if the beam is asymmetric along two orthogonal axes, you can use `LB.Rx` and `LB.Ry` to define the two transverse sizes. Note that `LB.R` is half the $1/e^2$ laser waist radius, w_0.

The parameter `LB.P` allows you to set the polarisation of the laser beam. At the moment, only linear polarisation has been implemented. If the unit vector $\hat{\mathbf{k}}$ defines the polarisation axis of the incoming laser beam in the plane orthogonal to the direction of propagation of the laser, then

$$\hat{\mathbf{k}} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix},$$

and $P = \sin\theta$. The default setting, `LB.P = NaN`, means *unpolarised* laser beam.

Note that the length of the interaction region `LB.length` can be zero, i.e., an interaction point can be inserted as a thin element in a lattice. Even in the case of a zero-length interaction region, RF-Track calculates the laser-beam interaction by reconstructing the full 3D shape of both bunch and laser pulses from the moment of first contact until the moment of separation of the two colliding bunches.

Define the direction of the laser

The direction of propagation of the laser beam is completely arbitrary and is specified as a 3D vector.

```
LB.set_direction(nx, ny, nz);
```

The possible arguments are:

`nx`	horizontal component	[a.u.]
`ny`	vertical component	[a.u.]
`nz`	longitudinal component	[a.u.]

For example, for a full head-on collision, one must set $n_x = 0$, $n_y = 0$, and $n_z = -1$. Notice that the vector defined by n_z, n_y, and n_z doesn't need a unitary length; whichever length, only its direction is considered.

6.1.2 Particles-laser interaction

Distinct functions and parameters define the laser-beam interactions in the `LaserBeam` element.

Define the interaction point

If the length of the `LaserBeam` element is not zero, you need to specify the position of the interaction point between the incoming beam and the laser along the longitudinal axis.

```
LB.set_position(Z);
```

The parameter `Z` must be a number between 0 and the length of the element.

Set number of generated macro-particles per slice

RF-Track implements macro-particles for tracking. The beam(s) generated from interactions are also structured in macro-particles. This function allows you to adjust the number of macro photons generated in each slice to ensure enough macro photons are created in one run to generate statistics. This affects both the precision of the interaction simulation and its runtime.

```
LB.min_number_of_gammas_per_slice(nr_gamma);
```

Where `nr_gamma` is an integer for the number of photon macro-particles simulated per slice.

Define the number of steps taken by the simulation

Tracking through elements is inherently discrete, as the beam is advanced through an element in a discrete number of steps. The number of steps taken across the interaction space can be changed in RF-Track. A large number of steps improves the precision of the simulation but increases the runtime.

```
LB.set_nsteps(nsteps);
```

Where `nsteps` is an integer, the number of steps taken during the simulation across the interaction region. An odd number of steps is recommended to capture the interaction in the central step.

6.2 Collecting the generated photons

When a bunch traverses a `LaserBeam` element, RF-Track computes the Compton scattering between the bunch and the laser. All photons generated during the interaction are added to the travelling bunch. The user can retrieve them using the `Bunch6d`'s method `get_phase_space()`. In particular, inquiring about the mass or the charge of the particles in the bunch allows the discrimination between the bunch's particles and photons. Follows an example:

```
% L is a lattice containing a LaserBeam element
B1 = L.track(B0);

% Read the output phase space
M1 = B1.get_phase_space('%x %xp %y %yp %t %P %m %N');

% Separate the photons from the electrons
is_photon = M1(:,7) == 0; % pick the photons (mass == 0)
M1p = M1( is_photon,:); % photons
M1e = M1(~is_photon,:); % other particles

% Total number of single photons generated
Np = sum(M1p(:,8));

% Plot the XX' photons' phase space
scatter(M1p(:,1), M1p(:,2));
xlabel('x [mm]');
ylabel('x'' [mrad]');
```

7. Bunch Generation at a Photocathode

7.1 Introduction

The `Bunch6dT_Generator` structure provides a flexible framework for defining and simulating the properties of a particle bunch in a photo injector. This data structure is designed to describe the characteristics of a particle bunch in a photoinjector. It allows the user to define the properties of particles and distributions in six dimensions, including spatial, temporal, and energy parameters.

The accurate simulation of space-charge forces during photo emission requires additional attention, especially when space-charge effects, optionally with mirror charges, are considered. This chapter covers these subjects.

7.2 Generator options

7.2.1 Particle properties

- `species` [STRING]: Type of particle (e.g., “electron”).
- `noise_reduc` [BOOL]: Indicates whether noise reduction is active.
- `rand_generator` [STRING]: Random generator type (e.g., “halton”, “sobol”, ...).
- `cathode` [BOOL]: Whether is a cathode or 3D distribution.
- `charge` [REAL]: Charge of the particle.
- `mass` [REAL]: Mass of the particle in MeV/c^2.
- `p_ref` [REAL]: Reference momentum in MeV/c.

7.2.2 Longitudinal distribution

- `dist_z` [STRING]: Type of longitudinal particle distribution (e.g., “gaussian”, “fd_300”).
- `sig_z` [REAL]: RMS value of the bunch length in mm.
- `shape_z` [REAL]: Shape parameter of the generalized Gaussian distribution. (default: 2, Gaussian)
- `c_sig_z` [REAL]: Cutoff for the Gaussian distribution.
- `lz` [REAL]: Length of the bunch in mm.

- `rz` [REAL]: Rising edge of the bunch in mm (for plateau distribution).

7.2.3 Transverse spatial distribution

- `dist_x` [STRING]: Horizontal distribution type (e.g., “gaussian”, “pleateau”, . . .).
- `dist_y` [STRING]: Vertical distribution type (e.g., “gaussian”, “pleateau”, . . .).
- `sig_x` [REAL]: RMS size in the horizontal direction (mm).
- `sig_y` [REAL]: RMS size in the vertical direction (mm).
- `shape_x` [REAL]: Shape parameter for the horizontal distribution. (default: 2, Gaussian)
- `shape_y` [REAL]: Shape parameter for the vertical distribution. (default: 2, Gaussian)
- `c_sig_x` [REAL]: Cutoff for the horizontal distribution.
- `c_sig_y` [REAL]: Cutoff for the vertical distribution.
- `lx` [REAL]: Width of the horizontal distribution in mm.
- `ly` [REAL]: Width of the vertical distribution in mm.
- `rx` [REAL]: Rising edge of the horizontal distribution in mm.
- `ry` [REAL]: Rising edge of the vertical distribution in mm.
- `x_off` [REAL]: Horizontal offset in mm.
- `y_off` [REAL]: Vertical offset in mm.
- `disp_x` [REAL]: Horizontal dispersion in meters.
- `disp_y` [REAL]: Vertical dispersion in meters.

7.2.4 Transverse momentum distribution

- `dist_px` [STRING]: Horizontal momentum distribution type (e.g., “gaussian”).
- `nemit_x` [REAL]: Normalized transverse emittance in the horizontal direction (mm mrad).
- `sig_px` [REAL]: RMS value of horizontal momentum (keV/c).
- `c_sig_px` [REAL]: Cutoff for the horizontal momentum distribution.
- `lpx` [REAL]: Width of the horizontal momentum distribution (keV/c).
- `rpx` [REAL]: Rising edge of the horizontal momentum distribution (keV/c).
- `cor_px` [REAL]: Correlated beam divergence in the horizontal direction (mrad).

7.2.5 Additional parameters

- `emit_z` [REAL]: Longitudinal emittance in keV mm.
- `emit_t` [REAL]: Temporal emittance in keV ps.
- `cor_ekin` [REAL]: Correlated energy spread in keV.
- `e_photon` [REAL]: Photon energy for Fermi-Dirac distribution (eV).
- `phi_eff` [REAL]: Effective work function for Fermi-Dirac distribution (eV).

7.3 Example of Generator

Follows an example:

```
%% Beam creation at cathode
G = Bunch6dT_Generator();

G.species = 'electrons';  % species
G.cathode = true;         % cathode, true or false
G.noise_reduc = true;     % noise reduction (quasi-random)
```

```
G.q_total = Q_pC/1e3;      % nC, bunch charge
G.ref_ekin = 0;            % MeV, energy of reference particle
G.ref_zpos = 0;            % m, position of reference particle
G.ref_clock = 0;           % ns, clock of reference particle
G.phi_eff = 3.5;           % eV, effective work function
G.e_photon = 4.73;         % eV, photon energy for Fermi-Dirac distribution
G.dist_x = 'g';            % Specifies the horizontal particle distribution
G.dist_y = 'g';            % Specifies the vertical particle distribution
G.sig_x = sigr;            % mm, rms bunch size in the horizontal direction
G.sig_y = sigr;            % mm, rms bunch size in the vertical direction
G.sig_t = sigt;            % ns, rms emission time
G.c_sig_x = 3;             % cuts off a Gaussian horizontal distribution
G.c_sig_y = 3;             % cuts off a Gaussian vertical distribution
G.c_sig_t = 3;             % cuts off a Gaussian longitudinal distribution
G.dist_pz = 'fd_300';      % Cathode emission following a Fermi-Dirac dist

Nparticles = 10000;        % number of macroparticles to generate
B0 = Bunch6dT (G, Nparticles);
```

7.4 Photo emission simulation

When simulating photoinjectors, it is important to correctly account for **space-charge effects during the emission process** itself. RF-Track provides two `TrackingOptions` to improve the accuracy of this modelling:

- `emission_nsteps` — Specifies the number of **space-charge kicks** to be applied while the bunch is being emitted from the photocathode. A higher value increases the temporal resolution of the emission process, allowing a more accurate description of the rapidly evolving electromagnetic fields in the early stages of the beam formation. The default value is `10`, which generally offers a good compromise between accuracy and computation time. Increasing `emission_nsteps` may be beneficial in cases of very high bunch charge or extremely short emission times, where space-charge effects evolve rapidly. (Default = 10)
- `emission_range` — Controls how long RF-Track remains in **emission tracking mode** after the nominal emission time has elapsed. This option is given as a **multiplicative factor** of the bunch's emission time. For example, with the default value `10.0`, RF-Track will continue to apply the emission tracking mode for twice the nominal emission duration. This ensures that residual space-charge interactions that are still relevant immediately after emission are correctly taken into account, before switching to the standard tracking mode. Increasing `emission_range` may improve accuracy when simulating beams with long emission tails or when post-emission dynamics are particularly sensitive to space-charge forces. (Default = 10.0)

These options are particularly important for high-brightness photoinjector simulations, where the interplay between emission dynamics and collective effects (in particular space charge and intra-beam scattering) strongly influences the initial beam quality.

7.5 Mirror charges at cathode

As the effect of mirror charges on the cathode is mainly manifested through space-charge effects, mirror charges are simulated in the space-charge module. For details on space-charge algorithms *per se*, see section 5.1. To enable the simulation of mirror charges, it is sufficient to activate a mirror in the space-charge simulation. The following example illustrates how:

```
SC = SpaceCharge_PIC_FreeSpace (32, 32, 32)  % 32x32x32 mesh
SC.set_mirror (0.0)  % set cathode position and activate mirror charges

# Set RF-Track to use 'SC' for space-charge calculations
RF_Track.SC_engine = SC
```

The method `set_mirror` comes in several flavours, allowing the simulation of cathodes placed in any location in space:

```
set_mirror(S);            % longutidinal position of the mirror in m
set_mirror(x, y, z)       % x,y,z in meters, angles in radians
set_mirror(x, y, z, roll, pitch, yaw) % x,y,z in meters, angles in radians
set_mirror(frame_matrix)  % works with Volume::get_s0
```

7.5.1 Misaligned cathode

The last form enables the simulation of a misaligned cathode. In this case, it is useful for the mirror charge position to correspond to the Volume's `s0` surface, which will effectively act as a cathode when a `Bunch6d` is tracked through a Volume, or when a Volume is part of a Lattice.

8. Extending RF-Track

8.1 Introduction

RF-Track provides a set of user-definable constructs designed to make the code extensible and adaptable to advanced simulation needs. These constructs allow users to customize particle tracking behavior, define new field or element types, and introduce new collective effects. They also support the dynamic creation of particles, which enables seamless coupling with external codes such as Geant4, BDSIM, and FLUKA.

The four main constructs introduced for user customization are:

- `UserElement` — a customizable Lattice element.
- `UserEffect` — a customizable collective effect.
- `UserField` — a customizable electromagnetic field.
- `UserVisitor` — a customizable Lattice or Volume visitor.

All four are implemented as Python classes that the user can subclass and customize. The resulting objects can be added to RF-Track's standard simulation chains, both in `Lattice` and `Volume` contexts.

8.2 Custom Lattice Elements with `UserElement`

The `UserElement` construct allows users to introduce a new element into a `Lattice`. This element can implement any desired behavior by defining how the 6D phase space of each particle evolves as it travels through the element.

8.2.1 Key use cases

- Modeling novel accelerator components.
- Prototyping diagnostic elements or correction schemes.
- Implementing custom transfer maps or analytical approximations.

8.2.2 Implementation

To define a `UserElement`, one creates a subclass of `RF_Track.UserElement` and overrides the appropriate method to apply the desired transformation to the particle coordinates.

8.2.3 Example: A Custom Element in Python

The following example defines a simple user element that transports the bunch (in this case, a single particle) and appends a new particle to it.

All user-defined elements must inherit from the Python base class `RF_Track.UserElement` and implement a method:

```
UserElement.track(self, bunch)
```

The required input arguments are:

`self`	The instance of the user-defined element class.
`bunch`	The bunch being tracked through the element (type: `Bunch6d`).

The custom Python class *must* initialize the parent class by calling its `__init__()` method, passing the element length (in meters) as argument.

The input `bunch` can be freely modified within the `track()` method. New particles can be added using the method `bunch.append(new_bunch)`, where `new_bunch` is an instance of `Bunch6d` or an extended phase-space, as described when illustrating the `Bunch6d` constructors, in section 2.2.1.

```
import RF_Track as rft
import numpy as np

class myElement(rft.UserElement):
    def __init__(self, length=0.0):
        super().__init__(length) # Passing the length [m] is mandatory

    def track(self, bunch):
        print('PYTHON::Hi from Python!')
        print(f'PYTHON::bunch.S = {bunch.S} m\n')

        # Modify the phase space of the incoming bunch
        bunch[0].x += 5
        bunch[0].Pc = 50

        # New particles are created through a new phase space
        #  np.array([ %x %xp %y %yp %t %P %mass %Q %N ])
        new_particles = np.array([ 1, 2, 3, 4, 5, 6, rft.muonmass, +1, 1e4]);

        # Add new particles to the current bunch
```

```
        bunch.append(new_particles)

# User-defined element
P = myElement(length=1.0)

# Crate a lattice containing it
L = rft.Lattice()
L.append(P)

# Create one test particle and track it
phase_space = np.array([ 0., 0., 0., 0., 0., 100. ])
B0 = rft.Bunch6d(rft.electronmass, 0.0, -1, phase_space)
B1 = L.track(B0)

# Print results (two particles)
print(B1.S)
print(B1.get_phase_space('%m %x %xp %y %yp')) # mass x xp
```

8.3 Custom Electromagnetic Fields with `UserField`

The `UserField` construct enables the definition of user-defined electromagnetic fields—either static or time-dependent—which can be attached to a `Volume` or a `Lattice` element.

8.3.1 Applications

- Implementing analytical field models.
- Interpolating custom field maps from measurements or simulations.
- Simulating pulsed magnets or time-varying RF fields.

8.3.2 Field Evaluation

To insert a user-defined field, the user needs to define a method to return the electromagnetic field at a given point in space and time. This method is called internally by RF-Track when integrating the equations of motion, or when using `Volume::get_field()` for plots. Fields must have a finite length larger than zero in order to be properly inserted in a RF-Track's Volume or Lattice. The length must be passed to the parent class through its method `__init__(length)`.

Note: Due to the complexity of integrating a user-defined field into the core tracking engine of RF-Track, simulations involving a `UserField` must run in single-threaded mode. As a result, RF-Track cannot utilize multiple CPU cores in this case.

To enforce this, the number of threads must be explicitly set to one at the beginning of each script that uses a `UserField`:

```
import RF_Track as rft

rft.cvar.number_of_threads = 1
```

This limitation may significantly slow down complex or large-scale simulations. Users are advised to isolate user-defined fields to the minimum necessary sections of the beamline.

8.3.3 Example: A Custom Electromagnetic field in Python

The following snippet defines a simple collective effect that applies a constant transverse kick to all particles.

```
import RF_Track as rft
import numpy as np

rft.cvar.number_of_threads = 1  # see text

# Define a user-supplied quadrupole magnetic field
class myQuadrupole(rft.UserField):
    def __init__(self, length=0.0, gradient=0.0):
        super().__init__(length) # Passing the length [m] is mandatory
        self.G = gradient / 1e3  # Convert from T/m to T/mm

    def get_field(self, x, y, z, t):
        print("PYTHON::Hi from Python!")

        # Magnetic field for ideal quadrupole: Bx = G*y, By = G*x
        Bx = self.G * y
        By = self.G * x
        Bz = 0.0

        # No electric field
        E = np.zeros(3)
        B = np.array([Bx, By, Bz])

        return E, B

# Instantiate the custom quadrupole field with gradient 4 T/m
Q = myQuadrupole(length=0.2, gradient=4.0)

# Add the field to a Volume, at position (0, 0, 0)
V = rft.Volume()
V.add(Q, 0.0, 0.0, 0.0)

# Query the field at position (x=0, y=2, z=4) mm and time t=6 mm/c
E, B = V.get_field(0.0, 2.0, 4.0, 6.0)

# Print the field values
print(f"E field = {E} V/m")
print(f"B field = {B} T")
```

8.4 Custom collective effects with UserEffect

The UserEffect construct allows users to define custom collective effects that act on the particle distribution. These effects can be attached to any RF-Track element and evaluated at configurable intervals along the element length.

8.4.1 Capabilities

- Applying custom forces based on the beam distribution.
- Modeling beam–environment interactions (e.g., wakefields).
- Introducing energy loss mechanisms or external feedbacks.

8.4.2 Returning modified bunches

The user-provided function may optionally return a modified bunch. This enables the implementation of complex effects such as energy spread growth, halo generation, or particle-matter interactions with secondary production.

8.5 Secondary particle generation and external interfaces

The three constructs —UserElement, UserField, and UserEffect—, used at runtime during tracking, support the injection of new particles into the simulation. This feature is essential for coupling RF-Track with external codes, for example:

- **Geant4** — for detailed simulation of particle–matter interactions.
- **BDSIM** — for radiation and shielding studies.
- **FLUKA** — for hadronic cascade modeling and energy deposition.

New particles can be added by returning a new Bunch6d or Bunch6dT object from the user-defined function. These secondaries are then added to the bunch and tracked like all other particles, interacting with fields and collective effects.

8.5.1 Example: A Custom Collective Effect in Python

All user-defined collective effects must inherit from the Python base class RF_Track.UserEffect and implement the method compute_force().

```
[force, new_particles] = UserEffect.compute_force(self, bunch, S_mm, dS_mm)
```

The required input arguments are:

self	The instance of the user-defined collective effect class.
bunch	The bunch being subjected to the force. Type: Bunch6d or Bunch6dT.
S_mm	The initial position along the element [mm].
dS_mm	The step length over which the force acts on the bunch [mm or mm/c in case of Bunch6dT].

The method must return two values:

`force`	An $N \times 3$ array, where N is the number of particles in `bunch`. Each row contains the components of the applied force (F_x, F_y, F_z) in units of MeV/m.
`new_particles`	A bunch containing any new particles to be added to the simulation. Type: `Bunch6d` or `Bunch6dT`. Can be empty or omitted.

Note: The incoming `bunch` *must not be modified* directly. Collective effects are intended as *thin* kicks. Although the parameter `dS_mm` must always be provided (and may be used internally by the effect—for example, when computing scattering events or radiation emission). Any new particles should be returned through the `new_particles` object instead.

The following code defines a simple collective effect that applies a constant transverse kick to all particles:

```
import RF_Track as rft
import numpy as np

class mySpaceCharge(rft.UserEffect):
    def compute_force(self, bunch, S_mm, dS_mm):
        # Apply a uniform transverse kick
        N = bunch.size()          # number of particles in the bunch

        # Compute the force acting on the existing particles
        force = np.zeros((N, 3))  # Fx, Fy, Fz
        force[:,0] = 0.1          # Kick in x [MeV/m]
        force[:,1] = 0.1          # Kick in y [MeV/m]

        # Add new particles
        new_particles = rft.Bunch6d([...])

        return force, new_particles
```

This minimal implementation can be extended to include diagnostics, conditionals based on beam shape, or coupling with external codes or data. The optional second return value (here, the unchanged bunch) may be used to append new particles to the beam.

8.6 Traversing Beamlines with `UserVisitor`

RF-Track provides a user-extensible visitor interface that allows custom traversal of a `Lattice` or a `Volume`. This is implemented via the `UserVisitor` class, which follows the classical *visitor pattern*. It enables Python users to define actions that are applied to each element of a beamline during its traversal.

8.6.1 Key use cases

`UserVisitor` can be used to perform read-only inspections, logging, graphical rendering, or even to build auxiliary data structures. Some example applications include:

- Generating a survey report of all elements and their 3D coordinates.

- Drawing a schematic of the beam line for visualization.
- Exporting lattice data to an external format (e.g., JSON, XML).

8.6.2 Implementing a Visitor in Python

To define a custom visitor, the user must subclass `RF_Track.UserVisitor` and implement the method:

```
def visit(self, element):
    ...
```

This method is called by RF-Track during traversal. The parameters are:

`element` The current item being visited.
Type: `Element` or any derived classes.

The user can detect the exact type of each element, using the following construct:

```
if type(element) == rft.Quadrupole:
    print('The element is a Quadrupole')
```

or alternatively

```
if isinstance(element, rft.Quadrupole):
    print('The element is a Quadrupole')
```

A visitor can access and process each object's parameters, fields, or identifiers. The visitor can stop the traversal early by calling the method:

```
my_visitor = myCustomVisitor()
my_visitor.stop_at(element)
```

where `element` is the desired end element in the lattice.

Using the Visitor

Once defined, the user visitor is applied to a `Lattice` or a `Volume` using the `accept()` method. For example:

```
L.accept(my_visitor)
```

RF-Track will traverse the internal structure of the lattice (or volume) and invoke the appropriate method on the visitor for each encountered object.

Example: Printing Element Locations

The following Python example prints the global position of each element in a lattice:

```
import RF_Track as rft
import numpy as np

## Define a FODO cell
...

## Start UserVisitor
class myVisitor(rft.UserVisitor):
    def visit(self, e):
        print('PYTHON::Hi from Python!')

        if type(e) == rft.Quadrupole:
            g = e.get_gradient()
            print(f'PYTHON::This is a quadrupole with gradient = {g} T/m')

        if type(e) == rft.Lattice:
            n = e.size()
            print(f'PYTHON::This is a lattice containing {n} elements')

V = myVisitor()
FODO.accept(V)
```

This mechanism provides a clean and powerful way to extend RF-Track's beamline traversal logic without modifying the core C++ code or duplicating structural logic.

8.7 Getting started

Users are encouraged to begin with the examples provided in the RF-Track Jupyter tutorials directory, and consult this manual's reference sections for further customization details.

9. Examples

9.1 Example of bunch creation

9.1.1 Bunch6d from an arbitrary user-defined distribution

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.

```
% create a bunch of 1e12 100 MeV/c electrons, using 1000 macroparticles
P = 100; % MeV/c total momentum
O = zeros(1000,1); % define a column vector of zeros
I = ones(1000,1); % define a column vector of ones
X = randn(1000,1); % mm, column vector of Gaussian-distributed positions
Y = randn(1000,1); % mm, column vector of Gaussian-distributed positions

% create the beam matrix
M = [ X O Y O O P*I ]; % Bunch6d %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.rft'); % save the beam in RF-Track binary format
B0.save_as_dst_file('my_bunch.dst', 750.0); % save as DST, 750 MHz RF
B0.save_as_sdds_file('my_bunch.sdds', 'my useful comment'); % save as SDDS

% save as an Octave matrix
save -text my_bunch.txt T0;
```

9.1.2 Chirped Bunch6d from Twiss parameters

```
%% Load RF-Track
RF_Track;

%% Beam
Pref = 100; % MeV/c, reference momentum
Q = -1; % electrons
Nparticles = 10000; % 10k particles

%% Twiss parameters
Twiss = Bunch6d_twiss();
Twiss.emitt_x = 1; % mm.mrad normalised emittance
Twiss.emitt_y = 1; % mm.mrad
Twiss.beta_x = 1; % m
Twiss.beta_y = 1; % m
Twiss.sigma_t = 1; % mm/c, bunch length
Twiss.sigma_pt = 5; % permille, energy spread
Twiss.disp_z = 1; % m, longitudinal dispersion (chirp)

%% Create bunch
B0 = Bunch6d(RF_Track.electronmass, 0.0, -1, Pref, Twiss, Nparticles);
```

Index

G

I

L

M

P

S

T

V

W