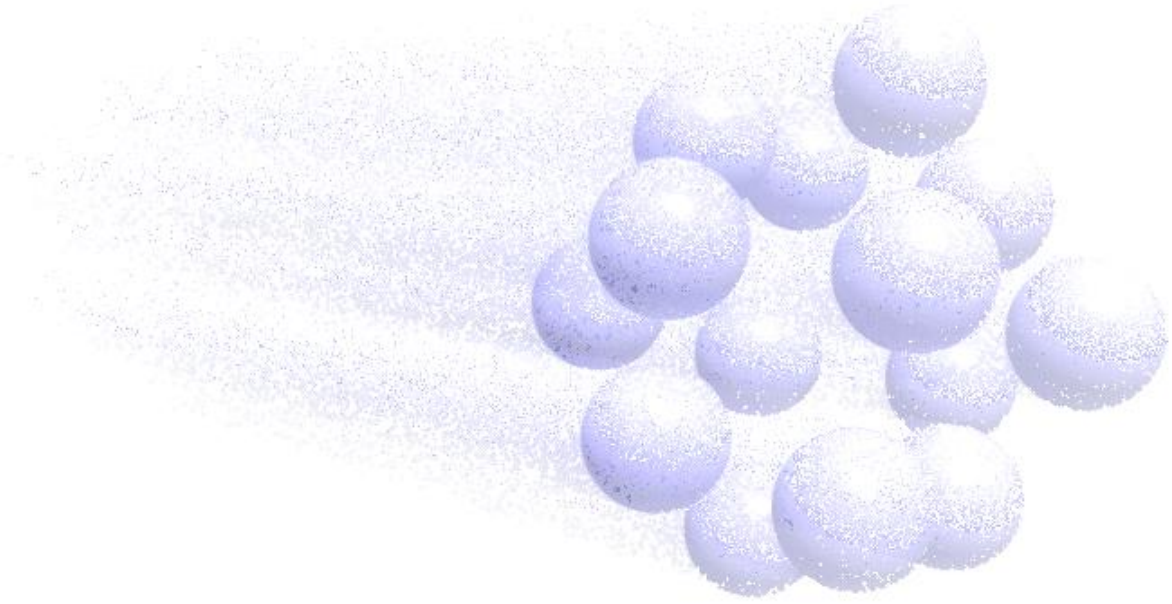
General Particle Tracer



User Manual
Version 3.10

About GPT

The General Particle Tracer GPT is developed, written and maintained by:
Dr. S.B. van der Geer
Dr. M.J. de Loos

For questions about GPT, please contact:
Pulsar Physics tel.: +31 40 2930583
Burghstraat 47 fax: +31 40 2930586
NL-5614 BC Eindhoven info@pulsar.nl
The Netherlands www.pulsar.nl

Acknowledgements

The GPT code could never have matured without the constant flow of suggestions and constructive criticism of the GPT user community. It is slightly unfair to single out individuals, but it would be even more unfair not to mention anyone. We would like to express our thanks to Wim van Amersfoort and Andy Sessler for their confidence in and support of the first version of GPT. Furthermore, we would like to thank Gisela Pöplau for sharing her fast multigrid poisson solver and Marnix van der Wiel for his decade-long support of the GPT project as a whole. Many of the built-in components of GPT were originally developed by GPT users from universities, research institutions and the industry. We would like take this opportunity to thank them all for generously sharing their work with us.

Preface

Particle tracing is a very powerful tool in the design of charged-particle accelerators and beam lines. However, the computers used to be too slow to trace the number of particles needed to obtain reliable statistics in a reasonable amount of computing time. Therefore the particle tracing method was often abandoned and especially matrix/optical methods became popular. These alternative methods however are limited because they fail to simulate an actual beam. Most of these methods yield inaccurate results or cannot be applied at all when space-charge effects are dominant and/or paraxial approximations can not be used.

Currently it is possible to trace millions of particles through complex electromagnetic fields. Because of the enormous advances in computer technology all 3D effects and space-charge forces can be taken into account. This offers much more accurate results compared to matrix and paraxial codes. The General Particle Tracer (GPT) code offers all of these capabilities in a user-friendly and easy-to-customize package.

In order to be able to read this manual, basic knowledge of electromagnetism theory and the special theory of relativity is required. Experience with accelerator or beam-line design is useful, but is not necessary. For information about how to write custom GPT code, we refer to the GPT Custom Elements documentation.

Table of contents

Introduction

The General Particle Tracer (GPT) code is a well-established simulation platform for the study of charged particle dynamics in electromagnetic fields. The code is completely 3D, including the space-charge model. GPT can be conveniently customized without compromising its ease of use, accuracy or simulation speed, because of its modern implementation.

The first chapter of this manual describes the GPT kernel in detail. The second chapter is a tutorial introduction to GPT where simplified but practical examples are discussed. Although we recommend reading the first chapter first, some may prefer to start directly with the tutorial. Chapter three gives an overview of all pre- and postprocessing commands, many of which are also covered in the tutorial. Chapter four describes all the built-in GPT elements and keywords that can be used in the inputfile.

Customizing GPT is the subject of a separate manual, the GPT Programmer's Reference.

Overview

The General Particle Tracer (GPT) is a software package developed to study 3D charged-particle dynamics in electromagnetic fields. Its purpose is to aid in the design of accelerators and beam lines by using modern, particle tracking techniques.

Over the past decade the GPT code has become a well-established simulation tool for the design of accelerators and beam lines. The General Particle Tracer (GPT) is based on 3D particle-tracking techniques, providing a solid basis for the study of all 3D and non-linear effects of charged-particles dynamics in electromagnetic fields. An embedded fifth order Runge-Kutta driver with adaptive stepsize control ensures accuracy while computation time is kept to a minimum. GPT provides various space-charge models, ranging from 1D interaction to fully 3D point-to-point calculations. Most standard beam-line components are represented in GPT. Users of GPT can easily extend the code to perform highly specialized calculations for specific applications. Hierarchical data analysis, automatic parameter scans and graphical output allow for fast and detailed interpretation of simulation results.

The GPT package consists of the GPT executable, which performs the actual calculations, and an extensive set of pre- and post-processing tools including data analysis tools and a graphical user interface.

The GPT executable

A schematic of the GPT executable is shown in the figure below. The following subsections describe the individual components in detail.

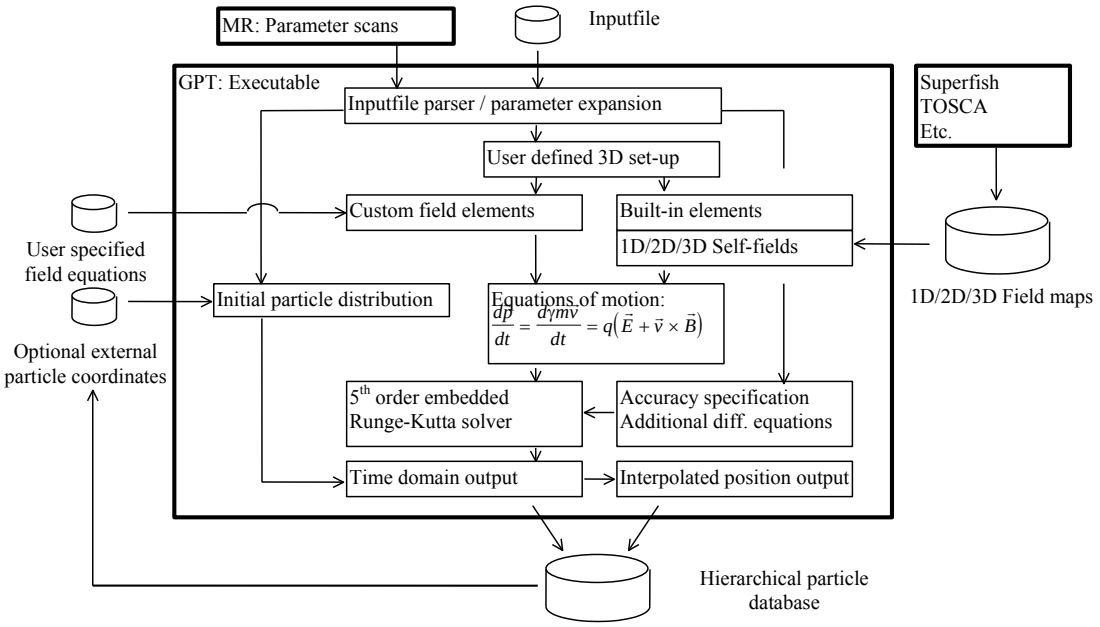


Figure 1: Schematic of the GPT executable.

The GPT executable starts by reading one or more ASCII inputfile(s) describing the simulation to perform. The inputfile specifies the initial particle distribution, the 3D electromagnetic field configuration (set-up), the required accuracy of the calculations and the output method. Standard expressions, functions and user-defined variables can be used for convenience. Optionally, a MR file (Multiple Run) can be used to automatically scan any number of parameters.

The initial particle distribution consists of a number of macro-particles, each typically representing a large number of elementary particles. Hammersley sequences are employed to minimize statistical errors due to the finite number of particles. Each projection of the initial distribution can be specified using the built-in particle generators or can be read from an external file. Complicated particle distributions can consist of any number of separate particle distributions, e.g. a beam with a halo or a mixture of different ion species each having their own distribution in position-momentum space.

The set-up defines the 3D electromagnetic field configuration as generated by the beam-line components. It can be composed of any number of built-in elements, external 2D or 3D field maps and user-defined expressions in custom elements. There is no limit to the number, location and orientation of the elements. The field-map elements can read electromagnetic field configurations calculated by external programs like TOSCA and the SUPERFISH set of codes. Measured field information can be used in the calculations as well. Furthermore, GPT users can easily write their own elements for specific configurations. The interface for writing custom elements is very broad and covers the development of user-defined electromagnetic field configurations, custom initial particle distributions, interfaces to other codes and additional differential equations. All GPT elements are developed in their own coordinate system and written in separate source files. These files are platform independent and allow users to freely exchange elements with each other without having to modify the GPT kernel. When an element is positioned in 3D space, the GPT kernel takes care of all required coordinate transformations.

The self-fields of the particles often form a crucial part of the electromagnetic fields through which these particles are tracked. Space-charge effects can be calculated using various depending on the type of simulation. The 3D space-charge routines include a fully 3D relativistic point-to-point interaction model, a fast hierarchical model and a very fast mesh-based PIC routine based on solving Poisson's equation in the rest frame of the bunch. For the simulation of (semi)continuous beams a 2D space-charge model can be used calculating relativistic point-to-ray interactions. Space-charge effects in cylindrical symmetric beams can be modeled using a relativistic 2D point-to-circle routine.

The equations of motion for the macro-particles are solved relativisticly in the time domain using a 5^{th} order embedded Runge-Kutta integrator with adaptive stepsize control. When required, the individual particle coordinates can be obtained with an accuracy of 10^{-10}. The adaptive timestep mechanism modifies the stepsizes according to the gradients of the electromagnetic fields to ensure that all output satisfies the user-specified accuracy. Optionally, the equations of motion are combined with additional differential equations. This mechanism can be used to calculate beam-loading or FEL interaction completely self-consistently.

GPT has two available output modes: time and position output. Time output writes all particle coordinates at user specified time(s). Position output writes all particle coordinates passing any plane in 3D space. This output mode is also known as 'nondestructive screen' output. Time and position output can be freely mixed and any number of time and position outputs can be specified. Optionally, the electric and magnetic fields at the particle coordinates are also output, which greatly helps in understanding particle dynamics.

Pre- and post-processing

Being able to simulate charged-particle dynamics in time-dependent 3D electromagnetic field configurations is usually far from sufficient for serious accelerator and beam-line design. The simulation data must be analyzed, parameters must be scanned and typically a comparison must be made between different scenarios. GPT is accompanied by a number of pre- and postprocessing programs as well as interfaces to other software packages to ease this step in the design process. These tools are combined with GPT and integrated into the Windows XP/Vista-based graphical user interface, GPTwin. GPT on UNIX machines uses command-line versions of these programs. The typical data flow within GPTwin is shown in Figure 2. The following subsections describe the individual components in detail.

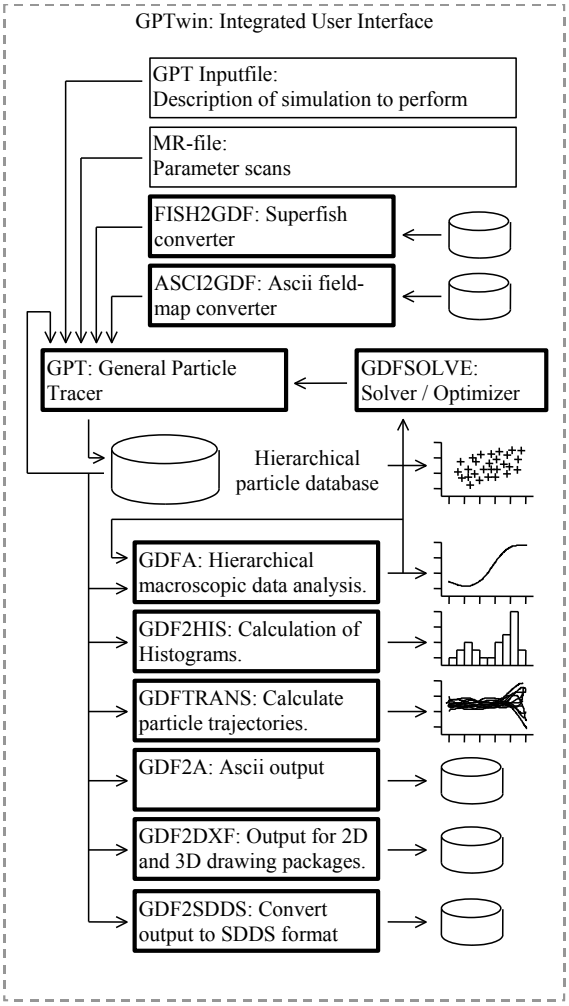


Figure 2: Typical data flow within GPTwin.

GPTwin is the all-integrated Windows XP/Vista-based user interface for GPT. It provides a standard editor with on-line help capabilities for editing the inputfile. Inputfiles can be run and combined with any number of pre- and post-processing tools. GPTwin is able to directly plot the output of GPT as well as the results of data analysis tools like GDFA. Any number of plots can be shown in separate windows within the GPTwin user interface. When more windows show different views of the same outputfile, the hierarchy between the different views always remains synchronized.

The GPT results are written to a binary file for off-line analysis and interpretation. This file is written in the General Datafile Format (GDF), a multi-purpose, hierarchical file format specifically designed for efficiently storing large quantities of numerical data. It allows the post-processing components to easily extract information. The GDF format is used throughout the GPT kernel, the analysis tools and the GPT elements to keep the code compact and efficient. As shown in figure 2, various conversion utilities are available to convert to and from the GDF format.

The main analysis program for GPT output is GDFA. It calculates macroscopic beam parameters as function of simulation time, position or any scanned parameter. Typical macroscopic quantities like emittance, bunch length, average energy and beam radius can be obtained. GDFA is capable of calculating many other useful quantities as well. When needed, the list of GDFA programs can be extended with custom code to calculate the beam parameter of interest. GDFA maintains the original hierarchy in the GDF file when more parameters are simultaneously scanned simultaneously.

The MR and GDFSOLVE utilities can be used to instruct GPT to run a sequence of simulations with one or more parameters varying. The MR program scans over a pre-defined multi-dimensional parameter range while GDFSOLVE automatically optimizes a design until user-defined criteria are met.

1 The GPT code

1.1 The GPT inputfile

The crucial part of every GPT simulation consists of an ASCII inputfile describing the simulation to be performed. Apart from describing the set-up, this file also specifies the initial particle distribution, the calculation accuracy and when/where to write output. If desired, the inputfile can be split separated into a number of individual files. The extension of a GPT inputfile is **.in** by convention. A typical inputfile is shown in Listing 1–1 sending a very basic 6 nC, 2 MeV bunch of electrons through a single solenoid. The formal specification of the GPT inputfile syntax is given in section 4.1.

Listing 1–1: **Experi.in**: A simple GPT experiment.

```
1.  # Basic beam parameters
2.  Eo   = 2e6 ;                        # Energy [eV]
3.  G    = 1+|qe|*Eo/(me*c^2) ;         # Corresponding Lorentz factor G
4.  Beta = sqrt(1-G^-2) ;               # Corrsponding normalized velocity
5.  rxy  = 5e-3 ;                       # Bunch radius [m]
6.  zlen = 1e-3 ;                       # Bunch length [m]
7.  Qtot = -6e-9 ;                      # Total charge in tlen [C]
8.
9.  # Simulation parameters
10. nps  = 50 ;                         # Number of particles
11.
12. # Start bunch
13. setparticles("beam",nps,me,qe,Qtot) ;
14. setrxydist("beam","u", rxy/2,rxy) ;
15. setphidist("beam","u", 0, 2*pi) ;
16. setzdist("beam","u", 0, zlen ) ;
17. setGdist("beam","u", G, 0 ) ;
18.
19. # Space-charge
20. if( Qtot!=0 ) spacecharge3D() ;
21.
22. # Set-up
23. solenoid("wcs","z",0.2, 0.1, 40000) ;
24.
25. # Output
26. tout(0,1.3e-9,0.02e-9) ;
```

The first section of this file defines the basic beam parameters. In this example, the bunch is cylindrically symmetric with an energy of 2 MeV. The bunch length is 1 mm, the beam radius is 5 mm and the total bunch charge is 6 nC. All parameters are in SI units, except when specified otherwise. None of the user variables have a special meaning; They are just for convenience to make the inputfile more readable.

The simulation parameters typically define the number of particles and the simulation accuracy. As the **accuracy** keyword, see section 4.3.2, is not specified in this case, the default timestep mechanism is used.

The electron bunch is started using the start elements as explained in section 1.6. The **setparticles** element fills the particle set **"beam"** with **nps** particles carrying a total charge **Qtot**. The predefined variables **me** and **qe** are the mass and charge of an electron respectively. After the particles are created, the following keywords each modify part of the real- or velocity-space of the set, allowing full control over the initial particle distribution. In this example, the real-space is defined as a uniform cylinder with a 5 mm radius and 1 mm longitudinal length. Transverse velocity is not defined here which means that this component is zero. Longitudinal velocity is introduced by setting the Lorentz Factor γ without spread.

After the initial particle distribution is specified, the space-charge model must be chosen. In this case, a fully 3D treatment is requested with the **spacecharge3D** element as described in section 4.5.3. Because the beam is cylindrically symmetric, the **spacecharge2Dcircle** element, see section 4.5.6, could be used as an alternative. The **if** statement removes the space-charge calculation altogether if the total charge is set to zero.

This set-up consists of only one solenoid, see section 4.7.13, positioned at longitudinal position z=1 m. As explained in section 1.4.2 it is possible to position any element anywhere in 3D space in any orientation. A realistic set-up typically consists of more beam-line components, modeled as built-in elements, field maps or custom elements.

GPT is a time-domain particle tracer, as explained in section 1.7. Therefore, the most natural form of output is a list of all particle coordinates at specific times. This can be specified with a **`tout`** statement. Alternatively, interpolated particle coordinates can be calculated at specific positions using the **`screen`** element. Both forms of output are described in section 1.9.

1.2 Running GPT

GPTwin, the MS-Windows-based user interface for GPT, allows users to run a batch file containing the instructions to run GPT. Error messages and diagnostic output are shown in the message window. UNIX users are also advised to create a batch file, but this is not strictly necessary.

The GPT executable reads the ASCII inputfile(s) describing the initial particle distribution, the calculation accuracy, the locations and parameters of all beam-line components and when/where output is required. Inputfiles typically have extension **`.in`**. The output of GPT is in the General Data Format (GDF), an hierarchical binary database specifically developed for the GPT project. Outputfiles typically have extension **`.gdf`**.

For an experiment named **`experi`**, the GPT inputfile is typically named **`experi.in`**. The correct instruction to run GPT is:

```
gpt [-v] –o experi.gdf experi.in [GPTLICENSE=N]
```

The **`[-v]`** notation means that **`–v`** can be optionally specified. When specified, verbose output is written in the message window in GPTwin and written to the terminal on UNIX machines. On UNIX machines it is necessary to add your GPT license number ***N*** on the command-line to be able to run GPT. On MS-Windows machines the license number is automatically set by the graphical user interface.

This command is typically placed in a batch file named **`experi.bat`**. Running this batch file is identical to specifying the above command. GPTwin users click the Run button on the toolbar. UNIX users specify **`sh experi.bat`**, or any other instruction to run a sequence of commands. Especially when pre- or post-processing tools are required, such a batch file is very convenient. For example, the above GPT command is typically followed by an instruction to calculate particle trajectories and average beam parameters as shown in Figure 1–1:

```
gpt –o experi.gdf experi.in
gdftrans –o traj.gdf experi.gdf time x y z G
gdfa –o avgs.gdf experi.gdf time avgr avgz avgG
```

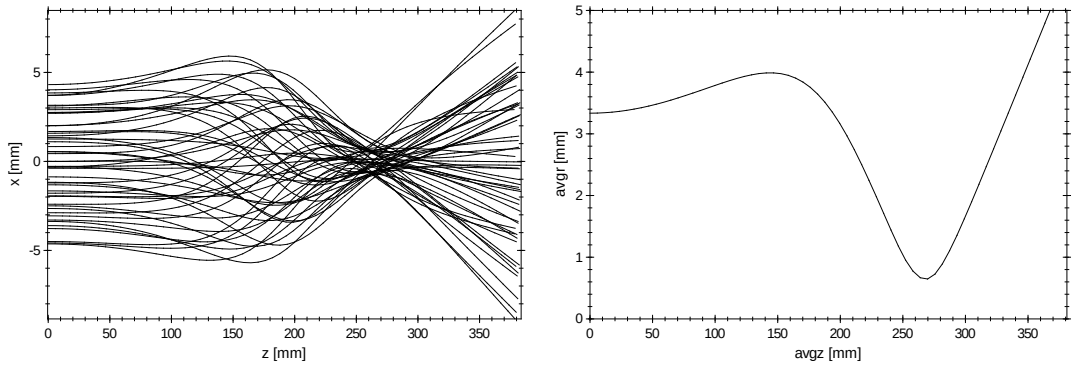


Figure 1–1: Particle trajectories and average beam parameters for the **`experi.in`** file as plotted by GPTwin. The experiment is a very strong solenoid lens at z=200 mm. Strong aberrations are present.

The complete syntax of GPT is:

```
gpt [-hHv] [-o outfile] [-a outfile] infile [var=value] [GPTLICENSE=N]
```

`-h` Print a short usage line.

`-H` Print a usage line and a brief explanation of all options.

`-v` Verbose mode: Various information about the progress of the simulation is written to **`stderr`**. Furthermore, the following numbers are printed at each timestep:

- The number of particles simulated in the current timestep.
- The simulation time *t* [s].
- The timestep *h* [s].

`-j nCPU` Run simultaneously over **`nCPU`** processors in a multi-processor system. When not specified, all processors in the system are used. MS-Windows only.

-o outfile	Write the result of the calculations to **outfile** If not specified output is written to **stdout**. We refer to sections 1.9 and 1.10 for more information about GPT outputfiles.
-a outfile	Append to **outfile**. This option is for internal use only.
infile	The name of the file that describes the experiment to be simulated. A dash ("**-**") can be specified to force GPT to read from **stdin**. It is possible to list more than one inputfile.
var=value	Assign a value to a variable used but not defined in the inputfile.
GPTLICENSE=*N*	Sets the GPT license number to ***N***. UNIX and standalone MS-Windows only.

stderr is Synonym for *Standard error*. This output is shown in the messages window of GPTwin. On UNIX machines, it can be redirected using the operator **>&**. For example:

```
gpt -o test.gdf test.in >& logfile
```

stdout is Synonym for *Standard output*. This output is shown in the messages window of GPTwin, but can also serve as input to other applications using the operator **|**. For example:

```
gpt test.in | gdf2a
```

stdin is Synonym for *Standard input*. It can be the output of some other program using the oprator **|**.as shown in the example above.

1.3 Error messages

A message is displayed when something in the inputfile or in a datafile is incorrect. Such a message typically looks like one of the following lines:

```
filename(linenumber): Warning: warningmessage.
filename(linenumber): Error: errormessage.
```

GPT continues execution after a warning but terminates after an error (with exit code 1). The difference between warnings and errors is sometimes vague and we prefer to give an error during initialization and a warning during simulation. When the program displays nothing, there are no errors or warnings. No news is good news! Continuous information can be specified with the **`-v`** option as described before.

UNIX users can stop GPT during the simulation by pressing the interrupt key, usually Ctrl-C. The current timestep is completed and the program terminates with exit code 2. Results up to that point are saved in the outputfile. If you don't have enough patience to wait for a timestep to be completed you can press the interrupt key again. The timestep is not completed and the outputfile may be corrupted.

Windows-NT users can stop GPT by ending the task with the Task Manager. You can activate the Task Manager by pressing the Delete key while holding down Ctrl and Alt. The current timestep is not saved and the simulation results may be lost.

1.4 Coordinate Systems

1.4.1 Element Coordinate System (ECS)

Elements in the set-up need to be positioned. The parameters of a single-turn solenoid for example are obvious: the radius, the current and the location. The code for a solenoid however is location independent; It is centered around the origin with the z-axis as its normal. The GPT kernel handles all necessary coordinate transforms needed to interface between the code for a solenoid and the actual solenoid positioned in the set-up.

The base coordinate system of GPT, the World Coordinate System or WCS, is a standard orthonormal right-handed Cartesian coordinate system. The GPT kernel assigns a private coordinate system to all elements of the inputfile. This coordinate system, the Element Coordinate System or ECS, is also right-handed orthonormal Cartesian. The various methods to specify such a coordinate system are described in later sections. Because both WCS and ECS are Cartesian, their relation can be defined by an orthonormal matrix **M** and an offset **o** as follows:

$$\mathbf{r}_{WCS} = \mathbf{M}\,\mathbf{r}_{ECS} + \mathbf{o} \qquad [1.1]$$

or

$$\mathbf{r}_{ECS} = \mathbf{M}^{-1}\left(\mathbf{r}_{WCS} - \mathbf{o}\right) \qquad [1.2]$$

where **r** is a coordinate measured in either WCS or ECS.

Because the base GPT coordinate system is WCS, all particle positions are stored relative to this coordinate system. To obtain the electromagnetic fields generated by an element, the following actions are needed:

- Convert the particle coordinates to the ECS using [1.2].
- Calculate the electromagnetic fields using the code of the element.
- Transform these fields back to WCS.

To transform the fields to WCS, the following transformations are used.

$$\begin{aligned}\mathbf{E}_{WCS} &= \mathbf{M}\mathbf{E}_{ECS}\\ \mathbf{B}_{WCS} &= \mathbf{M}\mathbf{B}_{ECS}\end{aligned} \qquad [1.3]$$

Because of the superposition principle, the electromagnetic fields of all the elements in the set-up can be added to obtain the total field at the position of a particle. This can be a time-consuming process when hundreds of elements are present. Therefore the kernel differentiates between two kinds of elements: global elements and local elements.

Global elements have a relatively long working range, such as a solenoid. Therefore the fields of all the global elements are added to obtain the total field generated by the global elements.

Local elements however have a relatively short working range. Furthermore, their ranges are expected not to overlap. Because of this, when the kernel has found a particle to be in the range of one local element, all the other local elements are neglected. To reduce CPU time, the kernel keeps track of the local element a particle is in. Because of this optimization all elements not global by nature should be local.

1.4.2 Specifying an Element Coordinate System (ECS)

The first parameter of almost all GPT elements is the Element Coordinate System. For example, the syntax of a solenoid is:

```
solenoid(ECS,R,I) ;
```

The specification of an ECS consists of the following two parts:

- The name of the coordinate system relative to which the element is located. This usually is `"wcs"`, but can also be the name of a previously defined Custom Coordinate System, CCS, see section 1.4.3.
- The offset **o** and matrix **M** as defined in [1.1].

Various methods are available for specifying an ECS. They all specify **o** and **M**, but the simple methods have restrictions. The transformation most commonly used is the "`z`" transformation. It moves an element in the z-direction without performing any rotations. An other useful coordinate system the "`I`" transformation, the identity transformation. All methods available to specify an ECS are listed in Table 1-A.

Table 1-A: Element coordinate system specifications

ECS specification	**o**	**M**
`"name","I"`	(0,0,0)	Identity
`"name","z",oz`	(0,0,`oz`)	Identity
`"name",ox,oy,oz,xx,xy,xz,yx,yy,yz`	(`ox`,`oy`,`oz`)	See below

The columns of the matrix **M** are the unit vectors of the coordinate system to be specified. Those unit vectors, named **x**, **y** and **z**, are calculated from the specified parameters by:

$$\begin{aligned}\mathbf{x}' &= (\mathtt{xx},\mathtt{xy},\mathtt{xz})\\ \mathbf{y}' &= (\mathtt{yx},\mathtt{yy},\mathtt{yz})\\ \mathbf{x} &= \frac{\mathbf{x}'}{|\mathbf{x}'|}\\ \mathbf{y} &= \frac{\mathbf{y}'-\mathbf{x}'\cdot\mathbf{y}'}{|\mathbf{y}'-\mathbf{x}'\cdot\mathbf{y}'|}\end{aligned} \qquad [1.4]$$

By normalizing the **x**' to obtain the unit **x** vector and by subtracting the **x** part from **y**' while normalizing, the specified vectors need not to be of unit length nor orthogonal. The **z**-direction is calculated by the cross product:

$$\mathbf{z} = \mathbf{x}\times\mathbf{y} \qquad [1.5]$$

1.4.3 Custom Coordinate System (CCS)

When a large set-up needs to be modeled, additional coordinate systems can be defined. Consider for example a number of beam-line components downstream a bending magnet. Determining the position and orientation of the elements after the first bending magnet is difficult in the WCS system. Especially in the design phase of a project when elements are relocated on a regular basis, specifying the element positions relative to a custom coordinate system can be convenient. In our example, the needed coordinate system would have a z-axis in the direction of the beam after the bend and an origin at the intersection of the axes.

Any number of additional coordinate systems can be defined provided they are orthonormal and Cartesian. The coordinate transformations relating the particle coordinates between WCS and a Custom Coordinate System, CCS, are similar to [1.1] and [1.2]. See section 4.11.1 for the reference of CCS.

The ECS to WCS transforms are calculated by the GPT kernel by multiplying the transform from WCS to CCS and the transform from CCS to ECS:

$$\begin{aligned}\mathbf{M} &= \mathbf{M}_{CCS}\mathbf{M}_{ECS}\\ \mathbf{o} &= \mathbf{M}_{CCS}\mathbf{o}_{ECS} + \mathbf{o}_{CCS}\end{aligned} \qquad [1.6]$$

They are used as in [1.1] and [1.2].

Because matrix-vector multiplications are costly in terms of CPU time, they are not performed for identity transformations. The code checks for these identity transformations during the initialization, thus reducing the coordinate transformation overhead for straight beam-line sections completely.

1.5 Space charge

The GPT code has a number of built-in elements to calculate Coulomb interactions, each optimized for a different application. All routines have certain characteristics that make them suitable for a subset of parameter space, but they can be very inefficient or even wrong if used incorrectly. To obtain correct simulation results, it is important to understand the assumptions made in every routine before selecting one. A summary of all built-in space-charge models in GPT is listed in Table 1-B.

Reference section 4.5 of this manual describes the inputfile syntax and the equations of each routine individually. This section focuses on the differences between the various models and gives guidelines for selecting one. Independent on the model selected, we strongly advise to regularly scan the number of particles using the MR/GDFA program combinations and to test convergence by plotting the parameters of interest as function of the logarithm of the number of particles.

Table 1-B: Summary of all GPT space charge elements. Complexity is in terms of the number of particles *N*.

Name	Complexity	Granularity effects	Dim	Description
spacecharge3Dmesh	O($N^{1.1}$)	No	3D	PIC in rest frame
spacecharge3Dtree	O(N log N)	Yes	3D	Barnes-Hut in rest frame
spacecharge3D	O(N^2)	Yes	3D	All pair-wise relativistic interactions
spacecharge3Dclassic	O(N^2)	Yes	3D	All pair-wise interactions
spacecharge2Dcircle	O(N^2)	No	2D	Cylindrically symmetric
Spacecharge2Dline	O(N^2)	No	2D	Continuous beam

All routines in GPT dealing with Coulomb interactions are derived from the electric field at the position of particle *i* due to Coulomb interaction with all particles *j*:

$$\mathbf{E}_i = \sum_{j\neq i} \frac{q_j}{4\pi\varepsilon_0} \cdot \frac{\mathbf{r}_j - \mathbf{r}_i}{\left|\mathbf{r}_j - \mathbf{r}_i\right|^3} \qquad [1.7]$$

Space charge is usually defined as the collective effect only, governed by long-range interactions with charge *density*. The summation hereby changes into an integration. This implies a smooth, fluidlike charge distribution, where the point-like nature of charge can be ignored. In that case, assuming the electrostatic case only, Coulomb interactions can be rewritten in terms of Poisson's equation:

$$-\nabla^2 V = \frac{\rho}{\varepsilon_0} \qquad [1.8]$$

where ρ is the charge density and V the electrostatic potential. This equation is numerically much easier to solve compared to the previous point-to-point calculation. However, it is plainly wrong in the case where granularity/stochastic effects play a dominant role. For example, random emission processes start bunches with excess electrostatic potential energy, which is subsequently converted via Coulomb interactions into additional momentum spread (heat). This process is not included in Poisson's equation. To determine if stochastic effects are important, one can use the following argument: A particle bunch with number density n [m^{-3}] and no initial momentum will heat up to a temperature T_f given by:

$$kT_f = \frac{q^2}{4\pi\varepsilon_0 a} \qquad [1.9]$$

where $a=(4\pi\ n/3)^{-1/3}$ is the Wigner-Seitz radius. This final temperature is reached on a timescale of the order of the inverse plasma frequency $\omega_p^{-1} = (m\varepsilon_0/nq^2)^{1/2}$. Consequently, stochastic effects are important if the initial temperature of the bunch is below T_f, and if the total simulation time is larger than ω_p^{-1}. For typical thermionic and photo-emission processes this is typically not at all a concern.

We realize that the title of this section is confusing because the term space charge is used to denote all Coulomb interactions, including stochastic effects. For historical reasons the confusion is also present in the names of the GPT elements, all having **`spacecharge`** in their names. Some of them include granularity effects, some don't.

The **`spacecharge3Dmesh`** routine, see section 4.5.1, is GPT's workhorse space-charge routine. It is based on solving Poisson's equation in 3D in the rest frame of the bunch. It is by far the fastest space-charge

model in the GPT code, but unsuitable for the calculation of granularity/stochastic effects. The method scales $O(N)$ in terms of CPU time and allows millions of particles to be tracked on a normal PC due to a tailor-made non-isotropic multi-grid solver
A single Lorentz transformation is used to convert the calculated electrostatic fields in the co-moving frame into both electric and magnetic fields for the tracking engine. Consequently, the model cannot be used if there is so much energy spread that velocities in the zero-momentum frame approach the speed of light. This is very unusual.

`The spacecharge3Dtree` routine, see section 4.5.2, calculates *all* Coulomb interactions from first principles, thus including space charge *and* granularity effects. It employs a Barnes-Hut method to hierarchically group particles together. This reduces the computational effort from $O(N^2)$ to $O(N \log N)$, but the prefactor is considerably larger compared to the mesh version described above. Use this routing if granularity effects are expected to be significant.
Analogous to the PIC **`spacecharge3Dmesh`** method, a single Lorentz transformation is used to calculate the magnetic field in the laboratory frame.

The **`spacecharge3D`** element also calculates *all* Coulomb interactions from first principles. However, **`spacecharge3D`** uses Lorentz transforms for *each* pair-wise interaction, where **`spacecharge3Dtree`** transforms the bunch as a whole (analogous to the PIC **`spacecharge3Dmesh`** method). This approach makes this routine the 'best' with respect to physics included, but the drawback is significant: The computational cost is $O(N^2)$, compared to $O(N \log N)$ for **`spacecharge3Dtree`**, limiting the amount of particles that can be tracked in practice to the order of a few thousand. Use this routine for cases with several ten percent relative energy spread, where particles are relativistic in the co-moving frame. Because of its 'from-first-principles' approach, this routine provides a great benchmarking opportunity for both **`spacecharge3Dmesh`** and **`spacecharge3Dtree`** for people with patience.

The **`spacecharge3Dclassic`** routine, see section 4.5.4, models non-relativistic point-to-point interaction. Because this routine scales $O(N^2)$ in terms of CPU time its use is restricted to simulations with just a few particles. In all other cases, **`spacecharge3Dtree`** is the best choice.

The **`spacecharge2Dcircle`** routine, see section 4.5.6, is a cylindrically-symmetric version of **`spacecharge3D`**. It also scales $O(N^2)$ in terms of CPU time, but because it uses one dimension less, the number of particles required before convergence is reached is reduced significantly. Longitudinal Lorentz transformations are applied to every binary interaction, allowing bunches with very large longitudinal energy spread to be simulated correctly. Bunches with extremely high azimuthal velocity, or relativistic transverse velocities cannot be modeled.

The **`spacecharge2Dline`** routine, see section 4.5.8, is a version of **`spacecharge3D`** written for continuous beams. It also scales $O(N^2)$ in terms of CPU time, but far less particles are required compared to full 3D modeling with any of the other space-charge models. Although no Lorentz transformations are applied, the model is relativistically correct. Bunches with large longitudinal velocity spread and extreme divergence cannot be modeled.

The GPT Custom Elements documentation describes how to extend GPT with a custom space-charge model in case none of the built-in routines are suitable for your application.

1.6 Initial particle distribution

Before GPT can start tracking the particle trajectories, the initial particle distribution must be specified. The initial particle distribution contains the initial position, momentum, mass, charge and release time of all particles. The start elements provide a very flexible method to start all kinds of initial particle distributions, ranging from GeV positron beams to less than eV electrons leaving a photo-cathode surface. Different ion species with different phase-space distributions can be specified using the same elements. All space-charge calculations are compatible with multiple particle species.

The GPT code uses particle-sets to store any number of initial particle distributions. Each set contains a number of initial particles and each set can be freely manipulated by the start elements before the simulation starts. For example, particles can be added to a set and the initial positions of the particles in a set can be modified according to a specified distribution. Different sets can contain different particle species, providing precise control over the initial distribution per particle species.

The philosophy behind the start elements is to use a number of keywords to specify the initial particle distribution, where each keyword only sets a certain projection of the distribution. For example, **`setzdist`** modifies the z-distribution of the particles, withot affecting the xy-position coordinates and the velocity distribution.

It is possible to specify a uniform, Gaussian, linear, quadratic or cosine distribution for all one-dimensional projections of the initial phase-space distribution. Alternatively, if none of these distributions is applicable, it is possible to read the distribution from a separate file.

In two dimensions, a Gaussian distibution in the *xy*-plane can be obtained by specifying a Gaussian distribution in *x* with **`setxdist`**, followed by a Gaussian distribution in *y* with **`setydist`**. The result is shown in the left plot of Figure 1–2. However, in general a transverse distribution can not be set by setting the *x*- and *y*-projections independently. For example a uniform hard-edged cylinder in the *xy*-plane can not be produced by any combination of *x* and *y*-projections. Such transverse distributions require **`setrxydist`** to specify the distribution of particle radius and **`setphidist`** to set the angular distribution. To obtain a uniform distribution in the *xy*-plane, the radial distribution must be uniform between 0 and the beam radius, whereas the angular distribution must be uniform between 0 and 2π. The result is shown in the right plot of Figure 1–2.

It is important to note that a uniform transverse (Cyl) distribution differs from a uniform one-dimensional (1D) distribution. To obtain a uniform distribution in the *xy*-plane, a linearly increasing number of particles must be launched as function of radius. In both cases, the user must specify a Uniform distribution, and GPT takes care of the difference.

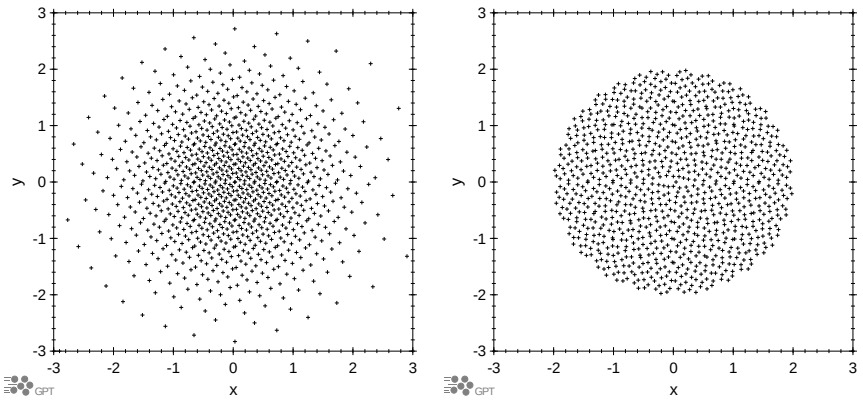


Figure 1–2: Gaussian distribution in the xy-plane (left) and uniform distribution in the xy-plane (right) with Hammersley sequences.

It is apparent that the points in Figure 1–2 are not drawn randomly from the underlying distribution. The reason for this is the following: Typically, GPT is uses less particles in the simulations than are present in the actual experiment. Statistical fluctuations of a random creation process, also known as shot noise, scale as $N^{-1/2}$ with N being the number of particles. Consequently, if both the actual and the simulated particles

are created randomly, then the statistical noise of the small number of simulated particles is much larger than the actual shot noise. This is not desirable and GPT uses Hammersley sequences [0F1] to artificially reduce the shot noise to $1/N$. This results in ordered patterns that provide a good tradeoff between 'smooth' distributions in both real and Fourier space. These low-discrepancy sequences reduce CPU time because less particles are needed compared to a random distribution for the same final accuracy.

It is important to realize that these Hammersley sequences should *not* be used when the number of macro-particles in the simulation matches the number of actual particles in the experiment. The combination of random starting conditions and one of the point-to-point space-charge models is essential for example for the investigation of disorder-induced heating, trajectory displacement and Boersch effect. To illustrate the difference, Figure 1–3 shows the same distributions as shown in Figure 1–2 but now without making use of Hammersley sequences.

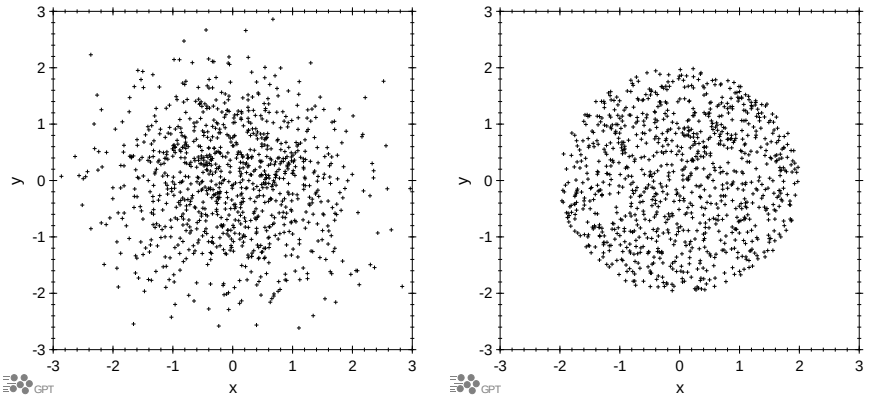


Figure 1–3: Gaussian distribution in the xy-plane (left) and uniform distribution in the xy-plane (right) drawn from a random distribution.

The Hammersley sequence samples N points with an r-dimensional uniform distribution quasi-randomly as:

$$x_i = \{(i-\tfrac{1}{2})/N, \Phi_2(i), \Phi_3(i), \ldots, \Phi_r(i)\} \text{ where } i=1\ldots N. \quad [1.10]$$

The function $\Phi_r(i)$ is the radical inversion function in the base of prime number r. It is obtained by first inverting the base r representation of an integer i, given as

$$i = a_0 + a_1 r + a_2 r^2 + \ldots \quad [1.11]$$

and then computing

$$\Phi_r(i) = a_0 r^{-1} + a_1 r^{-2} + a_2 r^{-3} + \ldots \quad [1.12]$$

Each coordinate of a Hammersley sequence is uniformly distributed between 0 and 1. To prevent correlations, every GPT element modifying a particle distribution uses the next prime number for the generation of the Hammersley sequence. To further illustrate the Hammersley sequences, the top row of Figure 1–4 shows the particle histograms for a Gaussian distribution centered around 0 with a sigma of 1 sampled with 1000 particles. The first plot shows a distribution based on the first equidistant Hammersley sequence, resulting in a perfect histogram. The next plots show the next Hammersley sequences, calculated from the prime bases 3, 5 and 7. The bottom row of Figure 1–4 shows the corresponding correlation between particle position and particle ID. The first plot shows the ascending equidistant case. The next plots show the typical 'interference' pattern between position and particle ID, getting slowly more profound for each next sequence. For a particle tracking application with typically 6 or 7 dimensions, this poses no problem.

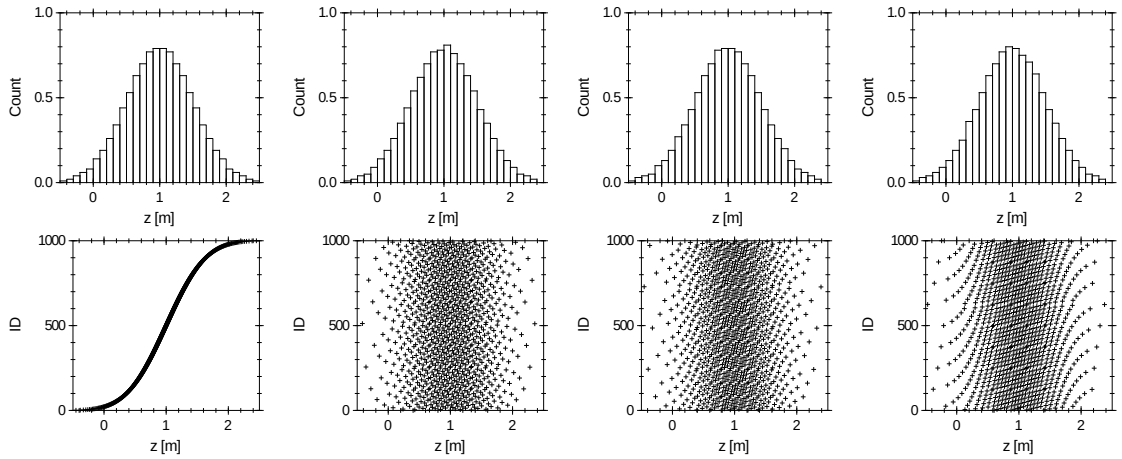


Figure 1–4: Histograms (top) and position versus particle ID correlation (bottom) for a Gaussian distribution based on the first four Hammersley sequences. The first equidistant and the next sequences based on prime numbers 3,5 and 7 are shown.

1.6.1 Syntax

Before a specific particle distribution can be defined, first the number and type of particles must be specified using the **setparticles** keyword:

```
setparticles(set,N,m,q,Qtot) ;
```

The keyword **setparticles** defines a particle set name, for example **"beam"**, and generates **N** particles with all initial coordinates at zero. The total charge **Qtot** is uniformly distributed among all particles. To start 1000 electron macro-particles at the origin without velocity having a total charge of 100 pC one could use the line:

```
setparticles("beam",1000,me,qe,-100e-12) ;
```

me and **qe** are built-in constants having the mass and charge of an electron in [kg] and [C] respectively.

After all these 1000 particles are initialized, the initial distributions can be specified. This is done using a number of keywords, each setting only a particular aspect of the distribution. For example the longitudinal particle distribution, the *z*-coordinates of the particles, can be specified using the **setzdist** keyword.

```
setzdist(set,distribution) ;
```

The particle set must be **"beam"** again to match the **set** parameter of **setparticles**. The distribution is specified as a character indicating the distribution type, followed by its specific parameters. Examples are:

```
setzdist("beam","u",1,2) ;
```
Uniform centered on *z*=1 m having width 2 m.

```
setzdist("beam","g",1,1,2,2) ;
```
Gaussian centered on z=1 m, sigma=1 m and cut-off left and right at 2 sigma.

To switch off the Hammersley sequences, the distribution specifier must be prepended by a tilde "**~**" character. This results in a set of pseudo-random points, where it should be noted that each GPT run uses the same random number sequence. The **randomize** keyword must be used explicitly to instruct GPT to use a new random number sequence at each run.

The plots in Figure 1–2 are created using:

```
sigma=1 ;
setxdist("beam","g",0,sigma,3,3) ;
setydist("beam","g",0,sigma,3,3) ;
```

and

```
R=2 ;
setrxydist("beam","u",R/2,R) ;
setphidist("beam","u",0,2*pi) ;
```

The plots in Figure 1–3 are created using:

```
sigma=1 ;
setxdist("beam","~g",0,sigma,3,3) ;
setydist("beam","~g",0,sigma,3,3) ;
```

and

```
R=2 ;
setrxydist("beam","~u",R/2,R) ;
setphidist("beam","~u",0,2*pi) ;
```

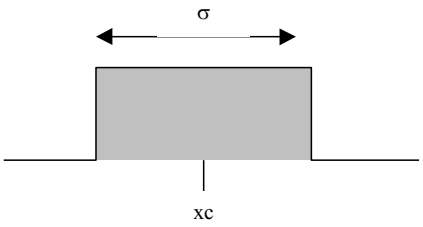
Table 1-C lists the possible distribution types and their parameters. The columns 1D, Cyl and Sphere correspond to Cartesian, Cylindrical and Spherical coordinates used in some of the start elements.

Table 1-C: Distributions, availability and parameters

Specifier	Type	1D	Cyl	Sphere	Parameters
U	Uniform	√	√	√	xc, width
L	Linear	√	√		xc, width, hstart, hend
Q	Quadratic	√			xc, width
C	Cosine	√			xc, width, angle
G	Gaussian	√	√		xc, sigma, sleft, sright
S	Sphere		√		xc, half-width
F	File	√	√		"filename", "x", "P(x)", scale, offset

1.6.2 Uniform distribution: "U",xc,width

The uniform distribution is independent of its variable *x* in the interval: $x \in \left[xc - \frac{1}{2}\sigma, xc + \frac{1}{2}\sigma\right]$:

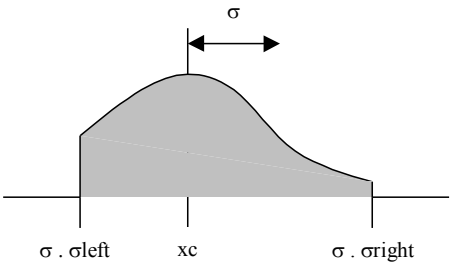


"U"	Uniform distribution.
xc	Center of the distribution.
σ	Width of the distribution.

The distribution is implemented for simple 1D, cylindrical and spherical distributions.

1.6.3 Gaussian distribution: "G",xc,sigma,sleft,sright

The Gaussian distribution is Gaussian in its dependent variable *x* in the interval:
$x \in \left[xc - \sigma\sigma left, xc + \sigma\sigma right\right]$:

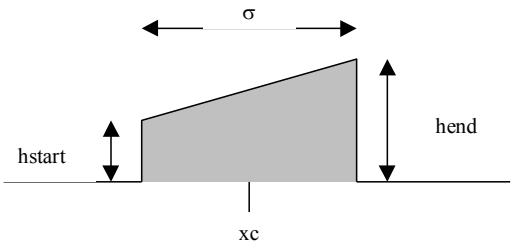


"G"	Gaussian distribution.
xc	Center of the distribution, must be 0 for a cylindrical distributions.
σ	Typical width of the distribution.
$\sigma left$	Left span of distribution relative to σ. A value of 10 effectively makes this parameter ∞. This parameter must be 0 for a cylindrical distribution.
$\sigma right$	Right span of distribution relative to σ. A value of 10 effectively makes this parameter ∞.

The distribution is implemented for simple 1D and cylindrical distributions.

1.6.4 Linear distribution: "L",xc,width,hstart,hend

The linear distribution is linear in its dependent variable *x* in the interval: $x \in \left[xc - \frac{1}{2}\sigma, xc + \frac{1}{2}\sigma\right]$:

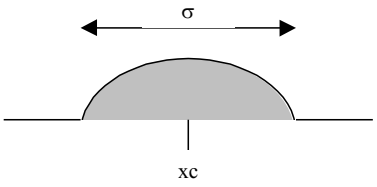


"L"	Linear distribution.
xc	Center of the distribution.
σ	Width of the distribution.
hstart	Start value of the distribution at $x = xc - \frac{1}{2}\sigma$.
hend	End value of the distribution at $x = xc + \frac{1}{2}\sigma$.

The distribution is implemented for simple 1D and cylindrical distributions.

1.6.5 Quadratic distribution: "Q",xc,width

The quadratic (parabolic) distribution is quadratic in its dependent variable *x* in the interval: $x \in \left[xc - \frac{1}{2}\sigma, xc + \frac{1}{2}\sigma\right]$:

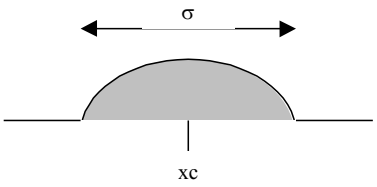


"Q"	Quadratic (parabolic) distribution.
xc	Center of the distribution.
σ	Width of the distribution.

The distribution is implemented for simple 1D distributions.

1.6.6 Cosine distribution: "C",xc,sigma,alpha

The quadratic distribution follows a cosine in its dependent variable *x* in the interval: $x \in \left[xc - \frac{1}{2}\sigma, xc + \frac{1}{2}\sigma\right]$:

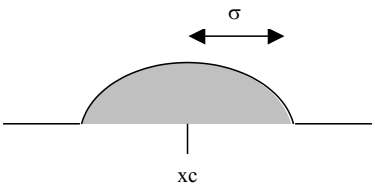


"C"	Cosine distribution.
xc	Center of the distribution.
σ	Width of the distribution.
α	Opening angle [rad] between 0 and 2π selecting what part of the cosine to use.

The distribution is implemented for simple 1D distributions.

1.6.7 Sphere distribution: "S",xc,width

The sphere (half circle) distribution follows a half-sphere in its dependent variable *x* in the interval: $x \in \left[xc - \sigma, xc + \sigma\right]$:

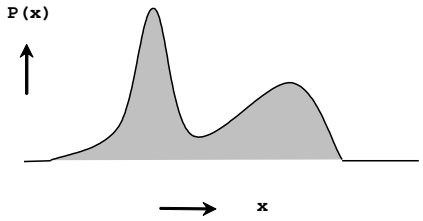


"S"	Sphere (half circle) distribution.
xc	Center of the distribution, must be 0 for a cylindrical distribution.
σ	Half-width of the distribution.

The distribution is only implemented for cylindrical distributions.

1.6.8 File distribution: "F","filename","x","P(x)",scale,offset

The file distribution reads a probability curve from a user-specified GDF file. This is particularly useful if the distribution can not be set by one of the built-in functions listed above.



"F"	File distribution.
"filename"	Name of the GDF file containing the tabulated probability curve.
"x"	Name of the array containing the *x* variables.
"P(x)"	Name of the array containing the probability for each *x* value.
scale	Scale for the *x* values. Typically 1.
offset	Offset for the *x* values. Typically 0.

The probability distribution is interpolated linearly between the specified (*x*,*Px*) points on the probability curve. The *x*-coordinates are scaled by *x*·**scale**+**offset** to accommodate a parametric linear coordinate transform. If this is not desired, set **scale** to 1 and **offset** to 0.

The *x* array can be of arbitrary length, where all *x* values must be in ascending order. The probability *P*(*x)* array must be of equal length and negative probabilities are not allowed. It is the responsibility of the user to specify sufficient points to model the desired curve with sufficient accuracy.

The distribution is implemented for 1D distributions and cylindrical distributions.

Example: Triangular *z*-distribution.
To set a triangular distribution between 4 and 8 for the initial z-coordinates, first a GDF file containing such a distribution must be created. The easiest way to do so is to first create an ASCII file containing the desired distribution **dist.txt** and subsequently convert this file to the GDF format. The latter can easily be done with the ASCI2GDF utility program in a batch file:

```
asci2gdf -o dist.gdf dist.txt
```

```
1. z    Pz
2. 4    0
3. 5    1
4. 6    2
5. 7    1
6. 8    0
```

Listing 1–2: File **dist.txt** specifies a triangular distribution between 4 and 8.

The produced **dist.gdf** can subsequently be used to set the desired distribution in a GPT inputfile:

```
setzdist("F","dist.gdf","z","Pz",1,0) ;
```

1.6.9 Advanced syntax

To distribute a number of points along a given distribution $f(x)$, first the cumulative distribution $F(x)$ is defined by:

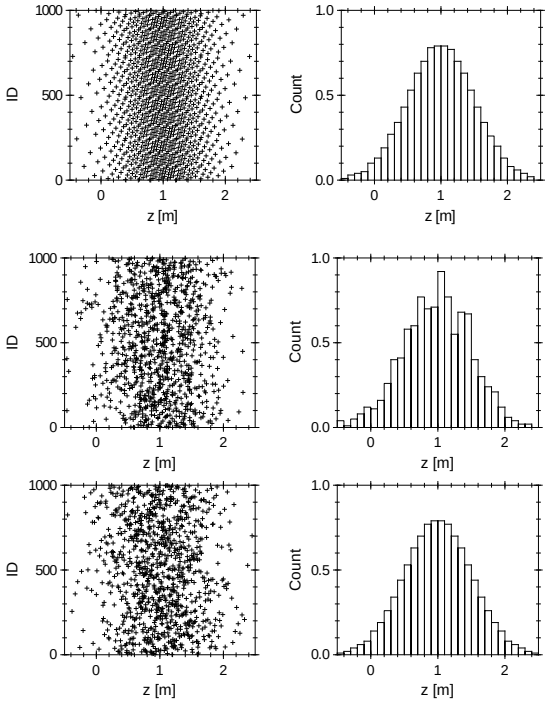
$$\begin{aligned} F(x) &\propto \int_{x'=-\infty}^{x} f(x')dx' && \text{For 1D distributions} \\ F(r) &\propto \int_{r'=0}^{r} f(r')2\pi r\,dr' && \text{For cylindrical distributions} \\ F(\theta) &\propto \int_{\theta'=0}^{\theta} f(\theta')\sin(\theta')d\theta' && \text{For spherical distributions} \end{aligned} \qquad [1.13]$$

In all cases, F is normalized by imposing $F(\infty)=1$. Statistically it can be shown that F^{-1} maps a homogeneous distribution in the interval between 0 and 1 to a distribution following f.

The default GPT approach is to apply F^{-1} to a Hammersley sequence between 0 and 1. This results in a quasi-random set of low discrepancy points, following the distribution f. Alternatively, F^{-1} can be applied to a number of points randomly distributed in the interval between 0 and 1. This results in a pseudo-random set of points, as can be obtained by prepending the distribution specifier with a tilde "**~**" in the GPT inputfile. These two options are shown in the top rows of Table 1-D, where a guassian distribution is approximated by 1000 points. Despite the low number of sample points, the Hammersley sequence results in a smooth histogram. The random distribution shows the actual fluctuations of 1000 randomly positioned particles.

Also shown in Table 1-D are three other methods to create a distribution. When a star "*****" is prepended to the distribution type, the distribution is created by applying F^{-1} to a randomized set of equidistant points. When a plus "**+**" respectively minus "**–**" sign is prepended to the distribution type, the same distribution is added in ascending respectively descending order. The latter two options result in either a monotonically rising or falling correspondence between the coordinate of interest and the particle number. The **+** order is identical to the first Hammersley sequence.

Table 1-D: Example of distribution modifiers.



Specifier: **h** or none
Example:
```
setzdist("beam","g",1,.5,3,3) ;
```

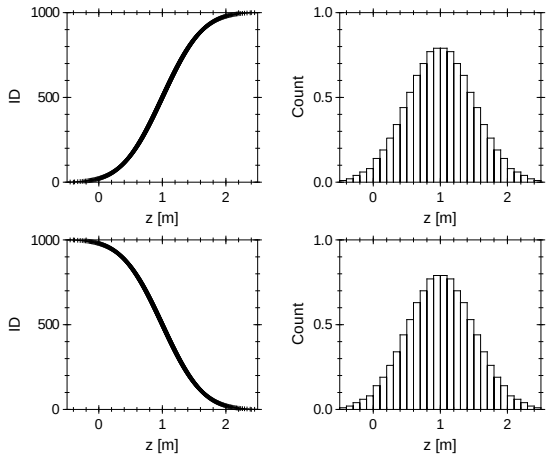
Description: Hammersley order following a specified distribution.

The 3rd sequence with prime base 5 is shown.
Specifier: **~**
Example:
```
setzdist("beam","~g",1,.5,3,3) ;
```

Description: Fully random set of points following a specific distribution.

Specifier: *****
Example:
```
setzdist("beam","*g",1,.5,3,3) ;
```

Description: Random order of an optimized sequence of points following a specified distribution.



Specifier: **+**

Example:

```
setzdist("beam","+g",1,.5,3,3) ;
```

Description: Ascending order of an optimized sequence of points following a specified distribution.

Specifier: **–**

Example:

```
setzdist("beam","-g",1,.5,3,3) ;
```

Description: Descending order of an optimized sequence of points following a specified distribution.

1.7 Equations of motion

The relativistic equations of motion of the individual macro-particles are solved by GPT. The position $\mathbf{x}$ and the momentum $\mathbf{p} = \gamma m\mathbf{v}$ are used as the coordinates of a particle. The equations of motion for particle i are given by:

$$\frac{d\mathbf{p}_i}{dt} = \mathbf{F}_i \qquad \frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i = \frac{\mathbf{p}_i c}{\sqrt{\mathbf{p}_i^2 + m_i^2 c^2}} \qquad [1.14]$$

The force on particle i is calculated using:

$$\mathbf{F}_i = q\left(\mathbf{E}_i + \mathbf{v}_i \times \mathbf{B}_i\right) \qquad [1.15]$$

These equations of motion can not be solved for each particle individually because, due to the spacecharge, the force on each particle depends on the position of all other particles. Therefore we introduce the vector $\mathbf{y}(t)$ containing the six coordinates of all the particles as a function of time t. The equations of motion [1.14] can then be rewritten as:

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}\left(t, \mathbf{y}(t)\right) \qquad [1.16]$$

where $\mathbf{f}\left(t, \mathbf{y}(t)\right)$ is a combination of the equations [1.14] and [1.15] for every particle. The only boundary conditions are the initial particle coordinates at a specified time.

When additional differential equations are added, the variables are added to $\mathbf{y}(t)$ and the differential equations are added to $\mathbf{f}$. The extra differential equations can be a function of all the particle coordinates, time and the variables of other differential equations. The boundary condition, the initial value of the variables, must be supplied for each value individually. Because GPT solves particles coordinates and extra variables simultaneously, the results are always self-consistent. The method used to solve [1.16] is described in the following section.

1.8 Runge-Kutta

In this section we describe the method GPT uses to solve equation [1.16]. This method is known as fifth-order embedded Runge-Kutta with adaptive stepsize control. The algorithm is efficient and has the capability to specify the accuracy of the calculations. For a detailed description we refer to Numerical Recipes [1F2] but we will briefly explain the algorithm here.

Basically the algorithm consists of a fifth-order Runge-Kutta step. It advances $\mathbf{y}(t)$ to $\mathbf{y}(t+h)$ by performing the following steps:

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t,\mathbf{y}) \\
\mathbf{k}_2 &= h\mathbf{f}(t+a_2h,\mathbf{y}+b_{21}\mathbf{k}_1) \\
&\ldots \\
\mathbf{k}_6 &= h\mathbf{f}(t+a_6h,\mathbf{y}+b_{61}\mathbf{k}_1+\ldots+b_{65}\mathbf{k}_5) \\
\mathbf{y}(t+h) &= \mathbf{y}(t)+c_1\mathbf{k}_1+c_2\mathbf{k}_2+c_3\mathbf{k}_3+c_4\mathbf{k}_4+c_5\mathbf{k}_5+c_6\mathbf{k}_6+O(h^6)
\end{aligned} \qquad [1.17]$$

The values of the constants *a*, *b* and *c* can be found in Table 1-E. To be able to monitor the accuracy of $\mathbf{y}(t+h)$ each step is calculated again using an embedded fourth-order formula:

$$\mathbf{y}^*(t+h)=\mathbf{y}(t)+c_1^*\mathbf{k}_1+c_2^*\mathbf{k}_2+c_3^*\mathbf{k}_3+c_4^*\mathbf{k}_4+c_5^*\mathbf{k}_5+c_6^*\mathbf{k}_6+O(h^5) \qquad [1.18]$$

The error estimate $\boldsymbol{\Delta}$ is given by:

$$\boldsymbol{\Delta}\equiv\mathbf{y}(t+h)-\mathbf{y}^*(t+h)=\sum_{i=1}^{6}(c_i-c_i^*)\mathbf{k}_i \qquad [1.19]$$

It is an indicator of the error made in the timestep. It is this difference that is kept to a specified degree of accuracy. Not too large, because this indicates an insufficient precision of the result. Not too small either, because a higher accuracy than needed is a waste of CPU time. Keeping the difference to the specified accuracy is done by adjusting the stepsize *h*.

$\boldsymbol{\Delta}$ is a vector of all the coordinates of all the particles. The maximum absolute values for the position and momentum coordinates are defined as x_{max}. and $mc\gamma\beta_{max}$ respectively. They are compared with x_{err} and $\gamma\beta_{err}$, set in the in the inputfile to define the desired accuracy. The error indicator Λ is defined as the largest of the two following ratios:

$$\Lambda=\max\left(\frac{\gamma\beta_{max}}{\gamma\beta_{err}},\frac{x_{max}}{x_{err}}\right) \qquad [1.20]$$

If $\Lambda>1$, thus $\gamma\beta_{max}>\gamma\beta_{err}$ or $x_{max}>x_{err}$, the step is not accepted and will be retried with:

$$h_{new}=\begin{cases} S(\Lambda)^{\frac{1}{4}}h_{old} & \text{if } S(\Lambda)^{\frac{1}{4}}>0.10 \\ 0.10h_{old} & \text{otherwise} \end{cases} \qquad [1.21]$$

where h_{new} is chosen in such a way that for the next timestep Λ will approximately equal the safety parameter *S*. *S* is default set to 0.85. The timestep can decrease by no more than a factor of 10.

If $\Lambda\leq1$ the step is accepted with $\mathbf{y}(t+h)$. The next timestep is performed with:

$$h_{new}=\begin{cases} S(\Lambda)^{\frac{1}{5}}h_{old} & \text{if } S(\Lambda)^{\frac{1}{5}}<5.0 \\ 5h_{old} & \text{otherwise} \end{cases} \qquad [1.22]$$

This ensures that no more than a factor of 5 increase in the stepsize is possible.

Table 1-E: Cash-Karp constants for embedded Runge-Kutta method.

i	a_i	b_{ij}					c_i	c^*_i
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j=$		1	2	3	4	5		

1.9 Output

The output of GPT mainly consists of the position and normalized velocity coordinates of all the macro-particles and optionally the electromagnetic fields at the positions of the particles. The output is written in GPT Datafile Format, GDF, explained in detail in section 1.10. Three output mechanisms are currently implemented: Snapshot, Position and Field output.

The GPT code solves the differential equations of motion in time-domain. Consequently, the most efficient output mechanism is output at specific simulation times. This can be done with the **`snapshot`** keyword that writes the phase-space coordinates of all particles to the GPT outpfile at arbitrary simulation time. Because the **`snapshot`** keyword does not affect the stepsizes taken by the ODE solver it produces no overhead, apart from disk I/O, and it does not affect the overall accuracy. It syntax is described in detail in section 4.3.11.

To obtain electromagnet field information, output can be generated with the **`tout`** keyword. This writes the electromagnetic fields at all particle positions to the outputfile. Internally, **`tout`** instructs the GPT ODE solver to decrease its stepsizes in such a way that the particle coordinates are calculated exactly at the specified times. The used algorithm looks ahead and starts to make small adjustments to the stepsizes ahead of the requested output time in order to end correctly. As a result, **`tout`** slows down the ODE solver. Furthermore, because stepsizes can be decreased but are never increased, **`tout`** increases the overall accuracy of the simulations. Relying on such a side-effect is not wise and we stongly suggest using the accuracy keyword to specify the overall simulation accuracy. For the exact syntax of the **`tout`** keyword please see reference section 4.3.15.

Position output can be requested using the **`screen`** keyword. The particle coordinates are recorded when they pass a specified non-destructive plane in 3D space. Any number of planes with any orientation can be specified. Starting from GPT version 3.10, the screen implementation in GPT is 'perfect' in the sense that the interpolation error is always smaller or equal compared to the tracking error. Screens need to buffer particle coordinates and involve high-order interpolations. Consequently they are slower and more resource intensive compared to snapshots. For more information we refer to reference section 4.3.10.

The coordinates of all the particles and optionally the electromagnetic fields are written as individual arrays in the GPT outputfile. For example, **`x`** is an array of all the x-coordinates of all the particles. A description of the arrays written to the outputfile is given in Table 1-F. How the outputfile is structured is explained in the next section.

Table 1-F: Description of the particle array columns in the GPT outputfile.

Name	Snapshot	Screen	Tout	Description
`ID`	√	√	√	Particle identification numbers
`x`	√	√	√	*x* coordinate [m]
`y`	√	√	√	*y* coordinate [m]
`z`	√	√	√	*z* coordinate [m]
`Bx`	√	√	√	Normalized velocity βx
`By`	√	√	√	Normalized velocity βy
`Bz`	√	√	√	Normalized velocity βz
`G`	√	√	√	Lorentz factor γ
`rxy`	√	√	√	Distance to z-axis [m]
`t`		√		Time when particles cross the screen [s]
`fEx`			√	*Ex* field at the particle coordinates [V/m].
`fEy`			√	*Ey* field at the particle coordinates [V/m].
`fEz`			√	*Ez* field at the particle coordinates [V/m].
`fBx`			√	*Bx* field at the particle coordinates [T].
`fBy`			√	*By* field at the particle coordinates [T].
`fBz`			√	*Bz* field at the particle coordinates [T].

Within a user element, the GPT kernel can be instructed to call a specified routine at every time output. Typically this mechanism is used to write the parameters of custom differential equations in the outputfile,

but it can also be very helpful for debugging purposes. For more information about this feature, we refer to the GPT Custom Elements documentation.

At the end of the simulation the variable **`cputime`** is written. It contains the CPU time, measured in seconds, between the previous and current time output. It can be used to track where most of the CPU time is spent. The variable **`cputime`** not listed in a time group is the total CPU time used by GPT.

1.10 GDF

The raw output of GPT consists of arrays containing the coordinates of the macro-particles and the electromagnetic fields. To efficiently accommodate additional output variables and parameter scans, the General Datafile Format (GDF) was developed. A GDF file consists of a number of GDF blocks, where every block contains binary data with a name. The *x*-coordinates of the particles for example are all written in a single block named **`x`**. Because all particle coordinates need to be written for every time output and every screen output, the GDF database is hierarchical. The blocks can be grouped together, where the groupname is itself a GDF block with name and data. For time output, the block name is **`time`** and the data is the simulation time itself. The GDF equivalent of such a structure is shown in Figure 1–5. Screen output is analogous, with block name **`position`** and the data the *z*-coordinate in the local coordinate system of the screen.

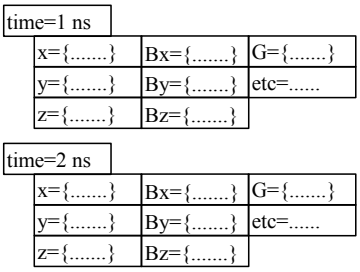


Figure 1–5: GDF representation of the outputfile. The blocks represent a GDF-object and its data. The indentation represents the internal hierarchy.

Although studying particle coordinates at consecutive output times can be very useful, generally macroscopic quantities like bunchlength, emittance and standard deviations are needed as well. The GDFA program is written to extract such information directly from the GPT outputfile.

When the effect of any parameter needs to be studied, the Multiple Run (MR) facility can be used. MR instructs GPT to run with one varying parameter and creates one outputfile containing the results of all the simulations. Such an outputfile is schematically shown in Figure 1–6 where the effects of different settings for the variable phi are calculated.

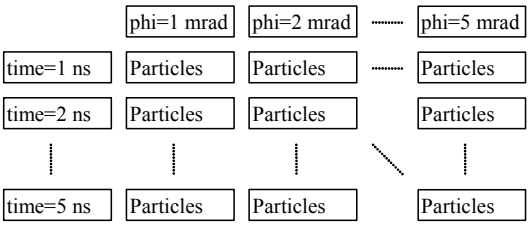


Figure 1–6: MR Output when multiple time output is written for various settings of the parameter **`phi`**.

GDFA needs extra information before such a file can be processed. When for example the average *z*-coordinate versus the standard *z* deviation, a measure for bunch length, need to be studied, there are two possibilities.

1. For every value of phi: Calculate average z and bunchlength. Every point is calculated at a different time.
2. For every time: Calculate average z and bunchlength. Every point is calculated for a different setting of **`phi`**.

One option is not arguably better than/preferable over the other, as depends on what needs to be studied. Therefore GDFA needs an additional parameter, which is named **`group_by`**. This parameters pecifies which level of hierarchy collapses. The first of the two options mentioned above groups by **`time`**, the second groups by **`phi`**. The **`group_by`** parameter itself is always output as an array in the produced outputfile.

For a complete description of all the GDF programs we refer to chapter 3. Table 1-E lists the description of the names in the GPT outputfile

1.11 Collector design

GPT can be used for the design of collectors and other beam-line elements in which scattering on the surface has to be taken into account. The main difference between collector design and normal beam-line simulations is found in the fact that scattered particles, generated when a particle hits a surface, play an important role.

The scattering process is modeled as two separate steps in GPT:

- Boundary elements calculate if, where and at what angle a particle hits a surface. Many basic surfaces like plates, cones, spheres, pipes, irises and toruses are built into GPT and custom shapes can be defined when needed.
- The physical properties of the surface material are defined by a scatter element. When a hit is detected, the corresponding scatter element removes the incident particle and optionally creates one or more new scattered particles with an energy and angle distribution as function of the incident energy and angle.

The separation between boundary elements and scatter elements allows the user to apply any material to any boundary.

The GPT kernel stores all raw scatter statistics to the GPT outputfile for subsequent analysis. The position, energy, charge and angle of incidence of both the incoming and scattered particle are recorded. For the design of a collector consisting of several plates, the FISHFILE program described in section 3.4 reads the raw scatter statistics and writes the statistics per collector plate. The electrostatic field inside the collector is typically calculated by Superfish and imported in GPT as a 2D field-map. To keep the GPT inputfile and the Superfish file consistent, it is possible to generate the boundary description required in the GPT inputfile automatically from the Superfish file.

1.11.1 Boundary elements

GPT boundary elements, like a hollow pipe or sphere, calculate 2D surfaces in 3D space. They determine if a particle trajectory crosses the boundary and, if so, calculate the intersection point, angle of incidence and interpolated momentum. The applied scatter model, as described in section 1.11.2, determines the fate of the incident particle. Typically it is removed and optionally (back) scattered particles are generated.

A 2D boundary in 3D space can be defined by:

$$f(x,y,z)=0 \qquad [1.23]$$

For example, the boundary of a hollow sphere with radius r is:

$$f(x,y,z)=x^2+y^2+z^2-r^2=0 \qquad [1.24]$$

All boundary elements implemented in GPT and their corresponding boundary equation are listed in section 4.10. The cone, iris, pipe, sphere and torus elements are used to model cylindrically symmetric geometries. The plate element can be used to model asymmetric geometries.

To detect if a particle trajectory crosses a boundary, the trajectory is first approximated by straight line segments between consecutive successful Runge-Kutta steps. As an optimization, the bounding-boxes of all elements are first used to determine if a line segment makes any chance of crossing the boundary. For example, a line segment running from (x_1,y_1,z_1) to (x_2,y_2,z_2) with positive z_1 and positive z_2 can never intersect with a plate at $z=0$ and a line with $x_1>R$ and $x_2>R$ can never intersect with a pipe with radius R. If the bounding-box tests does not rule out an intersection, the line segment is parameterized to a single variable λ:

$$\mathbf{P}(\lambda)=\begin{pmatrix} x_1+\lambda\,\Delta x \\ y_1+\lambda\,\Delta y \\ z_1+\lambda\,\Delta z \end{pmatrix} \qquad [1.25]$$

where $\mathbf{\Delta}=(x_2-x_1,y_2-y_1,z_2-z_1)$. To stay between the start- and endpoint of the line segments, λ must always be between 0 and 1. Solving the intersection point $\mathbf{P}(\lambda)$ is identical to solving λ from:

$$f(\mathbf{P}(\lambda))=0 \text{ where } 0<\lambda<1 \qquad [1.26]$$

In the case of a sphere, this yields the following second-order equation:

$$a\lambda^2 + b\lambda + c = 0 \qquad [1.27]$$

where

$$\begin{aligned} a &= \Delta x^2 + \Delta y^2 + \Delta z^2 \\ b &= 2\left(\Delta x\, x_1 + \Delta y\, y_1 + \Delta z\, z_1\right) \\ c &= x_1^2 + y_1^2 + z_1^2 - r^2 \end{aligned} \qquad [1.28]$$

We could have expected the boundary test for a sphere to be a second-order equation because a line and a hollow sphere can have a maximum of two intersections. Using the same argument, testing a plate intersection results in a first-order expression and a torus intersection test requires solving a fourth-order equation.

When more boundaries are present, they must all be tested. When more boundaries are crossed, the boundary with the smallest positive λ is the nearest and the one actually hit. When the intersection point of the nearest boundary is calculated, the surface normal **n** at the intersection point **P** is determined using the gradient of *f*:

$$\mathbf{n}(x, y, z) = \nabla f \qquad [1.29]$$

When the surface normal is known, the angle of incidence α is given by:

$$\alpha = \operatorname{acos}\left(\frac{\mathbf{n}\cdot\boldsymbol{\Delta}}{|\mathbf{n}|\,|\boldsymbol{\Delta}|}\right) \qquad [1.30]$$

1.11.2 Scatter elements

Whenever a boundary element detects a particle hitting its surface, the corresponding scatter element is called into action. This element defines the physical properties of the surface material. It is responsible for removing the incident particle and optionally generating one or more scattered particles. Just like regular GPT elements, a scatter element can have one or more parameters allowing a broader range of materials to be modeled using the same element.

Apart from removing the incident particles and optionally generating one or more scattered particles, scatter elements also write detailed information of every particle-boundary intersection to the GPT outputfile. These 'hits' are recorded for direct plotting, or for further analysis with the FISHFILE program described in section 3.4. Table 1-G gives the scatter arrays written in the GPT outputfile.

Table 1-G: Description of the scatter arrays in the GPT outputfile.

Name	Description
scat_x	Array of the *x* coordinates of the 'hits' [m]
scat_y	Array of the y coordinates of the 'hits' [m]
scat_z	Array of the z coordinates of the 'hits' [m]
scat_Qin	Corresponding total charge of incident particle [C]
scat_Qout	Corresponding total charge of scattered particle(s) [C]
scat_Qnet	**scat_Qin–scat_Qout**
scat_Ein	Corresponding total kinetic energy of incident particle [J]
scat_Eout	Corresponding total kinetic energy of scattered particle (s) [J]
scat_Enet	**scat_Ein–scat_Eout**
scat_inp	Cos(angle of incidence)

The simplest GPT scatter element is forwardscatter, described in section 4.10.1. It scatters a particle in the forward direction with user-defined probability *P*. A reflected macro-particle is generated with the same energy as the incident particle in the forward direction. The new number of elementary particles this macro-particle represents is *P*N*, where *N* is the number of elementary particles represented by the incident particle. When *P*N* is smaller than a user-defined threshhold the new particle is not created to avoid the simulation running forever. A more realistic scatter element for a copper surface is described in section 4.10.2.

1.12 GDFsolve

GDFsolve is a multidimensional root finder and optimizer that can be used as a 'driver-program' for all GPT simulations. When GDFsolve is used as root finder, it tries to solve any number of constraints on beam parameters by varying variables used in the GPT inputfile. The used method allows a non-equal number of variables and constraints as well as external boundary conditions for all variables. When GDFsolve is used as optimizer, it tries to minimize or maximize the weighted sum of any number of selected beam parameters by varying variables used in the GPT inputfile. The optimizer can be combined with the root finder to form a constrained optimizer.

The beam parameters are calculated by standard GDFA data-analysis following a GPT run, allowing both built-in and custom GDFA programs to be used as constraints. Any number of variables in the GPT inputfile can be varied by GDFsolve while trying to find a solution.

GDFsolve is not guaranteed to find an existing solution or minimum, simply because such algorithms do not exist. Furthermore it can easily be fooled into a local minimum and should therefore be used with care. Understanding the physics of the underlying system and good starting values are essential for a proper use of GDFsolve.

The following sections describe the internal algorithms of GDFsolve. More practical information about the use of GDFsolve can be found in tutorial section 2.4 and reference section 3.11.

1.12.1 Mathematical description of the root finder

GDFsolve used as root finder tries to find a simultaneous solution to the following set of non-linear equations:

$$\begin{aligned} f_1(x_1, x_2, \ldots) - ft_t &= 0 \\ f_2(x_1, x_2, \ldots) - ft_2 &= 0 \\ &\vdots \end{aligned} \qquad [1.31]$$

Or in vector notation:

$$\mathbf{f}(\mathbf{x}) - \mathbf{ft} = \mathbf{0} \qquad [1.32]$$

The very simple vector notation in [1.32] is misleading because the components of **f** can represent completely unrelated physical quantities. Solving them simultaneously is in general a very difficult problem.

When the dimensions of **f** and **x** are equal, a relatively simple solution can be obtained by writing a multi-dimensional Taylor-series expansion of **f** around a start value $\mathbf{x}_0$:

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \left.\frac{d\,\mathbf{f}}{d\,\mathbf{x}}\right|_{\mathbf{x}_o} (\mathbf{x} - \mathbf{x}_0) \qquad [1.33]$$

Subsequent solving of [1.33] provides a first order estimate for a new trial $\mathbf{x}_{n+1}$ based on previous guess $\mathbf{x}_n$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{M}^{-1}(\mathbf{ft} - \mathbf{f}(\mathbf{x}_n)) \text{ where } M_{ij} = \frac{d\,f_i}{d\,x_j} \qquad [1.34]$$

Iterating the above procedure is known as multidimensional Newton-Raphson and works very well in the vicinity of a root. As shown in Figure 1–7, the derivative of the function is extrapolated to produce the next trial. Sufficiently near a root convergence is quadratic, i.e. the number of significant digits in **x** doubles at each step.

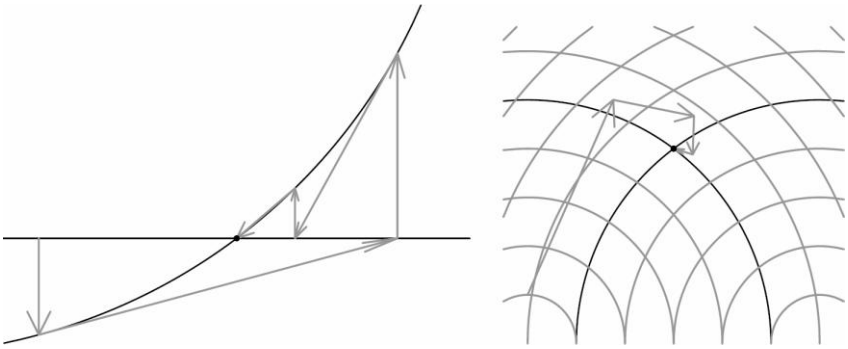


Figure 1–7: Typical Newton-Raphson iteration sequence in one and two dimensions.

There are however a number of problems with this approach:

- Typical variables *x* can have very different scaling. When one column in **M** denotes total bunch charge in [C] while the next is beam energy in [eV], inverting that matrix can result in unacceptable truncation errors.
- When the Jacobian matrix **M** can not be evaluated directly, it must be estimated numerically by a finite difference approach $M_{ij} \approx \Delta f_i / \Delta x_j$. This not only reduces the order of convergence, but also imposes a new question: How large should Δx be chosen?
- There are a large number of situations where basic Newton-Raphson sends the solution to outer space in the first iterations or where the method does not converge at all.
- Not all matrices **M** can be inverted. This is obvious when the number of variables and the number of constraints are not equal. But the same problem arises when one variable does not have any effect on **f**, or when one constraint is not affected by changing any *x*.
- Almost all variables are bounded by external constraints. For example, variables can be limited by specifications of existing hardware, budget restrictions or the location of other beam line components.
- The number of function evaluations required to obtain **M** is equal to the number of free parameters. This is very costly in terms of CPU time because every function evaluation neads a complete GPT run.

The following sections present solution(s) used to approach these problems.

1.12.1.1 Scaling and initial stepsizes

To avoid truncation error problems when the matrix *d***f**/*d***x** is inverted, it is made dimensionless by dividing both **f** and **x** by a typical set of scale-factors: **df** and **dx** respectively. The user can specify the appropriate scaling as a constant value, a relative fraction or both. The total scaling is given by the vector length:

$$\begin{aligned} df_i &= \sqrt{dfabs_i^2 + \left(f \cdot dfrel_i\right)^2} \\ dx_i &= \pm\sqrt{dxabs_i^2 + \left(x \cdot dxrel_i\right)^2} \end{aligned} \qquad [1.35]$$

The sign for the steps in **x** is determined by the external boundary conditions, as explained in section 1.12.1.4.

Although it is possible to automatically detect appropriate scaling, this will always result in more function evaluations. Especially because **f** can be a numerically noisy function depending on the number of particles and accuracy settings of the corresponding GPT run, we have decided not to implement dynamic scaling.

To simplify the equations we directly subtract the target value **ft** from **f** such that the new function **F** must be zero at the solution. As a convention, both the scaled dimensionless function components **F** and the dimensionless variables **X** are written in uppercase. Now that the variables are properly scaled, the first step to obtain derivative information **M** can be chosen simply as $\Delta\mathbf{X}$=1. This leads us to the following new set of equations:

$$\begin{aligned} \mathbf{F}(\mathbf{x}) &= \left(\mathbf{f}(\mathbf{x}) - \mathbf{ft}\right)/\mathbf{df} \\ \mathbf{X} &= \mathbf{x}/\mathbf{dx} \\ \mathbf{M} = \Delta\mathbf{F}/\Delta\mathbf{X} &= \mathbf{F}(\mathbf{x}_N + \mathbf{dx}) - \mathbf{F}(\mathbf{x}_N) \\ \mathbf{x}_{N+1} &= \mathbf{x}_N - \left(\mathbf{M}\right)^{-1}\mathbf{F}(\mathbf{x}_0)\,\mathbf{dx} \end{aligned} \qquad [1.36]$$

The scaling **dx** is a crucial parameter because it is directly related to the stepsize in the determination of **M**. It is our experience that almost all problems with GDFsolve are related to an incorrect scaling of the variables. Figure 1–8 shows a typical too small, good and too large *dx*. A too small *dx* is smaller than the simulation noise on the constraints. A too large *dx* sends the constraints into a non-linear regime. When no correct scaling can be found, the tracing accuracy of GPT must be increased or more particles must be added to the simulation.

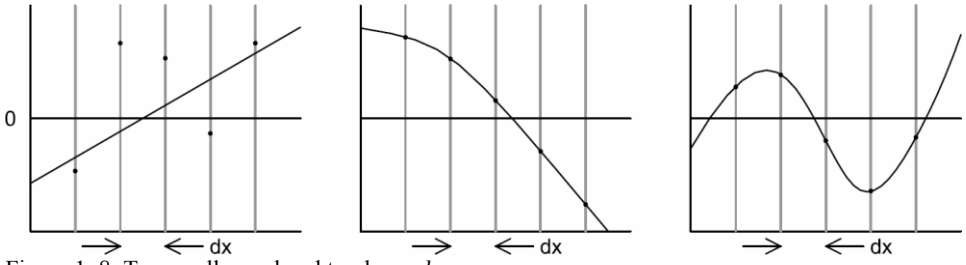


Figure 1–8: Too small, good and too large *dx*.

1.12.1.2 Backtracking

One of the problems with the proposed scheme is that it can send a solution to outer space when the first estimate of **x** is not sufficiently near a root in **F**. A related problem can occur when the sequence simply does not converge. As shown in Figure 1–9 a very small difference in start position for **x** can change a convergent solution into a cyclic or even divergent scheme.

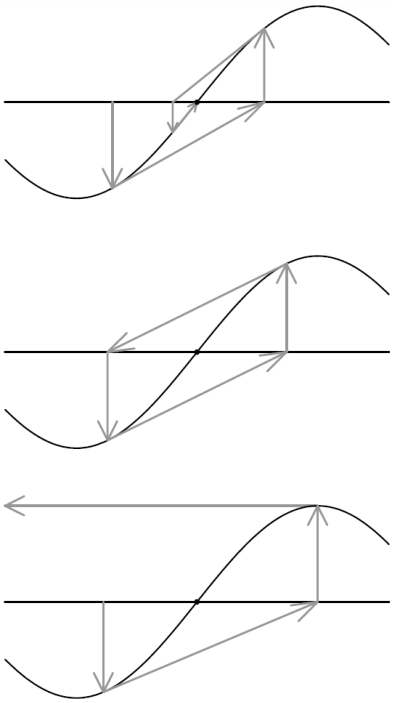


Figure 1–9: Convergent, cyclic and divergent iterations.

To solve these convergence problems, we first have to detect that the algorithm is not working. Because **F** is scaled, its vector length can be used as convergence indicator:

$$\left|F(x_{n+1}) \geq (1-\alpha)\, F(x_n)\right|$$

When [1.37] is not satisfied with sufficiently small α, convergence problems may occur. A typical convergence factor α is a few percent. Not meeting [1.37] usually indicates that the used step is too large. In that case the stepsize is reduced by a factor of 2, for three iterations if necessary, in an attempt to find a smaller **F**. If this fails, GDFsolve is unable to find a solution and terminates.

1.12.1.3 Singular Value Decomposition

Matrix inversion of the Jacobian **M** is numerically a dangerous process. Because the CPU time in all practical applications is dominated by the evaluation of **F**, it is in our opinion better to use Singular Value Decomposition (SVD). A full treatment of the method is beyond our scope here, but basically the method is as follows:

The matrix **M** is written as the product of three matrices0F[1]:

$$\mathbf{M} = \mathbf{U} \cdot \mathrm{diag}(w1, w2, \ldots) \cdot \mathbf{V}^T \qquad [1.38]$$

where the columns of both **U** and **V** are orthonormal. For our application the columns of **V** define an orthonormal set of directions of the variables $\mathbf{X}_i$ and the columns of **U** define the corresponding change in the constraints $\mathbf{F}_i$. The diagonal matrix containing w_i, the singular values, defines the scaling between these two.

Once the Singular Value Decomposition is calculated, inverting **M** is simple:

$$\mathbf{M}^{-1} = \mathbf{V} \cdot \mathrm{diag}(1/w_1, 1/w_2, \ldots) \cdot \mathbf{U}^T \qquad [1.39]$$

[1] In literature, sometimes **V** is defined without the transpose.

Numerically you run into trouble when one or more of the w_i's is much smaller than the others. A relatively small w_j indicates a change in **X** that does not affect **F**. So if you want to change **F** in that direction, you need a very large change in **X**. Typically, such giant steps lead you away from the solution rather than put you on top of it. Therefore the pragmatic approach is simple: Because you can not change **F** in the direction of a small w_i, do not try. Numerically, this means that in the inversion process all $1/w_i$ must be set to zero when w_i is sufficiently small.

When **M** is not square, the matrix **U** is not square either, but all observations still apply. In other words, the number of variables and constraints does not need to be equal. Thus we can use more variables, or even more constraints when desired.

Summarizing the advantages of SVD over matrix inversion:

- When **M** is ill conditioned, one or more w_i's are very small, **F** is very insensitive to one or more directions in **X**. Instead of sending the solution to near infinity, these directions can be ignored by setting the corresponding singular values to zero in the inversion process.
- When **X** has more dimensions than **F**, the matrix **M** is under-conditioned. This results in a nullspace of **M** where a change in **X** does not affect **F**. The new trial value for **X** will not move in the nullspace.
- When **F** has more dimensions than **X** the matrix **M** is over-conditioned. The new trial value for **X** is fitted in a least-squares sense in an attempt to solve for too many constrains.

One question remains: How small can w_i be before that direction must be ignored? This depends on correct scaling between the variables **x** and **X** and between the constraints **f** and **F**. Furthermore, it depends on the accuracy of the underlying GPT simulation. Finally, it depends on the machine precision but this is never a real concern for double-precision calculations. Specific situation-dependent experience is needed, but it appears that the method is not particularly sensitive: A typical range between 10^{-2} and 10^{-6} compared to the maximum w_i can be used as criteria for smallness.

1.12.1.4 External boundary conditions

In many design scenario's, the variables x_j can not be chosen freely. They are restricted to boundary conditions such as the location of other beam-line components. In our implementation all variables are bounded by a minimum and maximum value, forming a hypercube of free space.

When a new trial value $\mathbf{x}_{n+1}$ lies outside the hypercube, an attempt is made to move it inside by changing $\mathbf{x}_{n+1}$ in the nullspace of **M**. This corresponds to a move for any $\boldsymbol{\lambda}$ of the form:

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{V}' \cdot \boldsymbol{\lambda} \qquad [1.40]$$

where only the columns of **V** corresponding to small w_i are used in **V**'.

The additional unknown vector $\boldsymbol{\lambda}$ can be solved from: $\mathbf{xb} = \mathbf{x} + \mathbf{V}'' \cdot \boldsymbol{\lambda}$ where **V**'' contains only the rows of **V**' corresponding to the component of x that must be moved into the boundary hypercube, given by a corresponding xb. A unique solution for $\boldsymbol{\lambda}$ exists when there is an equal number of dimensions in the nullspace of **M** as there are unsatisfied boundary conditions. This process is illustrated in Figure 1–10 where nullspace of **M** is used to solve a boundary condition.

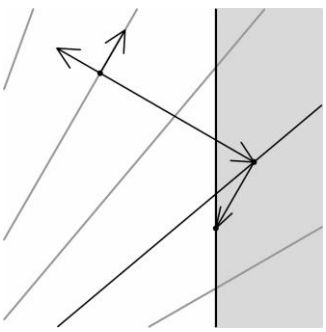


Figure 1–10: A boundary condition is solved by moving the new estimate in the nullspace of **M**.

Solving for one boundary condition might sometimes trigger a new boundary condition. Therefore, the complete boundary-solving process must be iterated until all boundary conditions are satisfied, or until no boundary condition can be solved anymore. Because the dimensions of the nullspace and the number of

boundary conditions are not always equal, SVD is used again to solve for $\boldsymbol{\lambda}$. This does not necessarily satisfy all boundary conditions, and they sometimes need to be enforced with a warning message as a result.

When a parameter is near the maximum of a boundary condition, the step to obtain the derivative information may violate the boundary condition. In that case, a negative step will be used.

1.12.1.5 Broydens method

Obtaining the Jacobian **M** requires as many function evaluations, complete GPT runs, as free parameters. This is expensive in terms of CPU time. However, it is possible to use the function information from the previously successful step to make an estimate of **M** without any additional function evaluations. This requires only one Jacobian to be calculated in the first step to start the process. The update method according to Broyden [2] is given by:

$$\mathbf{M}_{N+1} \approx \mathbf{M}_N + \frac{\left(\delta\mathbf{F}_i - \mathbf{M}_i \cdot \delta\mathbf{X}_i\right) \otimes \delta\mathbf{X}_i}{\delta\mathbf{X}_i \cdot \delta\mathbf{X}_i} \qquad [1.41]$$

where $\delta\mathbf{F}$ is the difference in **F** in a step with size $\delta\mathbf{X}$.

The method might fail to produce a good representation of the actual Jacobian. In that case, the backtracking algorithm will not find a solution and GDFsolve must reinitialize **M** by calculating a full Jacobian. Especially when there is no solution to be found, Broydens Method can increase the number of function evaluations significantly. Furthermore, the method is not advisable when the number of variables is larger than the number of constraints. The algorithm can be switched on and off as required.

1.12.2 Mathematical description of the optimizer

GDFsolve as optimizer tries to find the minimum of any function $g(\mathbf{x})$ by varying all components of **x**. The used algorithm is Powell and described in [2]. Our implementation is very close to [2] with the following modifications: Function evaluations with identical parameters are not repeated, one-dimensional optimization properly generalizes to a single-line minimization and relative termination detection is changed to an absolute value. Qualitatively, the algorithm is as follows:

The first steps find the minimum in the direction of the first component of **x**. Starting from there, the second component of **x** is varied until a minimum is found. This process is repeated as many times as there are dimensions in **x**. To improve the efficiency of the algorithm, the average direction resulting from these iterations is also used as minimization direction, replacing the direction of the largest function decrease. The complete process is iterated until a stable solution is found.

The line-minimization routine takes larger and larger steps downhill until a minimum is bracketed. That is three points where the middle point has the lowest function value. Then the actual minimum is found by either parabolic interpolation or golden section search.

1.12.2.1 Required number of iterations

Any smooth function *g* can be approximated as a second-order Taylor series around $\mathbf{x}_0$ by:

$$f(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \left.\frac{d\,g}{d\,\mathbf{x}}\right|_{\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_o) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_o)\mathbf{H}(\mathbf{x}_o)(\mathbf{x} - \mathbf{x}_o) \qquad [1.42]$$

where the components of the Hessian matrix **H** are defined as:

$$H_{ij}(\mathbf{x}_0) = \left.\frac{d^2 g}{d\,x_i\,x_j}\right|_{\mathbf{x}_0} \qquad [1.43]$$

Solving for an extremum in *g* yields:

$$\mathbf{x} = \mathbf{x}_0 - \mathbf{H}^{-1}(\mathbf{x}_0) \cdot \left.\frac{d\,g}{d\,\mathbf{x}}\right|_{\mathbf{x}_0} \qquad [1.44]$$

Because solving **x** requires knowledge of all components of **H**, at least N^2 functions evaluations are required to find a minimum. Compared to the *N* evaluations required to find a root, optimization typically takes much more iterations than root-finding.

1.12.2.2 Optimizing more functions simultaneously

In real life one typically wants to have it all. For example both the best emittance and the shortest bunch-length. Naturally, this is not possible with any single-valued g. Some tradeoff must be made between the two separate functions. This is accomplished by using individual weight factors m_i for a number of independent functions g_i to optimize. Making use of identical scaling df_i as defined in [1.35], the g to minimize is then given by:

$$g = \sum_i m_i \frac{g_i}{df_i} \qquad [1.45]$$

An additional advantage of this approach is the fact that a negative weight factor automatically maximizes the corresponding component.

1.12.2.3 Additional constraints

Apart from wanting to optimize a function, ideally a number of additional constraints must be met. For example: Make sure that "all charge must come through". To accomplish this, the scaled mismatch between the contraints and the function value is added as penalty to g.

$$g = \sum_i m_i \frac{g_i}{df_i} + \sqrt{\sum_i \left(\frac{f_i - ft_i}{df_i} \right)^2} \qquad [1.46]$$

We realize that this does not result in the most elegant algorithm for a number of reasons: Depending on the scaling the constraints will be met or not. And perhaps even worse, improper scaling of the constraints can move the location of the found optimum off the actual solution. On the other hand, with proper scaling, the method is reliable and does not require any additional function evaluations.

1.12.2.4 Boundary conditions

Just as with the root finder, not all parameters x_i can be chosen freely. Space restrictions or budget reasons can impose maximum possible values. All variables are bounded again by a minimum and maximum value, forming a hypercube of free space.

2 Tutorial

This chapter gives a tutorial introduction to the GPT package. All the files in the tutorial can be found in the GPT distribution in the **`tutorial`** directory. Within that directory, a subdirectory or folder exists for every section in the tutorial containing the described files.

2.1 Quadrupole focusing

In this section we will simulate a rectangular beam with Lorentz factor γ=100 through two quadrupole lenses. The first quadrupole lens focuses the beam in the *y*-direction and defocuses the beam in the *x*-direction. The second quadrupole lens does just the opposite. The net result is a focusing force in both directions and the length and strength of both quadrupoles is chosen so that the beam is sent through a focus in both directions at z=1 meter.

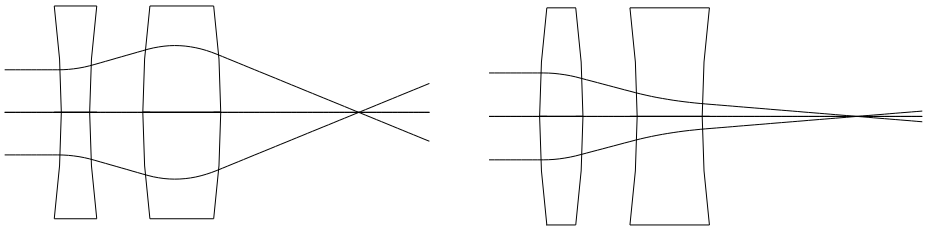


Figure 2–1: Two quadrupole lenses can focus a beam in two directions: xz-plane (left) and yz-plane (right).

2.1.1 The GPT inputfile

The GPT inputfile needed to describe this simulation is listed in Listing 2–1. Like all inputfiles listed in the tutorial, it can be found at the location indicated in the corresponding caption.

Listing 2–1: GPT inputfile for beam focusing using a quadrupole doublet. Filename:
tutorial/quads/quads.in

```
1.  # GPT Tutorial: Quadrupole focusing
2.
3.  # Define beam parameters
4.  gamma = 100 ;
5.  radius= 6e-3 ;
6.
7.  # Start initial beam
8.  setparticles("beam",100,me,qe,0.0) ;
9.  setrxydist("beam","u",radius/2, radius) ;
10. setphidist("beam","u",0,2*pi) ;
11. setGdist("beam","u",gamma,0) ;
12.
13. # Position the quadrupole lenses
14. quadrupole( "wcs","z",0.2, 0.1, 3.90 ) ;
15. quadrupole( "wcs","z",0.5, 0.2,-3.25 ) ;
16.
17. # Specify output times
18. snapshot(0,4e-9,0.05e-9) ;
```

This sample GPT inputfile is described below line-by-line. The very first line in the sample GPT inputfile is ignored by the parser because it starts with a hash mark ("**#**") indicating that the rest of the line is a comment. Comment-lines can appear anywhere in the GPT inputfile.

```
1.  # GPT Tutorial: Quadrupole focusing
```

The first actual section in the GPT inputfile defines various parameters for later use.

```
3.  # Define beam parameters
4.  gamma = 100 ;
5.  radius= 6e-3 ;
```

Line 4 defines the variable **gamma** to be equal to 100 and line 5 sets the **radius** to 6 mm. At this point, these variables don't have any special meaning; they are just for user convenience. In lines 11 and 9, they are used to define the Lorentz factor of the beam and set the beam radius. User-defined variables can have any name, but can not start with a digit.

The second section starts the initial beam.

```
7.  # Start initial beam
8.  setparticles("beam",100,me,qe,0.0) ;
9.  setrxydist("beam","u",radius/2, radius) ;
10. setphidist("beam","u",0,2*pi) ;
11. setGdist("beam","u",gamma,0) ;
```

The **setparticles** keyword, see section 4.4.1.1, is used to start 100 electrons in a particle set named **beam**. The predefined constants **me** and **qe** are set to the charge and mass of an electron. For other built-in constants we refer to Table 4-B on page 105. Because we will not include space charge in this simulation, the last parameter of **setparticles**, defining total charge, can be zero.
After the particles are created, they are all positioned at the origin with zero velocity. The **setrxydist** keyword, see section 4.4.2.5, modifies the x-coordinates of the particles according to a specific distribution: Uniform ("**u**") with center **radius/2** and width **radius**. The **setphidist** keyword, see section 4.4.2.6, sets a uniform angular distribution in the xy-plane over all angles. The combined effect of these two keywords is a disk-like beam, uniformly filled with particles in the xy-plane. Please note that the distribution of the x-coordinates as generated by **setrxydist** is far from uniform: It is linearly increasing with x such that after redistribution over all angles in the xy-plane the resulting beam is uniform. To produce an actual uniform distribution in x, the **setxdist** keyword can be used.
The **setGdist** keyword, see section 4.4.3.7, gives all particles a Lorentz factor of **gamma**, previously defined as 100.

The third section positions the two quadrupole lenses.

```
13. # Position the quadrupole lenses
14. quadrupole( "wcs","z",0.2, 0.1, 3.90 ) ;
15. quadrupole( "wcs","z",0.5, 0.2,-3.25 ) ;
```

The first parameter of the **quadrupole** element, see section 4.7.8, is the ECS, the Element Coordinate System. The default coordinate system of a **quadrupole** positions the lens around the origin, oriented in the z-direction. The ECS lists the new 3D position and 3D orientation of the **quadrupole**, relative to its default. The ECS specification **"wcs","z",0.2**, instructs GPT to position the **quadrupole** relative to the Word Coordinate System (WCS), the default coordinate system, but moved 0.2 m downstream. The second **quadrupole** is moved 0.5 m downstream. Only the positions are affected, the orientation remains unchanged. For the simulation of complex experiments it can be useful to define custom coordinate systems and position elements relative to those. We refer to section 1.4 for details about custom coordinate systems and element coordinate transformations.
The following two parameters of the **quadrupole** lens are length and strength. The first quadrupole has a length of 0.1 m and a positive strength. The second lens needs to work opposite to the first lens to focus the beam in two planes. This could have been achieved by rotating the element 90° around its axis using the ECS, but specifying a negative strength has the same result and is much easier.

The fourth section defines when to write output.

```
17. Specify output times
18. snapshot(0,4e-9,0.05e-9)
```

The **snapshot** keyword, see section 4.3.11, is used to specify the times when output needs to be generated. The parameters are **from**, **to** and **step**, all measured in seconds. Output is written every 50 ps, between 0 and 4 ns.

2.1.2 The GPT batch file

Every GPT simulation contains not only a GPT run, but also includes subsequent data analysis. To automate this process, GPT simulations are typically run as a batch file containing various commands.

Listing 2–2: GPT batch file for beam focusing using a quadrupole doublet. Filename:
tutorial/quads/quads.bat

```
1. gpt -o result.gdf quads.in
2. gdftrans -o traj.gdf result.gdf time x y z
3. gdfa -o std.gdf result.gdf time stdx stdy avgz
4. gdf2a -o std.txt std.gdf
```

Listing 2–2 shows a sample batch file that could be used to run the quadrupole focusing example. It is briefly explained below as a short introduction to some of the capabilities of GPT. We refer to section 2.2 for a description how to actually run this example using GPTwin. Chapter 3 lists all possible data analysis

and pre- and post-processing tools available in the GPT package. UNIX users can run GPT and subsequent data-analysis tools using any command processor of their choice. Note that the GPT line needs a **GPTLICENSE=*xxx*** specification for GPT to run properly.

1. **gpt -o result.gdf quads.in [GPTlicense=*xxx*]**

This line commands the actual GPT simulation. The set-up, simulation parameters and output method as described in the inputfile **quads.in** are now executed. The result is written to the file **result.gdf** as specified with the **-o** option. The outputfile contains all particle coordinates at the requested times or positions and can directly be plotted using GPTwin, see Figure 2–2. Using GPTwin, one can cycle through the plots to and inspect the results. As explained in the complete GPT syntax in section 1.2, an additional **-v** option is useful to track the progress of lengthy simulations.

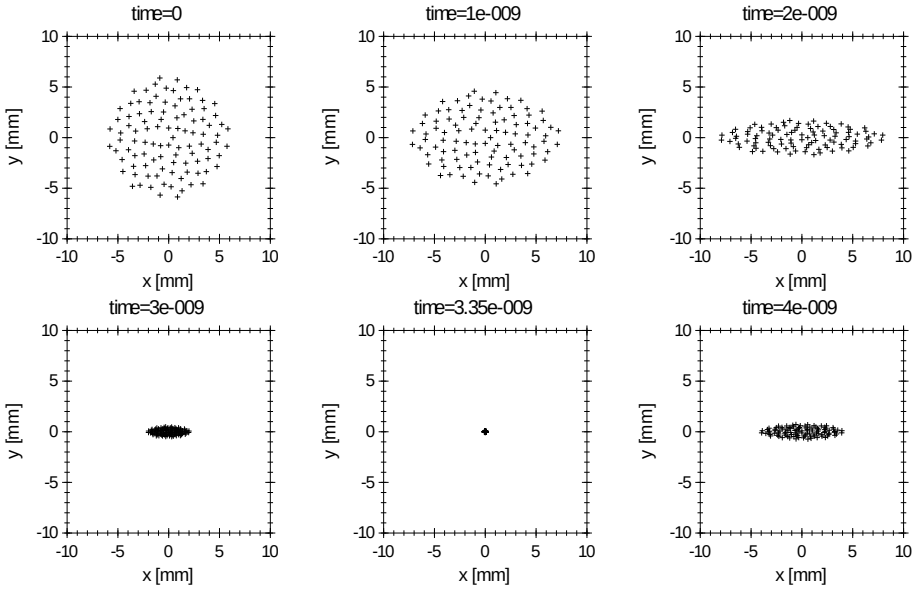


Figure 2–2: Sample raw GPT output: Particle coordinates as function of simulation time.

2. **gdftrans -o traj.gdf result.gdf time x y z**

The GDTRANS program converts the raw GPT output from **result.gdf** to trajectory output written in the file **traj.gdf**. Again, the **-o** option is used to specify the name of the outputfile. This file contains a list of particle trajectories, as shown in Figure 2–3. The trajectories are plotted as function of **time**, and to reduce the size of the file, only the **x**, **y** and **z** coordinates of the trajectories are stored. When a parameter is scanned using the MR program, the trajectories can also be plotted as function of the scanned parameter. See section 3.12.for more information about the use of GDFTRANS.

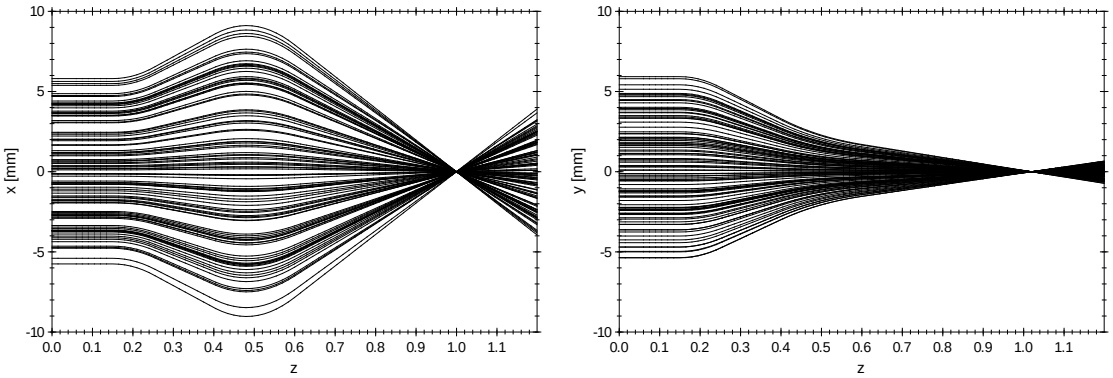


Figure 2–3: GDFTRANS trajectory output.

3. **gdfa -o std.gdf result.gdf time stdx stdy avgz**

The raw GPT output and the trajectory output are typically studied to verify that the inputfile describing the simulation is correct. After this step, one may be interested in overall quantities like bunch-length, beam size and emittance. The GDFA program reads the GPT output from the file **result.gdf** and writes a new file named **std.gdf** containing various macroscopic quantities. In this case, the file **std.gdf** contains only the standard x- and y-deviations and the average longitudinal position all as a function of time, see

Figure 2–4. As described in reference section 3.10, a very large number of data-analysis routines can be specified. It is even possible to create custom GDFA programs, as explained in the GPT Custom Elements documentation.

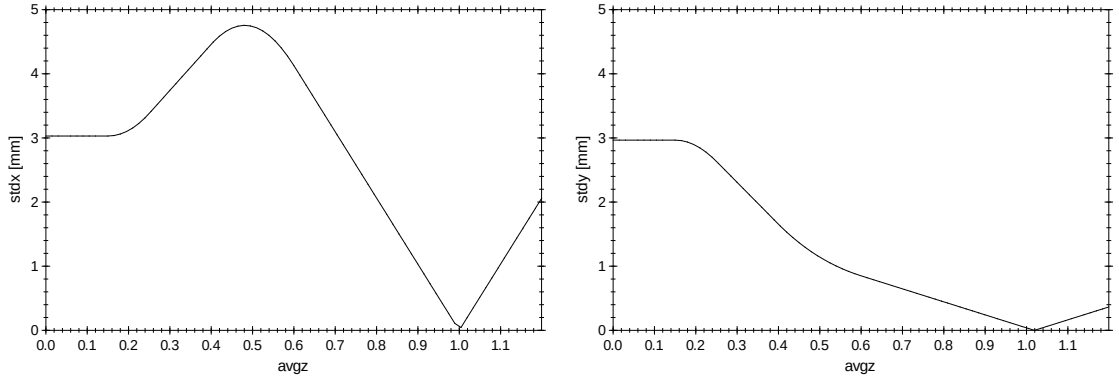


Figure 2–4: GDFA output: Standard deviation of x- and y-size as function of average longitudinal position.

4. **gdf2a -o std.txt std.gdf**

The last line in our batch file converts the file **std.gdf** to the ASCII file **std.txt** using the GDF2A program, see section 3.5. Because GDF files are written in a special binary format, only GPTwin and the GDF utility programs can open them. To be able to export GPT results to other programs, GDF files can be converted to ASCII using the GDF2A program. Part of the result is shown in Figure 2–5.

```
1.        stdx         stdy         avgz         time
2.   3.030e-003   2.966e-003   0.000e+000   0.000e+000
3.   3.030e-003   2.966e-003   1.499e-002   5.000e-011
4.   3.030e-003   2.966e-003   2.998e-002   1.000e-010
5.   3.030e-003   2.966e-003   4.497e-002   1.500e-010
6.   3.030e-003   2.966e-003   5.996e-002   2.000e-010
7.   3.030e-003   2.966e-003   7.494e-002   2.500e-010
8.   3.030e-003   2.966e-003   8.993e-002   3.000e-010
9.   3.030e-003   2.966e-003   1.049e-001   3.500e-010
10.  3.030e-003   2.966e-003   1.199e-001   4.000e-010
11.  3.030e-003   2.966e-003   1.349e-001   4.500e-010
12.  3.030e-003   2.966e-003   1.499e-001   5.000e-010
```

Figure 2–5: Selected GDF2A output.

2.2 The GPTwin user interface

GPTwin is the Microsoft Windows XP/Vista user interface for the GPT package: An easy-to-use environment that covers all aspects of a GPT simulation:

- It provides an editor for the development of inputfiles and custom elements with on-line help.
- It executes GPT and data-analysis tools, while logging diagnostic messages.
- It plots and prints the simulation results in a variety of formats ranging from scatter to color-density-plots.
- It fully automates the development of custom elements.

After installation, the GPTwin executable can be started using:
`Start/Programs/General Particle Tracer/General Particle Tracer`

GPT inpufiles can be opened by dragging and dropping the file into GPTwin or by selecting the Open command from the File menu. Please use any of these methods to open the files **`quads.in`** and **`quads.bat`** in the **`tutorial`** folder. The **`tutorial`** folder is typically located in a subfolder of **`c:\program files\general particle tracer`**, but this may have been modified during installation.

After performing the above steps, the GPTwin user interface should look similar to Figure 2–6. The toolbar (with the buttons below the menu), element bar (shown on the left) and the status bar (below) can all be switched on or off using the View menu. You can use a pointing device or the Window menu to arrange the documents within GPTwin.

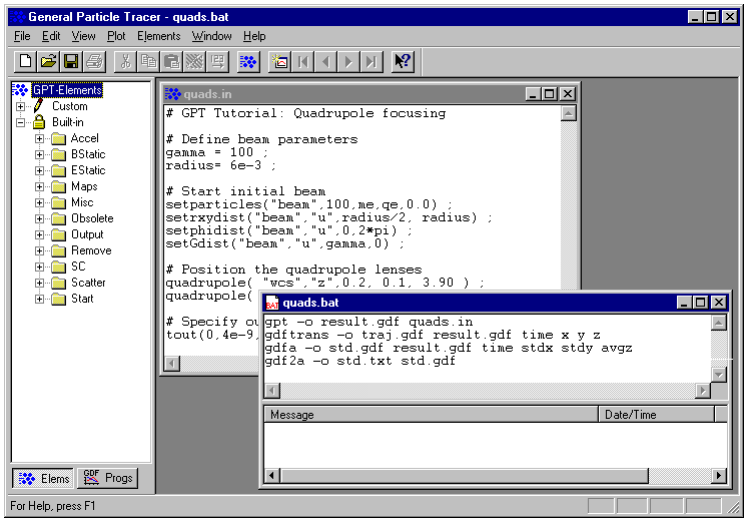


Figure 2–6: The GPTwin user interface with the files **`quads.in`** and **`quads.bat`**.

2.2.1 On-line help

On-line help can always be obtained by pressing the F1 key or selecting the Help menu. For example, when the cursor is located on a line containing the keyword **`quadrupole`** while F1 is pressed, a help screen as shown in Figure 2–7 is displayed. The buttons on the Help toolbar allow you to switch quickly between various help topics.

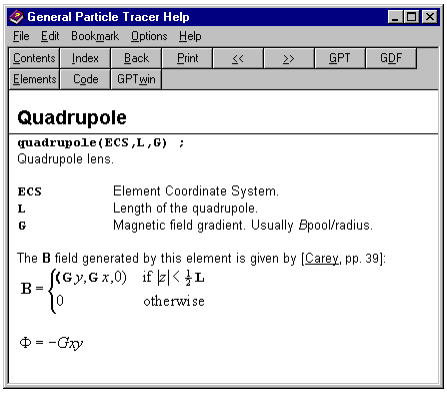


Figure 2–7: On-line help for a quadrupole lens.

2.2.2 Running the quadrupole example using GPTwin

The **quads.bat** batch file containing all commands to run the GPT simulation can be executed by selecting the file and clicking the button on the toolbar. Alternatively, File/Run can be specified on the menu or the CTRL+R key-combination can be pressed. After the run command is specified, simulation messages are shown in the bottom part of the **quads.bat** window, as shown in Figure 2–8. The simulation is completed when the Finished message appears.
More or less space can be reserved for the simulation messages by moving the splitter. When no messages are shown at all, you can still move the splitter, which is in this case located at the bottom of the window. Every diagnostic message is followed by its date and time for your convenience.

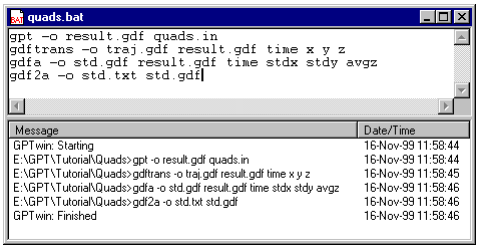


Figure 2–8: Run window with messages.

In this case, because there are no errors, only the commands themselves are shown. Otherwise, an error message will appear with the following format:
filename.in(linenumber): Errormessage
Double-clicking such an error opens the correct file and positions the cursor at the beginning of the line containing the error. Correct any errors and try again. Right-clicking in the messages section shows a pop-up menu with additional commands. It is for example possible to save all messages to an ASCII file for further analysis.

The button can be clicked to stop any running simulation. Using this feature may result in a corrupted and unusable outputfile.

2.2.3 Scatterplots

The raw output of GPT consists of all particle coordinates at the requested output times or positions. To inspect the simulation results using GPTwin, you can do the following:

- Open the outputfile **quads.gdf** either by double-clicking on it, by dragging it into GPTwin or by using the Open command from the File menu and selecting **.gdf** files types.
- You will now get a dialog-box asking you what you would like to see. To display xy-scatterplots at the various output times, select to plot variable **x** along the X-axis and **y** along the Y-axis. Press OK.
- You can now cycle through the plots using the cursor keys (← and →) or using the previous and next plot buttons (and) on the toolbar. The output times are listed in the subtitle of all plots.

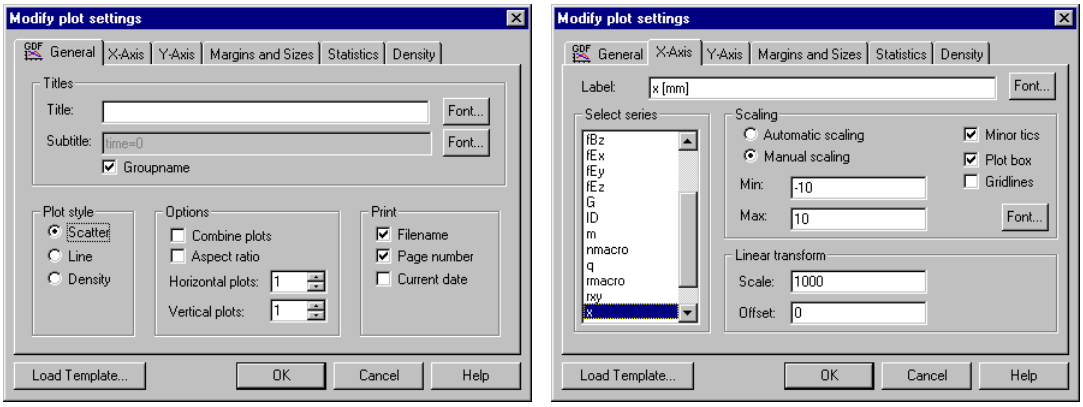


Figure 2–9: Plot settings for **`quads.gdf`**

Because all plots are autoscaled individually, you can not see the beam going through a waist, except by studying the range of the axes. Furthermore, the units would be more convenient when given in [mm]. To solve these shortcomings, set the scaling to manual and chance the units, as shown in Figure 2–9.

- To produce this settings dialog-box, double-click anywhere in the plot window or select the Settings command from the Plot menu.
- Go to the sheet specifying the X-axis. The values plotted along the axis are **`scale`****x*+**`offset`**. Set the scale to 1000 to convert from [m] to [mm]. When the axis scaling is changed, it is recommended to also change the axis label. Repeat for the Y-axis.
- Select Manual scaling. Now you can enter the values for the minimum and the maximum. We suggest using -10 and 10. Repeat for the Y-axis.

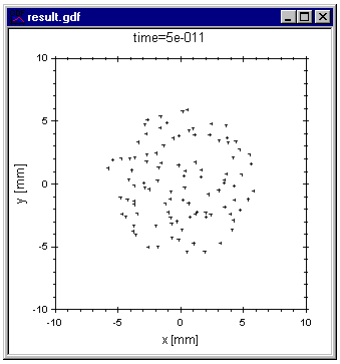


Figure 2–10: **`Result.gdf`** plotted in GPTwin. “Aspect ratio” is selected to plot a cylindrical-symmetric beam as a circle instead of an ellipse.

Now all plots have the same scale, as shown in Figure 2–10. When you cycle through the plots, you will clearly see the beamsize changing and going through a waist.

If you decide to rerun GPT with different inputfile parameters, you can update the selected plots either by pressing the Update button on the Toolbar, by pressing F5 or by selecting the Update command from the Plot menu. The plots shown are then based on the new data.

2.2.4 Statistics

For detailed data analysis, the GDFA program can be used as explained in section 3.10. However, sometimes it is more convenient to calculate simple statistics directly from the raw GPT data. Figure 2–11 shows the statistics dialog, obtained by double-clicking on the plot, calculating the average z-position of all particles. The result can be plotted in the window and is automatically updated.

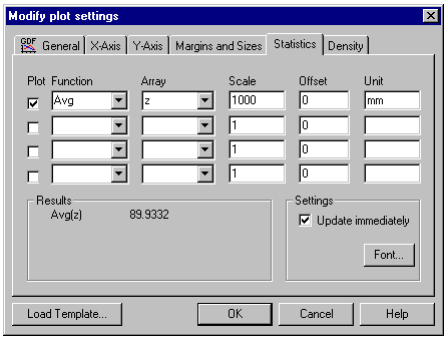


Figure 2–11: Statistics dialog calculating and plotting the average z-position of all particles.

The Plot check box must be checked to plot the selected statistic in the window after the dialog is closed. The scale and offset parameters are identical to those on the x- and y-axis dialogs. Selectable statistical functions are listed in Table 2-A. Statistics are always calculated over the complete data set. Therefore, cropping a plotted axis by selecting manual scaling does not affect the results.

Table 2-A: GPTwin statistics functions

Function	Description
`Min`	Minimum
`Max`	Maximum
`Max-Min`	Maximum-Minimum
`Avg`	Average
`Std`	Standard deviation
`Sum`	Sum
`N`	Number of particles

2.2.5 Trajectory output

Trajectory output is calculated in the batch file **`quads.bat`** from the raw GPT output by the GDFTRANS program:

```
gdftrans -o traj.gdf result.gdf time x y z
```

To display the results open the file **`traj.gdf`**, select z horizontally, x vertically and check the lineplot option on the General property page. Now the individual particle trajectories are shown one by one when cycling through the plots. Alternatively, you can check the combine plot option, also on the General page, to plot all trajectories in the same window.

Instead of plotting the x-coordinate along the y-axis, it is also possible to select the y-coordinate along the y-axis. Use the New Window command in the Window menu to open a new window and plot a different projection. The result is shown in Figure 2–12. When the combine plot option is selected in neither of the plots, cycling through the individual trajectories in one plot will automatically show the corresponding trajectory in the other projection. When the GPT inputfile is modified and rerun, updating the plots with the Update button on the Toolbar will automatically update both windows.

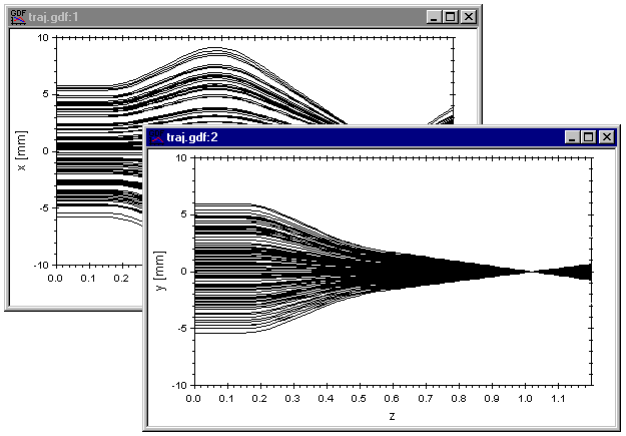


Figure 2–12: Two different projections of all particle trajectories.

2.2.6 Extracting beam parameters

Most data analysis is done by the GDFA program, capable of calculating a large number of different beam parameters from the raw GPT output. In this section, we will create plots of the transverse standard deviations as function of longitudinal position

```
gdfa -o std.gdf quads.gdf time stdx stdy avgz
```

The parameters **stdx**, **stdy** and **avgz** are standard programs of GDFA calculating the following beam properties:

stdx	Standard *x*-deviation of the particles: A measure for the beam size in the *x* direction.
stdy	Standard *y*-deviation of the particles: A measure for the beam size in the *y* direction.
avgz	Average *z*-coordinate of the particles: Beam position.

The variable **time** instructs GDFA to produce output so that the output contains the requested beam parameters at different times. The output is written to the file **std.gdf**, specified with the **-o** option. For a complete description of GDFA and the built-in programs, we refer to section 3.10.

To view **std.gdf** using GPTwin, you can do the following:

- Open **std.gdf**.
- Select Line plot, **avgz** on the horizontal axis and **stdx** on the vertical axis. Press OK.
- To also see a plot of **stdy**, select the New window command from the Window menu.
- Now select Line plot, **avgz** on the horizontal axis and **stdy** on the vertical axis. Press OK.

You will now see two windows showing different results from the same file, as shown in Figure 2–13.

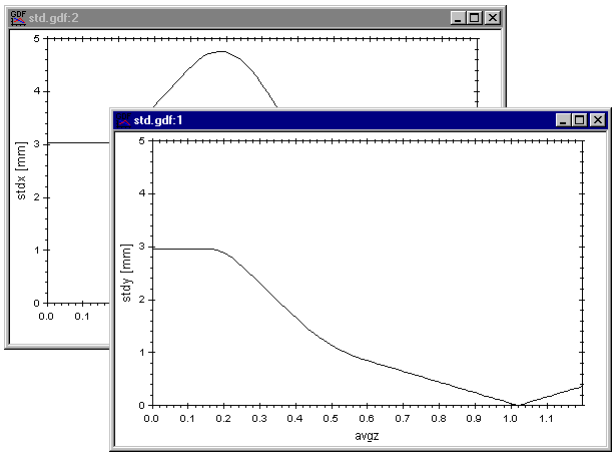


Figure 2–13: Beam size as function of z. The waist in the *x*-direction is much more position dependent than the waist in the *y*-direction.

2.2.7 Templates

Having to set all plot options can be time consuming. As a convenience it is possible to save and load all plot settings using a template file. Template files always have extension **.gdt**. For example, when the file **traj.gdf** is first opened, loading the template **traj_y_z.gdt** immediately sets all options for a trajectory plot of *y* as function of *z*. In all tutorial sections, template files are present having a name in this format **filename_yaxis_xaxis.gdt**. When **filename.gdf** is opened with such a template, an **yaxis** as function of **xaxis** plot is immediately shown. The most convenient way to open a template is clicking the Load Template button in the lower-left corner of the plot-settings dialog box. Alternatively, the Load Template command in the Plot Menu can be used.

The plot settings of an existing plot can be saved with the Save template command in the Plot menu. A dialog box appears asking what settings to store. To (re)load plot settings from a template file, the (Re)Load template command in the Plot menu can be used. A dialog box appears asking what plot settings to overwrite. Using this mechanism, it is possible to create templates for the x-axis only, or restore only the x-axis from a complete template.

2.2.8 Printing and exporting graphics

Everything shown in a GPTwin window can be printed with the Print command in the File menu. The appropriate printer and paper size can be selected using Print setup in the File menu.

When many plots are present, we recommend increasing the number of horizontal and vertical plots in the General dialog of the plot settings. To see if the results are as expected you can use the print preview command in the File menu. This allows you to cycle through the various pages and zoom in and out.

Various important plot settings for printing graphics can be found in the Margins and Sizes property page of the plot settings, shown in Figure 2–14.

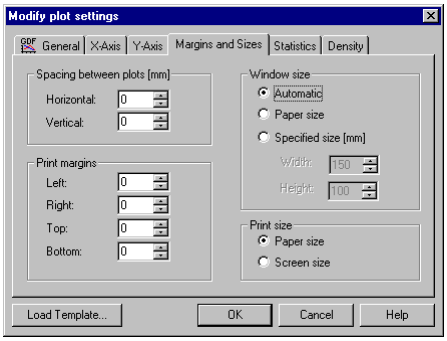


Figure 2–14: Various plot settings

Both text and graphics can be copied to the clipboard using the Copy command in the Edit menu.

It is also possible to save a plot in a Windows Metafile or Enhanced Metafile format using Save As in the File menu. Such metafiles can be included by a variety of other programs like AutoCAD and TeX.

2.3 Scanning the beam energy

This section continues with the example of the first section. Here the effect of the initial beam energy on the spot size at z=1 m is examined. The inputfile needed to examine this effect is shown in Listing 2–3 and can be found in the **tutorial/quadscan** folder. The corresponding batch file containing all the necessary commands, **quadscan.bat**, is explained below.

Listing 2–3: GPT inputfile to examine the effect of beam energy on spotsize:
tutorial/quadscan/quadscan.in

```
1.  # GPT Tutorial: Scanning beam energy
2.
3.  # Start initial beam
4.  setparticles("beam",100,me,qe,0.0) ;
5.  setrxydist("beam","u",radius/2, radius) ;
6.  setphidist("beam","u",0,2*pi) ;
7.  setGdist("beam","u",gamma,0) ;
8.
9.  # Position the quadrupole lenses
10. quadrupole( "wcs","z",0.2, 0.1, 3.90 ) ;
11. quadrupole( "wcs","z",0.5, 0.2,-3.25 ) ;
12.
13. # Specify output screen
14. screen( "wcs","I", 1.0 ) ;
```

The two differences with the previous inputfile are:

A The beam parameters **gamma=100** and **radius=6e-3** statements are removed. This is needed because they are defined in a separate file, as we will see later in this section.

B The **snapshot** statement is replaced with a **screen** statement. A screen interpolates the particle coordinates at a fixed plane in 3D space. Here the screen is positioned at z=1 m. For the reference of **screen**, see section 4.3.10.

Now that we have the inputfile, the next step is to scan the beam energy. To accomplish this, various values for gamma and radius could be specified on the GPT commandline as follows:

```
gpt -o res100.gdf quadbeam.in gamma=100 radius=6e-3
gpt -o res110.gdf quadbeam.in gamma=110 radius=6e-3
etc.
```

Although this works, having to analyze all the outputfiles is tedious. Therefore it is recommended to use Multiple Run, MR, to perform the same job. It runs GPT with one or more varying or fixed parameters, just like the commandline approach, but it collects all output in one structured GDF file. For a complete description of MR we refer to section 3.13.

MR reads which parameter to set or scan from an additional file. For example to scan **gamma** over the range from 50 to 150 in steps of 5, and set the beam radius fixed at 6 mm we typically create a file named **scan.mr** containing the following lines:

```
1.  gamma 50 150 5
2.  radius 6e-3
```

The command to run GPT with **gamma** varying from 50 to 150 then is:

```
mr -v -o result.gdf scan.mr gpt quadbeam.in
```

After running the MR command, all the output is contained in the file **result.gdf**. We can use GDFA again to calculate the beam's standard x- and standard y-deviation:

```
gdfa -o std.gdf result.gdf gamma stdx stdy
```

This time the **group_by** variable is **gamma**. This is because we want to make a plot where every point is calculated for a different **gamma**. As a rule of thumb, the **group_by** variable is usually the scanned parameter, **time** or **position**. The complete batch file is listed in Listing 2–4, where ASCII output of the result is generated using the GDF2A program.

Listing 2–4: Batch file for scanning the beam energy. Filename: **tutorial/quadscan/quadscan.bat**

```
1. mr -v -o result.gdf scan.mr gpt quadscan.in
2. gdfa -o std.gdf result.gdf gamma stdx stdy
3. gdf2a -o std.txt std.gdf
```

The result is shown in Figure 2–15. To reproduce these plots, open **std.gdf** in GPTwin with template **std_xgamma.gdt**, select New Window and open again with template **std_ygamma.gdt**.

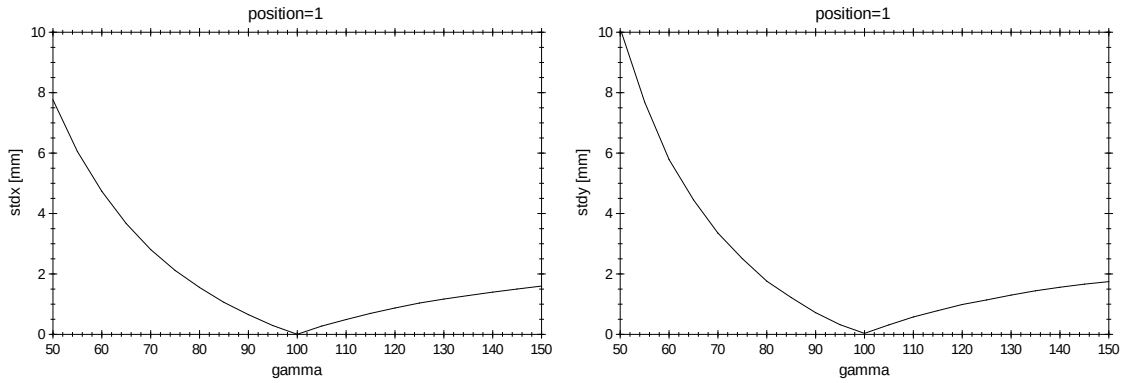


Figure 2–15: Spot size as function of beam energy.

The result shows that the effect of different beam energy on spot size is approximately equal in both planes.

In order to use the same GPT inputfile with and without MR, default values for variables can be defined in the GPT inputfile as shown below:

if(!defined(gamma)) gamma=100 ; If not defined gamma, then set gamma to 100.

2.4 Automatic solving

The GDFsolve program tries to satisfy multi-dimensional constraints and optimizes beam parameters by varying any number of GPT inputfile variables. This section demonstrates the use of GDFsolve by automatically finding the quadrupole settings of the previous sections. A description of the internal algorithms of GDFsolve can be found in section 1.12 and more practical information can be found in reference section 3.11.

In this section an electron beam is sent through two quadrupole lenses and needs to be focused at z=1 m. Unlike the previous sections, the beam starts with a finite emittance to be able to make use of the Courant-Snyder GDFA routines. Furthermore, both time and position output is requested to simplify reproduction of the results shown below. The complete inputfile is shown in Listing 2–5 where the beam energy **gamma** and the strengths of the two quadrupoles **Q1** and **Q2** still need to be specified. All files can be found in the **tutorial/gdfsolve** folder.

Listing 2–5: GPT inputfile describing a beam through two quadrupole lenses with strengths **Q1** and **Q2**. Both time and position output are requested. Filename: **tutorial/gdfsolve/quads.in**

```
# GPT Tutorial: GDFsolve

# Real space of initial beam
radius = 6e-3 ;
setparticles("beam",100,me,qe,0.0) ;
setrxydist("beam","u",radius/2, radius) ;
setphidist("beam","u",0,2*pi) ;

# Phase space of initial beam
GBr = 10e-3 ;
setGBrxydist("beam","u",GBr/2,GBr) ;
setGBphidist("beam","u",0,2*pi) ;
setGdist("beam","u",gamma,0) ;

# Position the quadrupole lenses
quadrupole( "wcs","z",0.2, 0.1, Q1 ) ;
quadrupole( "wcs","z",0.5, 0.2, Q2 ) ;

# Specify output screen
snapshot(0,4e-9,0.05e-9) ;
screen( "wcs","I", 1.0 ) ;
```

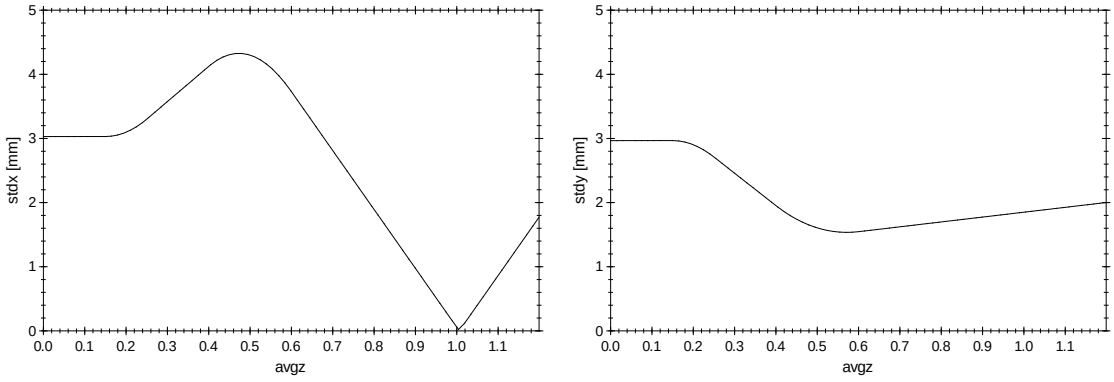


Figure 2–16: Standard deviation of x- and y-size as function of longitudinal position for **gamma**=100. The initial guess for the quadrupole settings, **Q1**=3 and **Q2**=–3 [T/m], does not produce a minimum in both planes at z=1 m.

Figure 2–16 shows the results for **gamma**=100 with an estimated **Q1**=3 and **Q2**=–3 [T/m]. The plots can be reproduced with the following commands and subsequently plotting the file **result.gdf**.

```
gpt -o quads.gdf quads.in gamma=100 Q1=3 Q2=-3
gdfa -o result.gdf quads.gdf time avgz stdx stdy
```

Clearly the beam is not sent through a minimum in the y-plane at z=1 m. The following sections describe how GDFsolve can be used to automatically adapt **Q1** and **Q2** such that a minimum is obtained in both planes.

2.4.1 Finding a minimum using GDFsolve

With some patience and a structured approach, it is possible to find quadrupole settings that produce a minimum in both planes at z=1 m. GDFsolve can do this automatically. The required inputfile for GDFsolve describing the free variables and beam parameters to minimize is shown in Listing 2–6.

Listing 2–6: Inputfile for GDFsolve requesting a minimum in x and y at z=1 m by varying **Q1** and **Q2**. Filename: **opti.sol**

```
1.  [VARIABLES]
2.  Q1 = +3 ;                  # Initial value
3.  Q1.absdelta = 0.01 ;       # Absolute stepsize
4.  Q1.min = 0 ;               # Q1 must be positive
5.
6.  Q2 = -3 ;                  # Initial value
7.  Q2.absdelta = 0.01 ;       # Absolute stepsize
8.  Q2.max = 0 ;               # Q2 must be negative
9.
10. [OPTIMIZE]
11. stdx = 0 ;                 # Approximate minimum in stdx
12. stdx.position = 1.0 ;      # At position z=1 m
13. stdx.abstol = 0.1e-3 ;     # Absolute tolerance
14.
15. stdy = 0 ;                 # Approximate minimum in stdy
16. stdy.position = 1.0 ;      # At position z=1 m
17. stdy.abstol = 0.1e-3 ;     # Absolute tolerance
```

The variables **Q1** and **Q2** are defined in the **[VARIABLES]** section and start from 3 and –3 respectively. Both **Q1** and **Q2** have an absolute initial stepsize of 0.01 specified using **absdelta**. This stepsize is very important. A too small step immediately leads to a local minimum caused by simulation noise. A too large step prevents the algorithm from getting started at all because all steps seem to deviate further from the minimum. When GDFsolve does not work as expected the initial stepsize is the first thing to check. To guarantee that **Q1** and **Q2** have the correct sign, **min** and **max** statements are used.

The **[OPTIMIZE]** section describes one or more GDFA analysis routines to optimize. Both standard x and y deviation, **stdx** and **stdy**, must be as small as possible at a **position** of 1 m. A change in beam size below 0.1 mm, specified using **abstol**, is not significant. This tolerance is as important as the **absdelta** specification for a variable. A too high tolerance regards every solution as a minimum, while a too small tolerance instructs GDFsolve to find a minimum in the simulation noise.

Instead of running GPT directly, we now instruct GDFsolve to run GPT repeatedly to find the desired minimum. Because the beam energy **gamma** is not specified in the GPT inputfile, it must also be defined on the commandline. The output of GDFsolve corresponds to the best GPT run found and can, just like any other GPT run, be analyzed by GDFA. The correct instructions to run GDFsolve and analyze the results are shown in Listing 2–7. The plotted output of GDFA immediately shows a minimum in both the horizontal and vertical plane, see Figure 2–17. When compared to manually finding a simultaneous minimum in two variables, GDFsolve saves a lot of tedious work in this example.

Listing 2–7: Batch file to run GDFsolve and analyze the obtained minimum with GDFA. Filename: **opti.bat**.

```
1. gdfsolve -v -o quads.gdf opti.sol gpt quads.in gamma=100
2. gdfa -o result.gdf quads.gdf time avgz stdx stdy
```

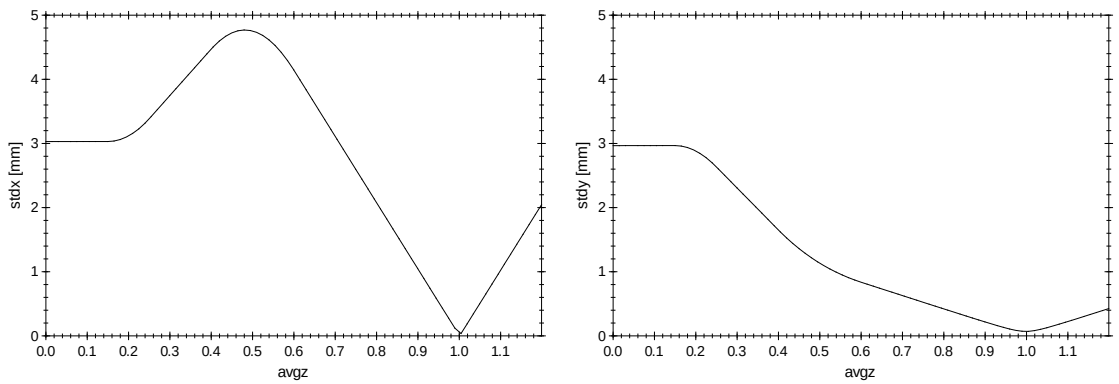


Figure 2–17: Output of GDFsolve used as optimizer, analyzed with GDFA: A minimum in both the horizontal and vertical plane.

Clearly GDFsolve is able to find the requested minimum by varying **Q1** and **Q2**. To know what settings of **Q1** and **Q2** correspond to the final minimum, the outputfile can be inspected using GDF2A. Alternatively, the verbose GDFsolve output can be examined. This output, shown when the **-v** option is specified, shows the progress of GDFsolve after each GPT run as shown in Figure 2–18. The first three columns specify the iteration number **N**, and the current settings for **Q1** and **Q2**. The following two columns specify the resulting **stdx** and **stdy**, where the number between parentheses is the scaled result obtained by dividing by the specified tolerance. The last column is the rms value, the square root of the sum of the squares of the scaled results. This rms column is the value that GDFsolve tries to minimize. As shown in Figure 2–18, GDFsolve works its way in 49 iterations from 3 and –3 for Q1 and Q2 to the final solution of 3.92 and –3.25 [T/m].

N	Q1	Q2	stdx		stdy		rms
1	3	-3	5.78e-005(	0.6)	0.00185(	18.5) (	19.1)
2	3.01	-3	6.36e-005(	0.6)	0.00183(	18.3) (	18.9)
3	3.03	-3	7.3e-005(	0.7)	0.00179(	17.9) (	18.7)
4	4.63	-3	0.00107(	10.7)	0.00174(	17.4) (	28.1)
5	3.64	-3	0.000445(	4.4)	0.000421(	4.2) (	8.7)
6	3.7	-3	0.00048(	4.8)	0.000298(	3.0) (	7.8)
7	4.05	-3	0.000706(	7.1)	0.000494(	4.9) (	12.0)
8	3.78	-3	0.000535(	5.4)	0.000121(	1.2) (	6.6)
9	3.89	-3	0.000596(	6.0)	0.000136(	1.4) (	7.3)
10	3.8	-3	0.000545(	5.4)	9.12e-005(	0.9) (	6.4)
11	3.82	-3	0.000554(	5.5)	7.25e-005(	0.7) (	6.3)
12	3.84	-3	0.00057(	5.7)	7.32e-005(	0.7) (	6.4)
13	3.83	-3	0.00056(	5.6)	6.74e-005(	0.7) (	6.3)
14	3.82	-2.99	0.00058(	5.8)	6.95e-005(	0.7) (	6.5)
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
43	3.92	-3.25	1.74e-005(	0.2)	6.97e-005(	0.7) (	0.9)
44	3.94	-3.25	2.55e-005(	0.3)	8.26e-005(	0.8) (	1.1)
45	3.92	-3.24	3.68e-005(	0.4)	7.01e-005(	0.7) (	1.1)
46	3.92	-3.27	3.33e-005(	0.3)	7.12e-005(	0.7) (	1.0)
47	3.93	-3.25	2.28e-005(	0.2)	7.61e-005(	0.8) (	1.0)
48	3.91	-3.25	1.47e-005(	0.1)	7.86e-005(	0.8) (	0.9)
49	3.92	-3.25	1.74e-005(	0.2)	6.97e-005(	0.7) (	0.9)

Figure 2–18: GDFsolve verbose output showing the progress from 3 and –3 for **Q1** and **Q2** respectively to the final solution of 3.92 and –3.25 [T/m].

2.4.2 Solving for a waist

GDFsolve used as optimizer is a great tool, but 49 iterations for a relatively simple minimization could be a problem. Especially when we realize that when more variables are included the number of iterations often scales with N^2. Your patience will be severely tested when each GPT run contains a non-trivial design including 3D spacecharge calculations. The solution is straightforward: Don't use the optimizer unless you absolutely have to. In the above case, there is a very good alternative: The waist is not only defined by a minimum in **stdx** and **stdy**, but also as a zero in the Courant-Snyder α parameter. Minimizing **stdx** and **stdy** is equivalent to finding a zero in **CSalphax** and **CSalphay**. Such a root-finding procedure is typically much faster and often more reliable than minimizing a function. The required inputfile for GDFsolve is shown in Listing 2–8.

Listing 2–8: Inputfile for GDFsolve finding a simultaneous zero in **Csalphax** and **CSalphay** by varying **Q1** and **Q2**. Filename: **root.sol**

```
1.  [VARIABLES]
2.  Q1 = +4 ;                  # Initial value
3.  Q1.absdelta = 0.01 ;       # Absolute stepsize
4.  Q1.min = 0 ;               # Q1 must be positive
5.
6.  Q2 = -3 ;                  # Initial value
7.  Q2.absdelta = 0.01 ;       # Absolute stepsize
8.  Q2.max = 0 ;               # Q2 must be negative
9.
10. [CONSTRAINTS]
11. CSalphax = 0 ;             # CSalphax must be zero
12. CSalphax.position = 1 ;    # At position z=1 m
13. CSalphax.abstol = 1 ;      # Absolute tolerance
14.
15. CSalphay = 0 ;             # CSalphay must be zero
16. CSalphay.position = 1 ;    # At position z=1 m
17. CSalphay.abstol = 1 ;      # Absolute tolerance
```

The **[VARIABLES]** section is identical to Listing 2–6, specifying to vary **Q1** and **Q2**. The initial step used to calculate the derivative is specified with an **absdelta** of 0.01 and the correct sign is enforced using the **min** and **max** statements. To use GDFsolve as root finder, the **[OPTIMIZE]** section is replaced by a **[CONSTRAINTS]** section: **CSalphax** and **CSalphay** must both be zero at z=1 m. A solution is found when **CSalphax** and **CSalphay** are within **abstol**, in the rms sense, of the target value of zero. For a tolerance of unity, this reduces to a termination criterion of $\sqrt{\mathbf{CSalphax}^2+\mathbf{CSalphay}^2}<1$. As shown in Listing 2–9 and Figure 2–19 running GDFsolve and the produced results are almost identical to the minimization case described in section 2.4.1.

Listing 2–9: Batch file to run GDFsolve and analyze the obtained minimum with GDFA. Filename: **root.bat**.

```
1. gdfsolve -v -o quads.gdf root.sol gpt quads.in gamma=100
2. gdfa -o result.gdf quads.gdf time avgz stdx stdy
```

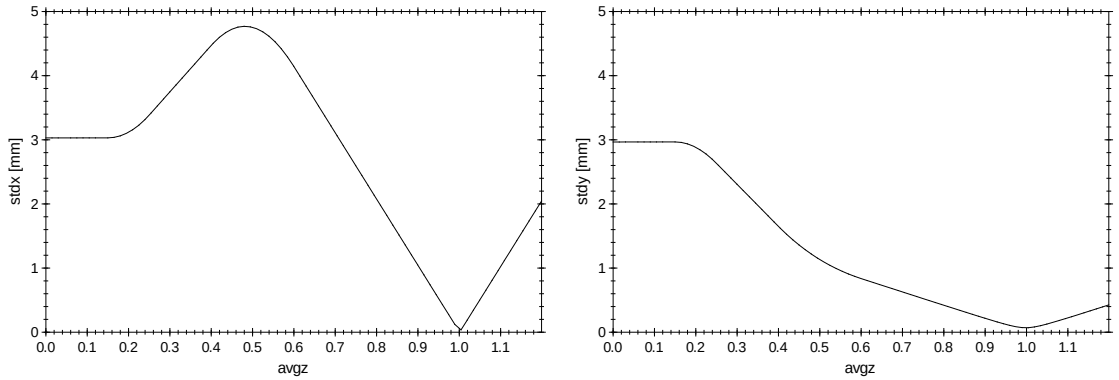


Figure 2–19: Output of GDFsolve used as root-finder, analyzed with GDFA: A minimum in both the horizontal and vertical plane.

The length of the verbose output shown in Figure 2–20 shows the big advantage of the root-finder approach. It takes only 5, compared to 49, attempts to get **Q1** and **Q2** to3.93 and –3.26. This is plus or minus the specified **absdelta**, with about an order of magnitude improvement in CPU time. The reason for this improvent is a completely different algorithm for the root finder. The first function evaluation is identical, but then the Jacobian matrix is estimated by varying each variable an amount of **absdelta**. These lines are indicated by a **J**, and only the difference with the previous run is shown. After the first three runs, a matrix-inversion in GDFsolve yields the first new attempt shown in run number four. A small correction to the Jacobian is then applied and the fifth estimate is already on target.

```
N          Q1          Q2    CSalphax           CSalphay              rms
1           4          -3        40.5(  40.5)       -7.22(  -7.2) (  41.2)
2J       0.01                 0.364(   0.4)      -0.502(  -0.5)
3J                    0.01     1.41(   1.4)      -0.198(  -0.2)
4        3.97       -3.28      -2.22(  -2.2)       -1.32(  -1.3) (   2.6)
5        3.93       -3.26      0.444(   0.4)      -0.442(  -0.4) (   0.6)
```

Figure 2–20: GDFsolve verbose output showing the progress from 4 and –3 for **Q1** and **Q2** respectively to the final solution of 3.93 and –3.26 [T/m].

Is it really that simple to change from minimization to root-finding? Unfortunately, No. Root-finding is fundamentally different from optimization, and typically new (undesired) solutions are introduced. With good initial estimates this provides no problem, but when we start Q1 and Q2 for example from +2 and –2 the results as shown in Figure 2–21 are produced: A collimated beam in the y-plane also produces a zero **CSalphay**, but this was not what we were looking for.

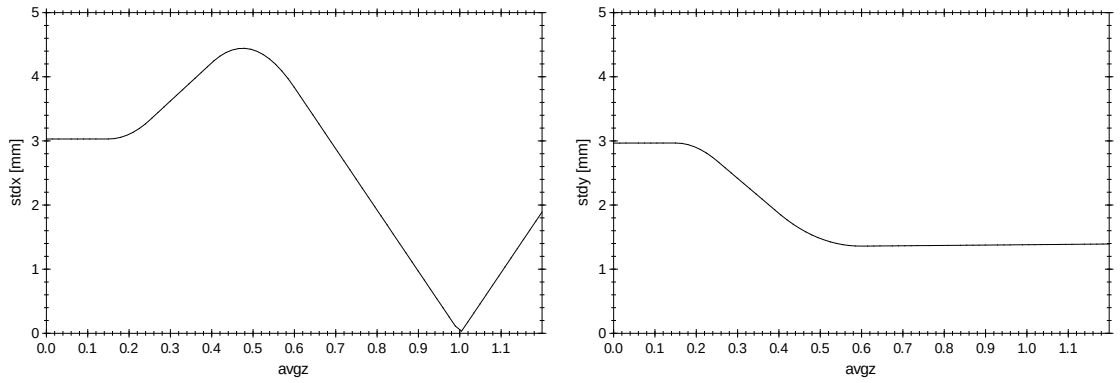


Figure 2–21: Undesired solution for **Q1** and **Q2** starting from an initial estimate of +2 and –2 respectively.

2.4.3 Scanning GDFsolve using MR

Scanning a GDFsolve command using MR allows you to calculate optimal variable values as function of any other variable. It is not the typical thing to do as a novice GPT user. To explain the procedure, the previous example of section 2.4.2 is continued while scanning the beam energy **gamma** using MR.

Listing 2–10: While MR scans the beam energy **gamma**, GDFsolve is used to find the optimal settings for **Q1** and **Q2**.
Filename: **scan.bat**

```
1. mr -v -o quads.gdf gamma.mr gdfsolve root.sol gpt quads.in
2. gdfa -o result.gdf quads.gdf time avgz stdx stdy
3. gdfa -o Q12.gdf quads.gdf gamma Q1 Q2
```

where the file **gamma.mr** contains the following line:

```
1. gamma 90 110 5
```

As shown in Listing 2–10 the first line of the batch file runs MR with **gamma** varying from 90 to 110 as described in the file **gamma.mr**. Instead of directly running GPT with the **quads.in** file, GDFsolve is used to find the corresponding waist described in the **root.sol** file with identical constraints as described in section 2.4.2. The output of all this, **quads.gdf**, contains the best results for every **gamma**. Here we go one step further: Plotting the optimal **Q1** and **Q2** as function of **gamma**. The values for **Q1** and **Q2** are written in the outputfile which can be inspected using GDF2A:

```
gdf2a -o result.txt result.gdf
```

In order to plot the results using GPTwin, two short custom GDFA programs need to be written:

```
int Q1_func( double *result ) { return( gdfmgetval( "Q1", result ) ) ; }
int Q2_func( double *result ) { return( gdfmgetval( "Q2", result ) ) ; }
```

The procedure how to add these functions is explained in the GPT Custom Elements documentation. After these programs have been added, the last GDFA line calculates **Q1** and **Q2** as function of **gamma** in the file **Q12.gdf**. The result is a perfectly linear correspondence, as shown in Figure 2–22.

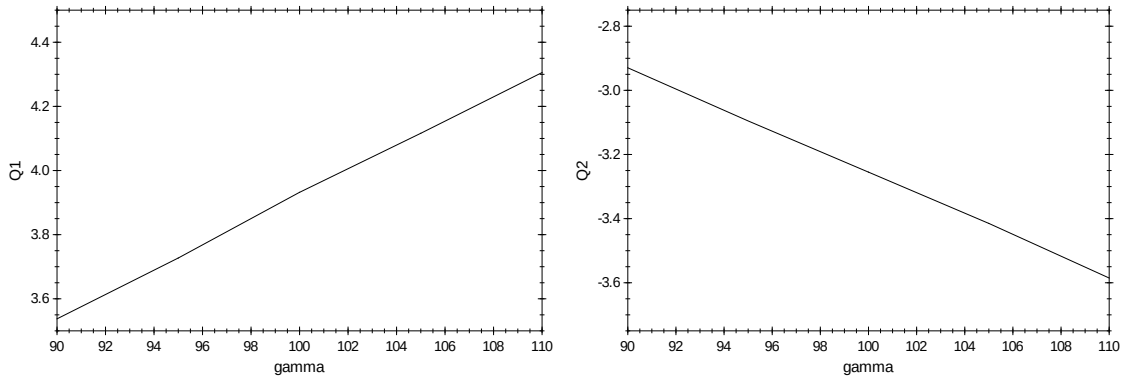


Figure 2–22: Optimum values for **Q1** and **Q2** as function of **gamma**.

2.5 Electrostatic accelerator

This GPT simulation contains only one particle that is accelerated in a homogeneous electric field. This particle starts at the origin and is accelerated over 1 meter by a constant electric field of 1 MV/m. It allows us to test the numerical results with analytical expressions. More realistic accelerator models are typically included using the **map2D_E** element, described in section 4.8.5. Parameters of interest are z, βz and γ. First we will explain the inputfile and then verify the results quantitatively.

The GPT inputfile located in the **tutorial/accel** folder is shown in Listing 2–11. The batch file **accel.bat** can be used to run the example.

Listing 2–11: GPT inputfile with homogeneous acceleration field: **tutorial/accel/accel.in**

```
1.  # GPT Tutorial: Electrostatic accelerator
2.
3.  # Homogeneous E-field: 1MV/m
4.  erect( "wcs", "z",0.5, 1,1,1, -1e6 ) ;
5.
6.  # Position particle at origin with zero velocity
7.  setstartpar("beam",0,0,0, 0,0,0) ;
8.
9.  # Output from 0 to 6 ns every 0.1 ns
10. snapshot( 0, 6e-9, 0.1e-9 ) ;
```

Line 4 positions an **erect**, see section 4.6.4, a uniform homogeneous electric field bounded by a rectangular box. The center of the box is at 0.5 m and all the box dimensions are 1 m. The electric field strength is 1 MV/m. Line 7 starts one particle at the origin with zero velocity using **setstartpar**, see section 4.4.1.12. Line 10 instructs the GPT kernel to generate output every 0.1 ns upto 6 ns.

The instruction to run GPT is:

```
gpt -o result.gdf accel.in
```

The outputfile, **result.gdf**, contains the particle coordinates as function of time, in a format devised for more than one particle to output. We want to plot the *z*-, β– and γ–coordinates of the single particle, but this can not be done immediately. As can be verified using **gdf2a result.gdf**, the outputfile has the following structure:

```
time=0.0 ns
    particle coordinates
time=0.1 ns
    particle coordinates
⋮
time=6.0 ns
    particle coordinates
```

When plotted directly, such a file produces 61 plots containing just one point. What is needed is a GDF file with the following structure:

z	βz	γ	time
...	...	...	0.0 ns
...	...	...	0.1 ns
⋮	⋮	⋮	⋮
...	...	...	6.0 ns

Such output can easily be obtained with the GDFTRANS program, but also with GDFA:

```
gdfa -o avg.gdf result.gdf time avgz avgBz avgG
```

Taking the average of just one particle is the same as returning the particle coordinates. Since the programs to calculate averages are standard in GDFA, these can conveniently be used. **avgBz** calculates the average normalized velocity in the *z*-direction, **avgG** calculates the average Lorentz factor. The **group_by** variable is **time**, because the points in the plot are all calculated at different times. Whether to use GDFTRANS or GDFA is a matter of taste, though we prefer GDFA because it allows us to switch between single- and multiple-particle simulations. The results are graphically shown in Figure 2–23.

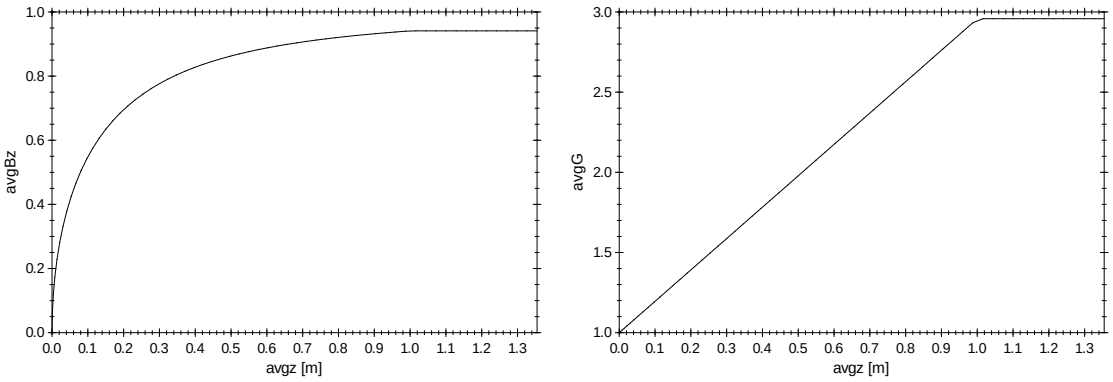


Figure 2–23: Normalized longitudinal velocity (left) and corresponding Lorentz factor (right) for an electron in a uniform acceleration field of 1 MV/m.

As expected the velocity βz=vz/c increases sharply in the beginning but then flattens asymptotically to 1 as the Lorentz factor $\gamma = 1/\sqrt{1-\beta^2}$ starts to differ significantly from unity. The Lorentz factor of the particle increases monotonically, because the acceleration force is constant. At *z*=1 the particle is at the end of the accelerator, beyond which βz and γ remain constant.

To plot the kinetic energy of an electron, the following relation can be used:

$$T[eV] = (\gamma - 1)\, mc^2 / |q|$$

$$T\,[MeV] \approx 0.511\gamma - 0.511$$

In other words, setting **`scale`** to 0.511 and **`offset`** to –0.511 converts the Lorentz factor of an electron to kinetic energy in MeV. The result is shown in Figure 2–24.

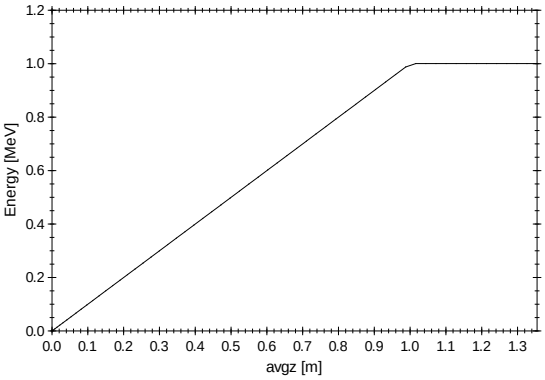


Figure 2–24: Kinetic energy of an electron accelerated in a uniform 1 MV/m electric field.

2.6 Accuracy

The quality of the numerical results as produced by GPT depends mainly on the following two factors:

- The number of particles in the simulation
- The specified **accuracy**

Increasing the number of macro-particles in a simulation generally gives a more accurate result, especially for macroscopic quantities. For large numbers of particles the accuracy generally increases with the square root of the number of particles. The needed CPU time however increases at least linear with the number of particles. Therefore using more particles than necessary is not recommended.

Using **accuracy** the number of significant digits in the individual particle coordinates can be specified to the GPT kernel. We refer to section 1.8 for a detailed description of the Runge-Kutta dynamic stepsize algorithm and section 4.3.2 for the **accuracy** reference.

Consider for example the set-up of the previous section: A single particle accelerated over 1 meter in a homogeneous electric field with E=1 MV/m. Listing 2–12 contains the corresponding GPT inputfile where output is generated by a screen at z=2 m. The calculation accuracy is a variable named **acc**.

Listing 2–12: GPT inputfile for electrostatic accelerator producing output z=2 meter with variable accuracy:
tutorial/accuracy/accuracy.in

```
1.  # GPT Tutorial: Accuracy of the electrostatic accelerator
2.
3.  # Accuracy to be used
4.  accuracy( acc ) ;
5.
6.  # Homogeneous E-field
7.  erect( "wcs", "z",1/2, 1,1,1, -1e6 ) ;
8.
9.  # Position particle at origin with zero velocity
10. setstartpar("beam", 0,0,0, 0,0,0 ) ;
11.
12. # Output at z=2 m
13. screen( "wcs","I", 2 ) ;
```

The analytical expression for γ when z>1 m is:

$\gamma_{Z>1} = 1 + E_{kin}/E_0$

where $E_{kin}=qE$ and $E_0=mc^2$. For 1 MV/m this yields: γ≈2.956935103.

An example of using this inputfile, see **acc5.bat**, with an accuracy of 5 is:

```
1.  gpt -o acc5.gdf accuracy.in acc=5
2.  gdf2a -w 16 acc5.gdf
```

The **-w 16** option specifies a larger column width to print more significant digits. The result is:
γ=2.956703025

Using MR a complete scan can be made:

```
1.  mr -v -o result.gdf accuracy.mr gpt accuracy.in
2.  gdfa -o acc.gdf result.gdf acc avgG tutacc numderivs
```

Where the file **accuracy.mr** contains the following line:
acc 4 10 0.05

This scans the variable **acc** from 4 to 10 in steps of 0.05. The Lorentz factor γ is then grouped by **acc** using GDFA. The number of significant digits and required CPU time as function of **acc** are shown in Figure 2–25. To plot the result with GDFA, we use a custom program to calculate the number of significant digits d:

$d = -\log 10\left((\gamma - \gamma_{ref})/\gamma_{ref}\right)$.

We refer to the GPT Custom Elements documentation for a description of custom GDFA programs.

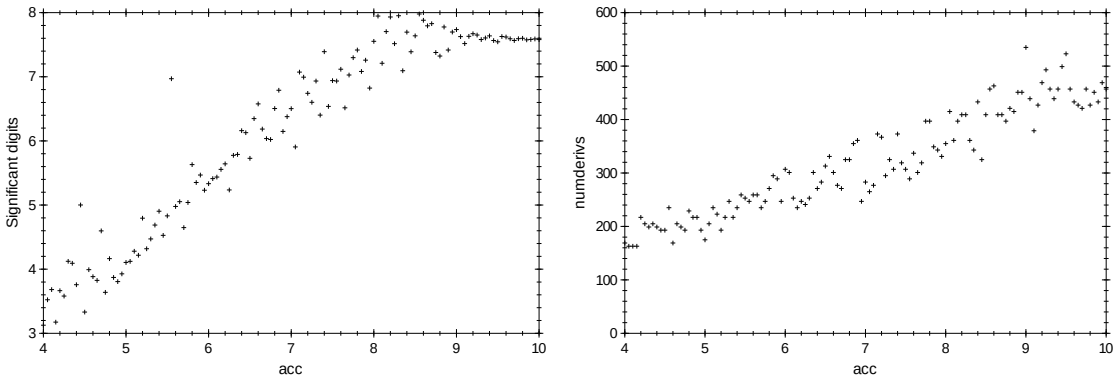


Figure 2–25: Number of significant digits in γ (left) and the number of timesteps in the Runge-Kutta algorithm (right) as function of the specified **`accuracy`**.

For almost all beamline simulations, the default accuracy setting of GPT of 4 is sufficient. However, high-accuracy, single-particle simulations sometimes require a higher accuracy setting, typically in the range between 6 and 8.

2.7 Magnetic mirror

As a demonstration of the high accuracy achievable with GPT a magnetic mirror is simulated. A magnetic field gradient reflects a charged particle gyrating around the magnetic field lines. The following set up is used to demonstrate this effect:
A particle starts at z=–1 m with initial velocity β=0.01 making an angle θ with the z-axis in the xz-plane. The y-coordinate of the particle is chosen to be the Larmor radius, causing the particle to rotate precisely around the z-axis. The field gradient is generated by a solenoid located at z=0 with a radius of 20 cm and 100 loops of 100 A.

The loss cone, the angles θ for which the particle is not reflected by the gradient, is given by [2F3]:

$$\sin^2(\theta) < \frac{\mathbf{B}_0}{\mathbf{B}_m} \qquad [2.1]$$

where $\mathbf{B}_o$ and $\mathbf{B}_m$ are the initial and maximum magnetic field strengths respectively. The expression yields θ=4.98° in our case. The inputfile used to perform the simulations is given in Listing 2–13.

Listing 2–13: GPT inputfile for a magnetic mirror: **tutorial/magmir/magmir.in**.

```
1.  # GPT Tutorial: Magnetic mirror demonstration
2.
3.  # Ultra-high accuracy for best results
4.  accuracy(8) ;
5.
6.  # Define beam parameters
7.  GBeta=0.01 ;                      # Initial norm. momentum
8.  GBetax=GBeta*sin(theta/deg) ;     # Norm. Momentum in x-direction
9.  GBetaz=GBeta*cos(theta/deg) ;     # Norm. Momentum in z-direction
10.
11. Bini=2.36968e-4 ;                # Initial B field (at z=-1)
12. Rl=m*c*GBetax/(q*Bini) ;          # Larmor radius
13.
14. # Start the particle
15. setstartpar( "beam", 0,Rl,-1, GBetax,0,GBetaz ) ;
16.
17. # Position the solenoid
18. solenoid( "wcs", "I", 0.2,10000 ) ;
19.
20. # Time output for trajectories and theta-scan
21. snapshot( 0,0.7e-6,0.5e-9) ;
22. # snapshot( 1e-6 ) ;
```

Mathematical functions like **sin**, **cos**, **tan** and **sqrt** can be used in a GPT inputfile like in normal computer code. The operator precedence is similar to C, with the exception of the exponential operator ("**^**"), not present in C, having the highest precedence. For a complete list of operators and built-in functions, we refer to Table 4-A and Table 4-C on page 104.

We will first examine the following two cases: θ=4.75° and θ=5.25°. In the first case the particle is expected to be lost, in the second case it is expected to be reflected. The results shown in Figure 2–26 were generated by performing the following commands and plotting the trajectory results:

```
gpt -o t475.gdf magmir.in theta=4.75
gdftrans -o traj475.gdf t475.gdf time x y z Bz
gpt -o t525.gdf magmir.in theta=5.25
gdftrans -o traj525.gdf t525.gdf time x y z Bz
```

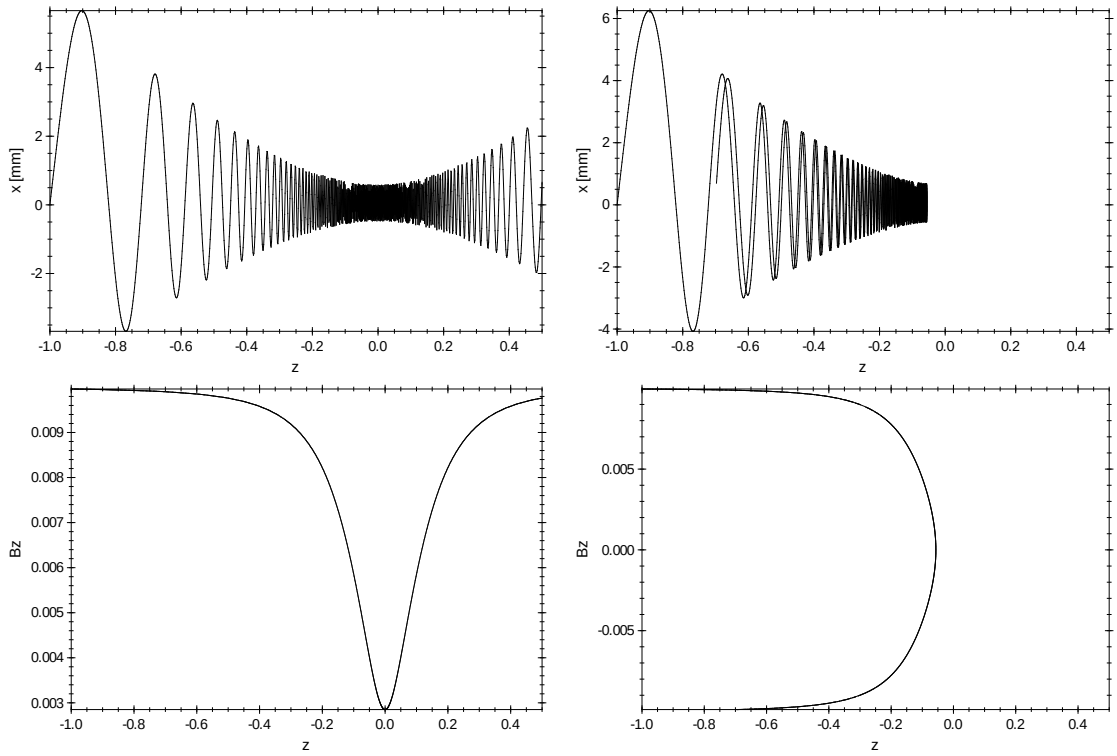


Figure 2–26: Particle trajectory when θ=4.75° (left) and θ=5.25° (right).

It is clear that in both cases the Larmor radius of the particle decreases because of the increasing magnetic field strength. The particle trajectory remains centered around the z-axis. When θ=4.75° the velocity remains positive and the particle is lost. When θ=5.25° the velocity changes sign and the particle is reflected.

Now we will verify [2.1] by scanning θ using MR. The following MR file can be used to scan θ from 0° to 10°, using ten times more steps in the interval 4°<θ<6°. To reduce the size of the outputfile, only one snapshot at 1 μs is needed.

Listing 2–14: **theta.mr**

```
1. theta 0  3 1
2. theta 4  6 0.1
3. theta 6 10 1
```

To plot the z-coordinate as function of theta the following commands have to be specified. Before running, please make sure that only snapshot output at 1 μs is specified.

```
mr -v -o result.gdf theta.mr gpt magmir.in
gdfa -o magmir.gdf result.gdf theta avgx avgy avgz avgBx avgBy avgBz
```

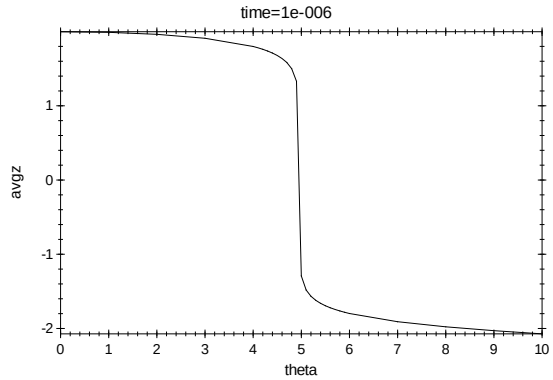


Figure 2–27: Final particle position as function of angle of incidence.

The result, Figure 2–27, indicating for which value of θ the particle is lost, is close to the analytical value: θ=4.98°. A larger final time or screen output and more points around θ=5° can be used to calculate the simulation's loss cone with more accuracy. It yields approximately 4.96° and is left as an exercise.

2.8 Element Coordinate System

All GPT elements, both standard and custom, can be positioned anywhere in 3D space with any orientation. This can be very useful when simulating misaligned or rotated beamline components. GPT uses orientation vectors to position elements and their working will be explained in this section. The use of these vectors is the same for all elements.

As an example we will use two current loops placed vertically above and below the beampath, as shown in Figure 2–28. Together these loops produce a vertical field which bends an electron beam in the horizontal plane. To model the current loops we have chosen to use the **`solenoid`** element because it has its magnetic field defined over all space, not only close the z-axis. For a reference of the element **`solenoid`** see section 4.7.13.

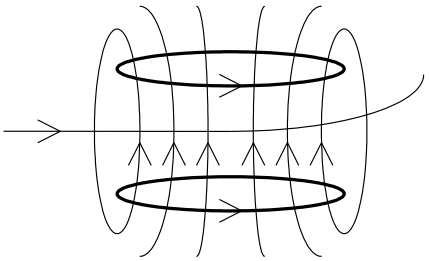


Figure 2–28: Schematic picture of the current loops and the generated field lines.

When using the standard orientation of a solenoid, it is positioned perpendicular to the beam path with the beam going through its center, as shown in Figure 2–29. This is the orientation of a standard focusing coil. As we have seen in previous sections, such a solenoid can be placed using the following syntax:

```
solenoid("wcs","z",1, R, I) ;
```

This places a solenoid at a distance of 1 m from the origin in the z-direction.

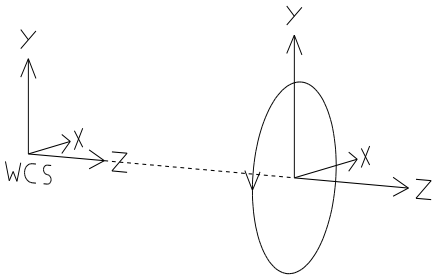


Figure 2–29: Standard orientation of a solenoid.

Because of the 3D capabilities of the GPT kernel, the standard solenoid can still be used to simulate our steering coils. The coils can be positioned off axis and facing a different direction by specifying an element coordinate transform (ECS) in the inputfile. More information about GPT coordinate systems can be found in sections 1.1 and 1.4.3.

First we rotate the solenoid around its origin by specifying its new orientation vectors. The new orientation is shown in Figure 2–30. The solenoid has been rotated 90° around its x-axis.

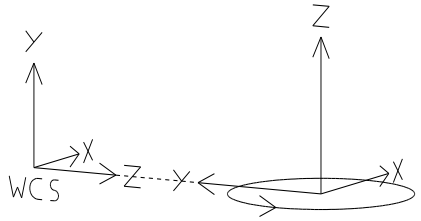


Figure 2–30: Orientation of a solenoid rotated 90° around its x-axis.

The complete syntax line for such a solenoid is:

```
solenoid("wcs", 0,0,1, 1,0,0, 0,0,-1, R, I) ;
```

The first parameter specifies the coordinate system relative to which the subsequent coordinates are given. In our case this is "**wcs**". Then follow three sets of x-,y-,z-coordinates specifying the position and orientation of the element.

The first coordinate set gives the position of the element in x-, y-, z-coordinates. The specified distances are in meters and relative to the coordinate system specified as the first parameter. In our example the center of the solenoid is placed at a distance of 1 m from the origin of "**wcs**" in the z-direction. It is not moved in the x- or y-direction. This translation is specified as **0,0,1**.

The second coordinate set specifies the direction of the new x-direction using directional cosines. The first parameter determines how much of the new x-vector lies in the direction of the x-axis of the reference coordinate system. The reference coordinate system is specified by the first parameter of the syntax line, in our case it is "**wcs**". The second parameter determines how much of the new unit x-vector lies in the direction of the y-axis of the reference coordinate system. And the third parameter determines how much of the new unit x-vector lies in the direction of the z-axis of the reference coordinate system. In our case the direction of the x-axis has not changed, because the solenoid has been rotated round its x-axis. Therefore we specify a unity transformation in the x-direction **1,0,0**.

The third coordinate set specifies the direction of the new unit y-vector the same way as we did for the new x-direction. The components of the new directions are given relative to the standard unit vectors. In our example the new direction of unit vector y is opposite to the "**wcs**" z-direction. This is specified as **0,0,-1**.

The new z-direction is calculated by the GPT kernel and does not have to be specified: $\hat{z} = \hat{x} \times \hat{y}$. The kernel also handles all normalizations.

Finally the reoriented solenoid has to be positioned above the beampath, as shown in Figure 2–31. This means that it has to be moved in the y-direction of the WCS coordinate system. To move the solenoid 5 cm in this direction should be specified as the translation **0,0.05,1**.

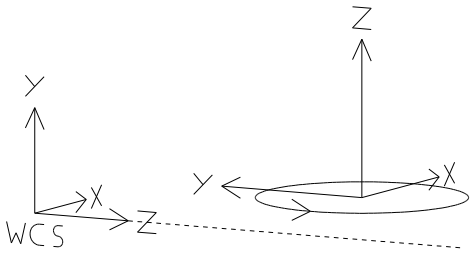


Figure 2–31: Solenoid with a new orientation positioned above the beampath.

The complete syntax line now becomes:

```
solenoid("wcs", 0,0.05,1, 1,0,0, 0,0,-1, R, I) ;
```

A second solenoid is placed 5 cm below the beampath with the same orientation. The corresponding inputfile is shown in Listing 2–15.

Listing 2–15: GPT inputfile demonstrating a non-trivial ECS: **tutorial/ecs/ecs.in**

```
# GPT Tutorial: Bending particles using two current loops

# Define particle parameters
gamma = 10 ;
betaz = sqrt(1-gamma^-2) ;
vz    = c*betaz ;

# Start particles
setstartpar( "beam", -.2e-3,0,0, 0,0,gamma*betaz ) ;
setstartpar( "beam",  .0e-3,0,0, 0,0,gamma*betaz ) ;
setstartpar( "beam", +.2e-3,0,0, 0,0,gamma*betaz ) ;

# Position the solenoids
solenoid( "wcs", 0, 0.05,1, 1,0,0, 0,0,-1, 0.1, 100 ) ;
solenoid( "wcs", 0,-0.05,1, 1,0,0, 0,0,-1, 0.1, 100 ) ;

# Specify output times
snapshot( 0,1.5/vz,0.05/vz ) ;
```

Now the two solenoids are placed above and below the beampath in their new orientation. To view the trajectory of the particle run the batch file: **tutorial/ecs/ecs.bat**. It contains the command to run GPT with inputfile **ecs.in** and outputfile **result.gdf**: Then the trajectories are calculated with the GDFTRANS program and written to **traj.gdf**.

```
gpt -o result.gdf ecs.in
gdftrans -o traj.gdf result.gdf time x y z
```

The particle trajectories can be shown by plotting the z-position horizontally and the x-position vertically, as shown in Figure 2–32.

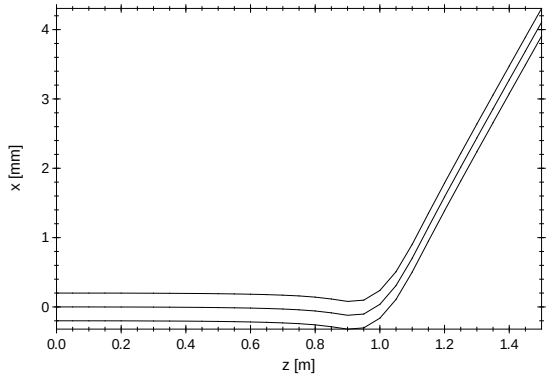


Figure 2–32: Particle trajectories through the steering coils, plotted using GPTwin.

We see in Figure 2–32 that the beam is bent in the positive x-direction due to the solenoid field. Please note that outside the coils there is a small magnetic field with opposite direction, .causing a small deflection in the negative x-direction.

For macroscopic beam investigation, we can also use the GDFA program:

```
gdfa -o avg.gdf result.gdf time avgx avgy avgz
```

2.9 Initial particle distribution

In this example, we will start a 1 MeV electron beam with a total charge of 100 pC. The position, momentum and energy distributions are all Gaussian, but this can easily be modified to uniform, linear or quadratic distributions when required. Such a beam can be specified in a number of well-defined steps:

Step	Plane(s)	Elements used
Specifying the particle set		**setparticles**
Specifying the real-space distribution functions	xyz	**setzdist**, **setxdist**, **setydist**, **setrxydist**, **setphidist**
Specifying the phase-space distribution functions	xβx, yβy	**setGBxdist**, **setGBydist**
Scale emittances	xβx, yβy	**setGBxemittance**, **setGByemittance**
Specifying the energy-distribution function	γ (Energy)	**setGdist**
Specifying divergence angles	xβx, yβy	**addxdiv**, **addydiv**

The files for this tutorial section are located in the folder **gpt/tutorial/cylbeam**. A **snapshot(0)** statement is used to inspect the initial particle distribution at t=0. In other words, a **snapshot(0)** statement writes the particle distribution to the outputfile exactly as it starts.

To start the particles in our sample 1 MeV electron beam, we first define a particle set named "beam", containing 1000 particles having a total charge of 100 pC. The **setparticles** element is described in section 4.4.1.1. The built-in constants **me** and **qe** represent the mass and charge of an electron respectively.

```
1. nps  = 1000 ;       # Start 1000 particles
2. Qtot = -100e-12 ;   # Total charge 100 pC
3. setparticles( "beam", nps, me, qe, Qtot ) ;
```

To give these particles a Gaussian longitudinal distribution centered around z=0 we use the **setzdist** element, see section 4.4.2.4. The typical length is specified as 4 mm and the distribution is extended to 3 sigma in both directions. In this case the distribution specification is: **"g",0,4e-3,3,3**. The resulting longitudinal distribution is shown in Figure 2–33.

```
4.  zlen   = 0.004 ;   # Buch length: 4 [mm]
5.  radius = 0.002 ;   # Bunch radius: 2 [mm]
6.  setzdist( "beam", "g", 0,zlen, 3,3 ) ;
```

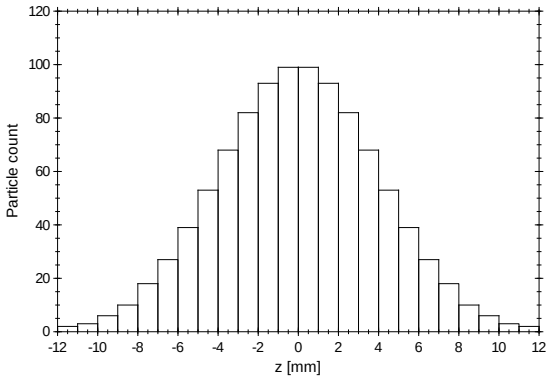


Figure 2–33: Longitudinal particle distribution obtained using GDF2HIS, see section 3.8.

It is quite simple to specify other distributions. For example a uniform distribution centered around z=1 mm could be specified by: **"u",1e-3,4e-3**. Please see section 1.6.1 or Table 1-C on page 17 for a description of all possible distribution specifications.

The elements **setxdist** and **setydist** are used in a similar way as **setzdist**. In this example, the radial distribution of the particles is Gaussian in r and specified using **setrxydist**, see section 4.4.2.5. For radial distributions it is important to never generate negative radii. Therefore, the left tail of the Gaussian distribution is set to 0 sigma. For a uniform distribution, the distribution parameters of **setrxydist** must always be: **"u",rmax/2,rmax**.
The **setrxydist** element only modifies the x-coordinates of the particles, assuming a rotational symmetric distribution in the xy-plane. Therefore, **setphidist**, see section 4.4.2.6, must be used to specify a uniform φ distribution over 2π in the xy-plane. The resulting distribution is shown in Figure 2–34.

```
7.  setrxydist( "beam", "g", 0, radius, 0,3 ) ;
8.  setphidist( "beam", "u", 0, 2*pi ) ;
```

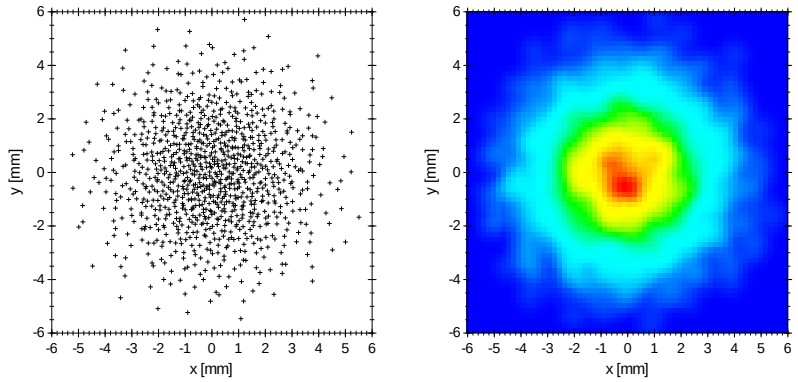


Figure 2–34: Beam distribution in xy-plane scatterplot (left) and density plot (right). The density plot is created with 10 times more particles.

As an alternative to **setrxydist** and **setphidist** the elements **setxdist** and **setydist** can be used to specify the distribution in the xy-plane. Therefore, it is very important to understand the difference between **setrxydist** and **setxdist**. Both elements set the *x*-coordinates of the particles to a specified distribution. However, a uniform distribution for **setxdist** sets the *x*-coordinates of all particles uniform in the x-plane only. A uniform distribution for **setrxydist** results in a uniform distribution in the xy-plane when a uniform **setphidist** is also used. To accomplish this, **setrxydist** adds relatively more particles at larger radii because the amount of 'space' on a circle grows linear with *r*.

The phase-space distribution is set using **setGBxdist** and **setGBydist**, see section 4.4.3.1 and 4.4.3.2 respectively. We use a Gaussian distribution centered around zero with a standard deviation of 1e-3, cut-off at three sigma. Please note that by specifying these distributions, we implicitly specify the transverse emittances. However, we don't need to worry about this too much, because the emittances can be fine-tuned later. Also the divergence angles are specified separately. Important in this step is only the type of distribution, e.g. Gaussian or uniform.

```
9.  setGBxdist( "beam", "g", 0, 1e-3, 3,3 ) ;
10. setGBydist( "beam", "g", 0, 1e-3, 3,3 ) ;
```

The transverse emittances are set using **setGBxemittance** and **setGByemittance**, see sections 4.4.3.8 and 4.4.3.9 respectively. These elements scale the transverse momentum coordinates linearly till the specified emittance is reached. In this example, the emittance is set to 5π mm-mrad. Running the GDFA programs **nemixrms** and **nemiyrms** will result in emittance values of 5π mm-mrad, independent on the average beam energy and divergence angles specified later.

```
11. setGBxemittance( "beam", 5e-6 ) ;
12. setGByemittance( "beam", 5e-6 ) ;
```

Now we can specify the beam energy distribution using **setGdist**. In this example, an average energy of 1 MeV and a uniform energy spread of 1% are used. The Lorentz factor is calculated from the average beam energy. The result is shown in Figure 2–35.

```
13. Eo  = 1e6 ;                 # Average beam energy: 1 [MeV]
14. G   = 1-q*Eo/(m*c*c) ;      # Corresponding gamma
15. dG  = (G-1)/100 ;           # 1% Energy spread
16. setGdist( "beam", "u", G, dG ) ;
```

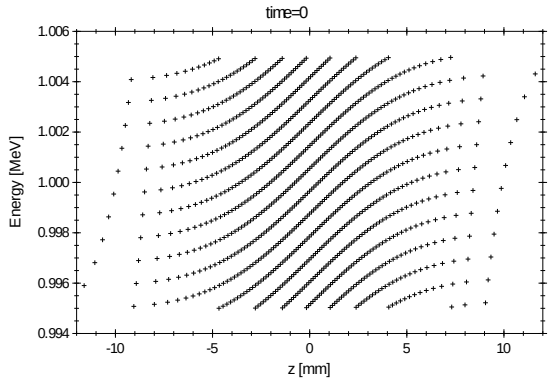


Figure 2–35: Correlation between longitudinal position and beam energy. The interference pattern is an artifact of the Hammersley sequence. The **addzdiv** element rotates this plot.

Optionally, the transverse phase-space divergence angles can be specified using **addxdiv** and **addydiv**, see section 4.4.4.1 and 4.4.4.2 respectively. When written as a separate element, it is not possible to include the beam radius in the divergence calculations because the beam radius is differently defined for a uniform and a Gaussian transverse distribution. Therefore, the divergence must always be specified in [rad/m]. To create a beam with a radius of 2 mm and a divergence angle of 0.1 rad, a divergence of 50 rad/m must be specified. In this example, a divergence of 1 rad/m is used, defocusing in the *x*-plane and focusing in the *y*-plane. The result is shown in Figure 2–36.

```
17. addxdiv( "beam", 0,  1 ) ;
18. addydiv( "beam", 0, -1 ) ;
```

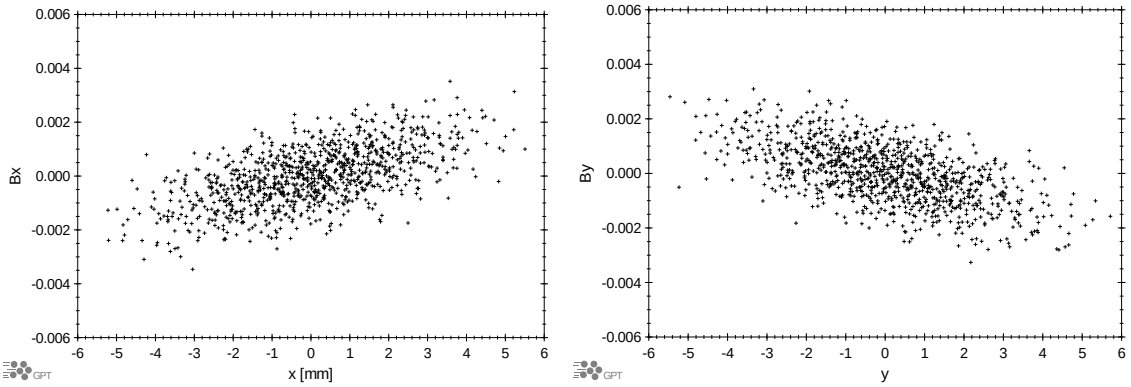


Figure 2–36: Transverse phase-space plots. The vertical axis is the transverse velocity, normalized by the speed of light.

2.10 Photo-cathode, starting particles as function of time

This section demonstrates how to start particles as function of time. All files are located in the **tutorial/cathode** folder. The example is a 100 pC electron bunch, emerging from a flat photo-cathode during 100 fs. A typical set-up consists of a copper or semiconductor cathode surface illuminated by a laser pulse. Either an RF cavity or an electrostatic accelerating field must be present to accelerate the electrons away from the cathode. Space-charge effects are not included in this example, but can easily be added with the **spacecharge2Dcircle** element.

To start the electrons emerging from the cathode, we first define a particle set named "beam", containing 1000 particles having a total charge of 100 pC.

```
# Photocathode
nps  = 1000 ;       # Start 1000 particles
Qtot = -100e-12 ;   # Total charge 100 pC
setparticles( "beam", nps, me, qe, Qtot ) ;
```

These particles are to be started uniformly in a 100 fs time interval, having an initial radius of 0.25 mm.

```
tlen = 100e-15 ;   # Laser pulse length: 100 fs
radius = 0.25e-3 ; # Bunch radius: 0.25 mm
```

The radial distribution of the particles is uniform **"u"** in *r* and is set using **setrxydist**. The third parameter of **setrxydist** is the center of the radial distribution, half the beam radius. The beam is cylindrical symmetric, specified as a uniform φ distribution over 2π using **setphidist**. This will result in a particle distribution as shown in Figure 2–37.

```
setrxydist( "beam", "u", radius/2, radius ) ;
setphidist( "beam", "u", 0, 2*pi ) ;
```

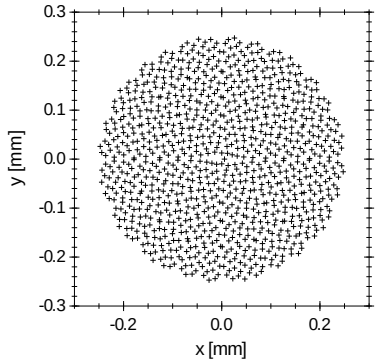


Figure 2–37: Particle distribution in xy-plane.

The following code assumes that all electrons leave the cathode having an energy of precisely 0.4 eV. Initially, **setGBzdist** is used to set the corresponding γβz of all particles. Then the velocity component is homogeneously distributed over half a sphere using **setGBthetadist**. This element sets a spherical particle distribution in γβz-γβr-plane, keeping the angle in the γβx-γβy-plane constant. **setGBphidist** must be used to enforce cylindrical symmetry. The half-sphere distribution of the particles is shown in Figure 2–38. Please note that these start conditions automatically determine the initial emittance.

```
Eo  = 0.4 ;               # Start with 0.4 eV
G   = 1-q*Eo/(m*c*c) ;    # Corresponding Gamma
GB  = sqrt(G^2-1) ;       # Corresponding Gamma*Beta
setGBzdist(     "beam", "u", GB, 0 ) ;
setGBthetadist( "beam", "u", pi/4, pi/2 ) ;
setGBphidist(   "beam", "u", 0, 2*pi ) ;
```

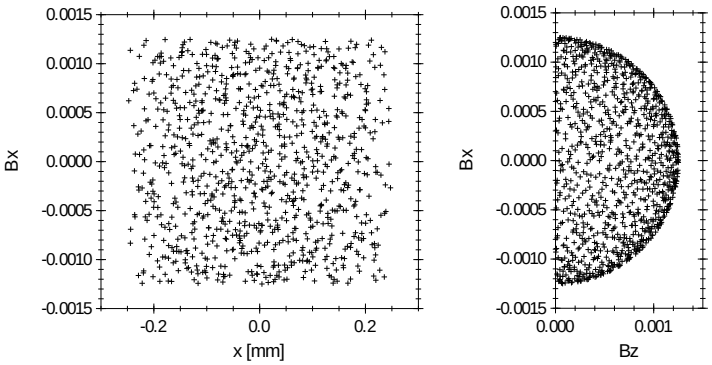


Figure 2–38: Phase space distribution.

Before setting the time-distribution with **settdist**, it is important to check the initial particle distribution. All plots shown before are created with the **snapshot(0)** line and without using **settdist**.

When the simulation is started using the above lines, all particles are emitted from the cathode at t=0. The **settdist** element sets the time-distribution of the particles. The release-time is independent on all other coordinates, creating the possibility to start any particle anywhere, anytime. To make this tutorial simulation a little more interesting, we add a constant 100 MV/m electric field with the **erect** element to accelerate the electrons away from the cathode surface. In an rf photo-gun simulation it would be appropriate to use the **map25D_TM** element.

```
20. # Start in time
21. erect("wcs","I", 1,1,1, -100e6) ; # Constant electric field 100 MV/m
22. settdist("beam", "u", 0, tlen) ;
23. snapshot(-tlen/2, 1.5*tlen, tlen/10) ;
```

The **settdist** keyword in this example sets a uniform distribution centered around zero having a total width of **tlen**. In other words, it starts all particles uniformly between –50 and 50 fs. As a result, the simulation time automatically starts at –50 fs to be able to take all particle trajectories into account. This is observed in the subsequent **snapshot** statement, where the first **snapshot** is requested at **-tlen/2**. In this case, this triggers a warning when running this inputfile because no particle starts at the outer left and right points of a distribution. The resulting particle distribution as function of time is shown in Figure 2–39.

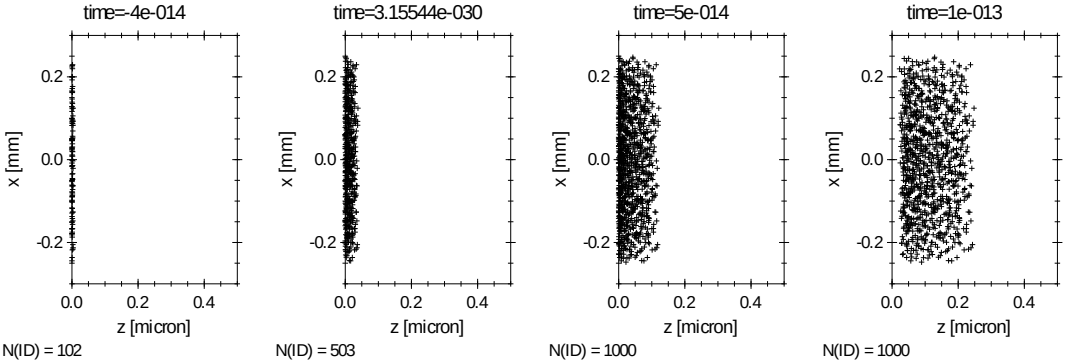


Figure 2–39: Particle distribution at various simulation times. The number of particles linearly increases in the interval between –50 and 50 fs.

To see the charge and particle emission as function of time, the GDFA program can be used:

```
gdfa -o avgs.gdf cathode.gdf time Q numpar
```

The results are shown in Figure 2–40.

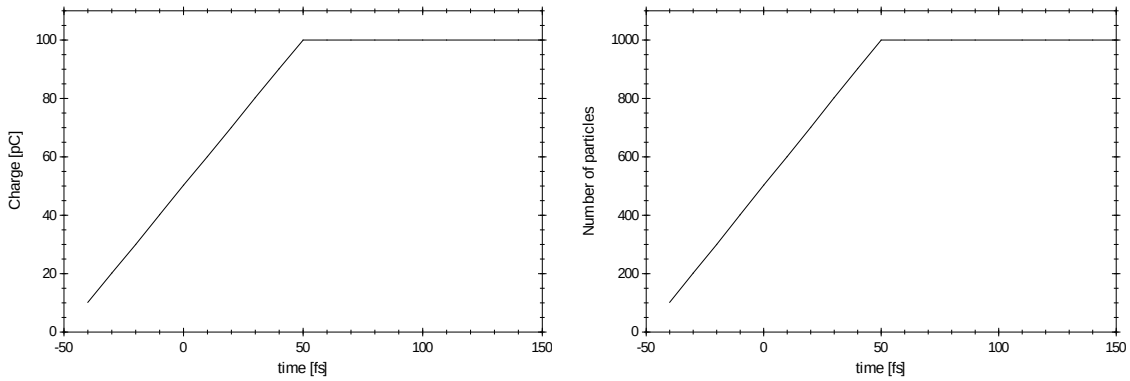


Figure 2–40: Charge (left) and number of particles (right) as function of simulation time.

2.11 Collector design

In this section we demonstrate how to use GPT to design a collector. The result is current and power density on the collector plate(s), where space-charge effects and multiple scattering are included. In this example, an artificial 20 mA, 500 eV electron beam is collected in a simple collector, stretched in y-direction, as shown in Figure 2–41. Section 2.11.7 explains how to design a cylindrical-symmetric collector. The potential on the electrode is –1 kV.

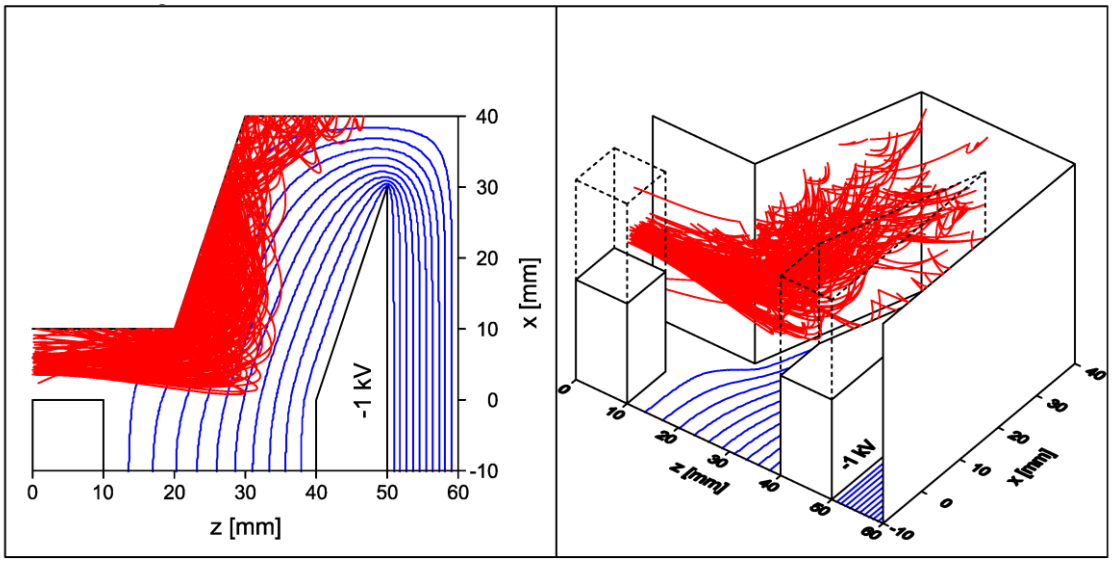


Figure 2–41: Sample collector with particle trajectories. The particle trajectories are imported in a 3D drawing program using the GDF2DXF program.

The files for this tutorial section are located in the folder **`gpt/tutorial/collector`**.

2.11.1 Creating a GPT field-map

In this example, the Poisson code is used to calculate the potential. It is possible to use other field-solvers such as TOSCA or MAFIA. We refer to the ASCI2GDF and RAW2GDF programs for the conversion of the field-output to a GPT field map.

The Poisson-file needed to describe the simple collector is shown in Listing 2–16. Together with all other field-map related files, it can be found in the **`fieldmap`** subfolder of the collector tutorial files. This folder also includes a batch file named **collector.bat** containing all required commands to run Superfish and convert to the output to GPT format.

The commands to run Poisson are:

```
automesh collector.sf
poisson
```

Listing 2–16: collector.sf: Poisson file to describe the collector shown in Figure 2–41.

```
1.  &reg dx=0.1, dy=0.1, conv=1.0, xmin=0,xmax=6, ymin=-1,ymax=4,icylin=0,
2.  nbsup=0,nbslo=1,nbsrt=1,nbslf=0,kprob=0,xjfact=0.0 &
3.
4.  &po x=0,y=-1 &
5.  &po x=6,y=-1 &
6.  &po x=6,y=4 &
7.  &po x=0,y=4 &
8.  &po x=0,y=-1 &
9.
10. &reg mat=0,voltage=0,ibound=-1 &
11. &po x=0,y=1 &
12. &po x=2,y=1 &
13. &po x=3,y=4 &
14. &po x=6,y=4 &
15. &po x=6,y=-1 &
16.
17. &reg mat=0,voltage=0,ibound=-1 &
18. &po x=0,y=0 &
19. &po x=1,y=0 &
20. &po x=1,y=-1 &
21. &po x=0,y=-1 &
22. &po x=0,y=0 &
23.
24. &reg mat=0,voltage=1000.0,ibound=-1 &
25. &po x=4,y=-1 &
26. &po x=4,y=0 &
27. &po x=5,y=3 &
28. &po x=5,y=-1 &
29. &po x=4,y=-1 &
```

The Poisson output cannot be used directly in GPT simulations. It must first be interpolated on a rectangular grid using sf7, and then converted to the GDF format with FISH2GDF:

```
sf7
fish2gdf -o collector.gdf outsf7.txt
```

The sf7 program requires an additional file, **`collector.in7`**, which describes the interpolation parameters as shown in Listing 2–17.

Listing 2–17: collector.in7: interpolation parameters

```
1.  rect noscreen
2.  0 -1 6 4
3.  121 101
4.  end
```

After the above commands are specified, the **`collector.gdf`** field-map is ready for use.

2.11.2 The GPT boundary file

Before the field-map created in section 2.11.1 can be used, the boundaries must be defined in a separate GPT inputfile. When the field-map is created using Poisson, this step is quite simple:

```
fishfile -g boundaries.in -y 1 collector.sf
```

The file **`boundaries.in`** now contains a description of all the boundaries in the **`collector.sf`** file. For a cylindrical-symmetric set-up, the boundaries are defined by **`scattercone`**, **`scattertorus`**, **`scatterpipe`** and other elements. Otherwise, the set-up contains just rectangular **`scatterplate`** elements.

2.11.3 Setting up the simulation

In the file listed in Listing 2–18 a 20 mA, 500 eV rectangular beam is started and send into the collector as described in section 2.11.1. Of course this is not a realistic beam, but serves only as an example.

Listing 2–18: collector.in: Main GPT inputfile for simple collector

```
# Basic beam parameters
Eo   = 500 ;                        # Energy [eV]
G    = 1-qe*Eo/(me*c*c) ;           # Corresponding Lorentz factor G []
Beta = sqrt(1-G^-2) ;               # Corrsponding Normalized velocity []
dE   = 0.2 ;                        # Energy spread [fraction of Eo]
I    = 0.02 ;                       # Beam current [A]

# Simulation parameters
tlen = 1e-9 ;                       # Gun time [s]
nps  = 100 ;                        # Number of particles []
Qtot = -I*tlen ;                    # Total charge in tlen [C]
nmac = (Qtot/qe)/nps ;              # Initial nmacro

# Start a collection of particles (unphysical but simple rectangular beam)
setparticles("beam",nps,me,qe,Qtot) ;
setxdist("beam","u", 5e-3,3e-3) ;
setydist("beam","u", 0e-3,3e-3) ;
settdist("beam","u", tlen/2, tlen ) ;
setGdist("beam","u", G, dE*(G-1) ) ;

# Import collector field-map
Zcol = 0.0 ;
map2Dr_E("wcs","z",Zcol, "fieldmap/fieldmap.gdf","Y","X","Ey","Ex", 1,-Efac ) ;
zminmax("wcs","I", -1e-6, 1 ) ;

# Define scatter model
copperscatter_fix("wcs","z",Zcol, "copper",0.01*nmac,2*nmac) ;

# Spacecharge calculations
spacecharge3D ;

# Output specification
accuracy(5) ;
outputvalue("tlen",tlen) ;          # Used by hierarchical data-analysis
snapshot(0,1e-7,1e-10) ;            # Trajectory plots
# tcontinue(1e-7) ;                 # Data analysis only
```

The field map of the collector is imported using the **map2Dr_E** element. For a cylindrical symmetric collector, **map2D_E** must be used. By convention, the collector is placed at longitudinal position **Zcol**. On the sample disk, the field strength is defined as **Efac** on the GPT command-line to be able to perform a scan in a later stage.

Normally collector simulations need to be performed for (semi) continuous beams. Because GPT is a time-domain code, such a beam is often represented by a much shorter bunch to save CPU time. In the above case, we start a beam with a uniform distribution during the variable **tlen**, equal to 1 ns. For reliable results, only 100 particles are typically insufficient. Because convergence is sometimes much slower, we strongly recommend rerunning the simulation with double the number of particles till the parameters of interest stabilize.

The **copperscatter** element is used to define all the scatter statistics. For a simulation of the primary beam only, the **forwardscatter** element with a probability of 0 can be used instead.

The **snapshot** statement is required for trajectory plotting. Only **tcontinue** is sufficient for power- or current-density analysis.

2.11.4 Trajectory output

The multiple-snapshot option must be selected to plot particle trajectories. The following commands can be used to produce the results as shown in Figure 2–42. Using GPTwin you can open the file **traj.gdf**, load template **traj.gdt** and the result should be identical.

```
gpt -o collector.gdf collector.in fieldmap/boundaries.in
gdftrans -o traj.gdf collector.gdf time x y z G
```

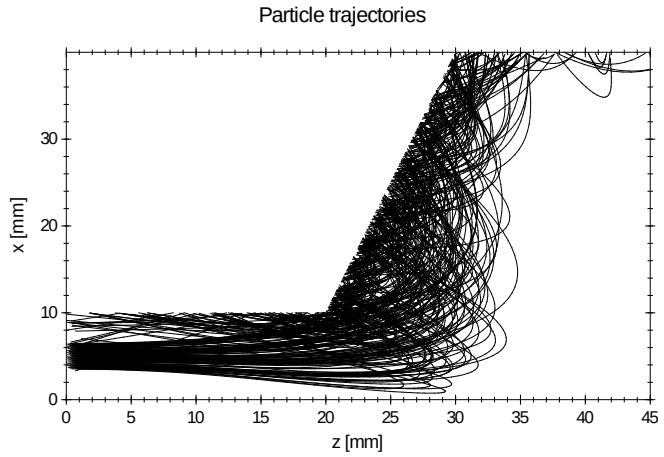


Figure 2–42: Particle trajectories inside the collector. File **`traj.gdf`** plotted with template **`traj.gdt`**.

For trajectory results, the combine-plot option must always be selected. As a result, the primary and secondary particles are always potted in the same graph.

2.11.5 Unrolled position

For power- and current-density calculations, only the **`tcontinue`** output statement is required and the **`snapshot`** keyword can be removed. This stores the 3D scatterstatistics in the outputfile. However, when this file is plotted directly, the results are not much illuminating, see Figure 2–43. For example the trajectory plot indicates that the vertical line at x=40 mm is caused by a z-dependence, but there simply is no reliable way to tell if the power density is lower or higher in that area from these plots.

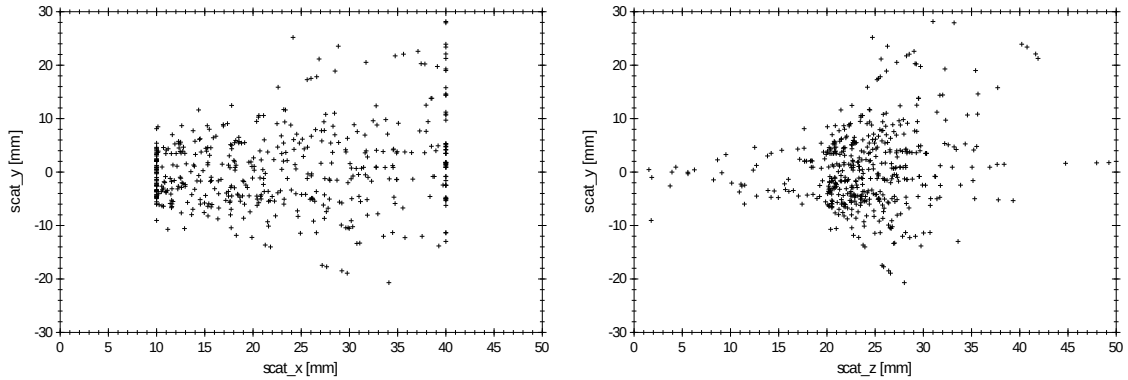


Figure 2–43: Direct output of the scatter elements. File **`collector.gdf`**.with **`tcontinue`** keyword only.

To overcome these problems, we define a coordinate d specifying the 'unrolled' position. This coordinate d runs along a collector plate and keeps measuring the distance along the edge. The unrolled coordinate for our single collector plate is listed in Table 2-B. Because d is monotonically increasing, following the distance of the plate, it is area preserving. As such, it is the most convenient parameter to plot along the horizontal axis for density plots.

Table 2-B: Collector plate coordinates and corresponding unrolled position.

z [mm]	x [mm]	d [mm]
0	10	0
20	10	20
30	40	51
60	40	81

The FISHFILE program can be used again to calculate this 'unrolled position' parameter:

```
fishfile -y 1 -o scatter.gdf fieldmap/collector.sf collector.gdf
```

The result of the above command is the file **`scatter.gdf`**, plotted in Figure 2–44. The vertical lines are completely gone as desired. A new problem is the fact that the scatter elements are able to generate fractions of the incoming particles to keep the statistics correct. In other words, not all dots in the plots

represent an equal amount of particles. Therefore, it is essential to use the density-plot feature of GPTwin using weighted data.

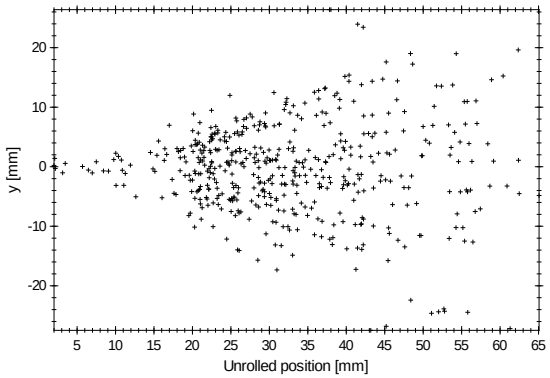


Figure 2–44: Scatter statistics as function of unrolled position. File **`scatter.gdf`** plotted with template **`scatter_d.gdt`**.

2.11.6 Current- and power-density

Creating a density plot using GPTwin is straightforward: Select the density-plot option. Using this option is scientifically correct however can be a challenge. When not enough data is present, it is tempting to increase the blur, but this in turn makes everything Gaussian, independent on the actual distribution.

The important things to do when creating a density plot of power density are:

- Select **`scat_01E`** in the weight_by box. This array represents the total **`E`**nergy for the collector plate **`01`**.
- Weight by bin area
- Set the scaling to manual.
- Set the maximum Energy in the Max box. The ratio between energy and power is the simulation time **`tlen`**. Therefore, in our 1 ns simulation, an energy density of 1 W/m^2 corresponds to a Max energy of 10^{-9}.

When properly done, the scaling adapts itself to the number of bins, is independent on the number of particles in the simulation and follows the units along the x- and y-axes in the plot. When these are both mm, the scaling is to W/mm^2, see Figure 2–45.

The total energy on the collector plate can be calculated using the statistics as Sum(scat_01E). Scaling to power can be done by dividing by **`tlen`** in the scale box.

Current density is completely analogous to power density, except that the **`scat_01Q`** array must be used. It represents the total charge **`Q`** for collector plate **`01`**.

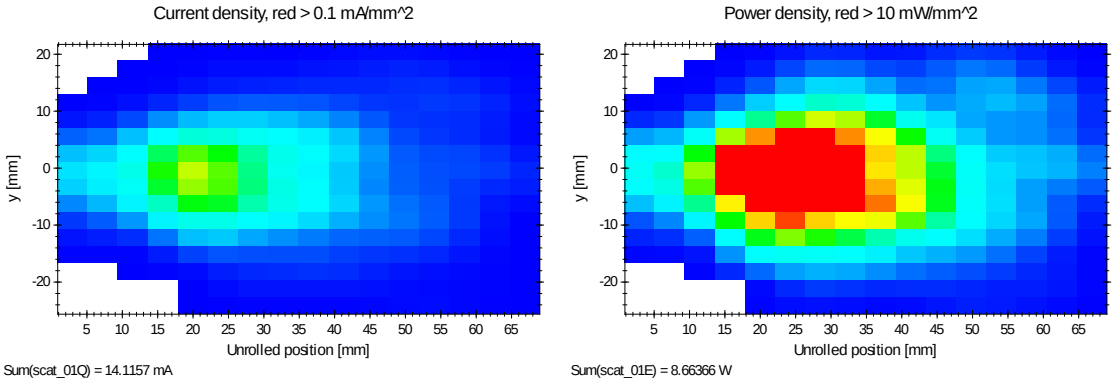


Figure 2–45: Current and power density as function of position on collector plate. File **`scatter.gdf`** plotted with templates **`scatter_Q.gdt`** and **`scatter_P.gdt`** respectively.

2.11.7 A Cylindrical symmetric collector

The GPT simulation of a cylindrical-symmetric cavity is analogous to the previous case. All required modifications are shown in Table 2-C. For a detailed explanation of the FISHFILE output, please see section 3.4.

Table 2-C: Modifications required for a cylindrical-symmetric cavity.

	Rectangular	Cylindrical	Description
GPT Fieldmap	**`map2D_Er`**	**`map2D_E`**	Cylindrical field map.
FISHFILE options	**`-y height`**		Don't specify the height.
Plot arrays (x,y)	**`scat_nnd`**, **`scat_nny`**	**`scat_nnd`**, **`scat_nnphi`**	The angle in the xy-plane is typically plotted.
Density scaling	**`scat_nnE`**, **`scat_nnQ`**	**`scat_nnEor`**, **`scat_nnQor`**	For correct scaling, E/r and Q/r must be used.

2.11.8 Scanning a parameter

Practically any parameter can be scanned with the MR program. The typical use is running GDFA on the output of the FISHFILE program to directly obtain total current and power statistics per collector plate as function of the scanned parameter. The **`scatterInn`** and **`scatterPnn`** programs can be used to calculate plot total current and power for the first ten collector plates. For example, the following commands calculate current and power on the first collector plate as function of potential. The results are shown in Figure 2–46.

```
mr -v -o collector.gdf Efac.mr gpt collector.in fieldmap/boundaries.in
fishfile -y 1 -o scatter.gdf fieldmap/collector.sf collector.gdf
gdfa -o stats.gdf scatter.gdf Efac scatterI01 scatterP01
```

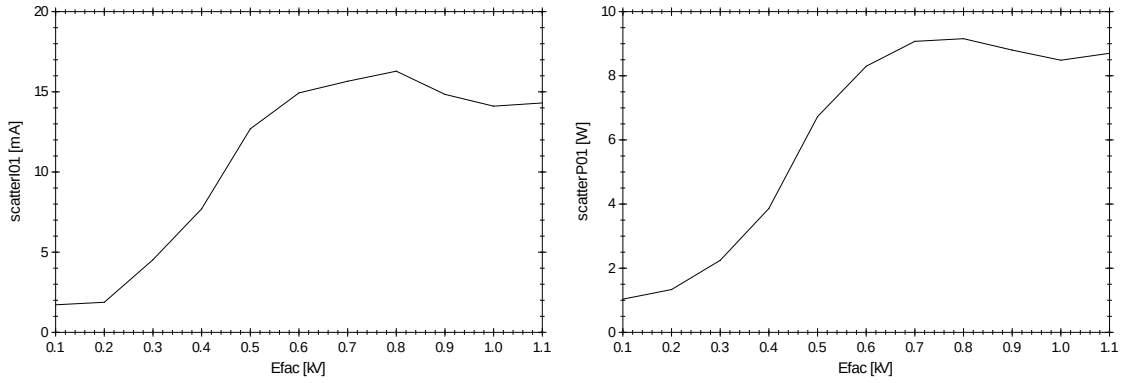


Figure 2–46: Total current and power on first collector plate as function of potential. File **`avgs.gdf`** plotted with templates **`avgs_Q.gdt`** and **`avgs_P.gdt`** respectively.

3 GDF

GPT Datafile Format, GDF, is a file format developed for programs producing large amounts of numerical data. It is the native output format of the General Particle Tracer, GPT, allowing it to automatically scan parameters and perform efficient data processing and visualization. GDF-files are machine independent and can be transferred between PC's and UNIX machines.

This chapter describes the basics of a GDF file and the following utility programs:

ASCI2GDF	Converts an ASCII file to a GDF file.
FISH2GDF	Converts SUPERFISH output to a GDF file.
FISHFILE	Calculates "unrolled" scatter statistics or creates a GPT inputfile based on a Superfish file.
GDF2A	Writes the contents of a GDF file in ASCII to the terminal or a file. The file can be processed by an ASCII-based editor or imported into a spreadsheet program.
GDF2DXF	Converts a GDF file to a point, line or vector drawing for further customization in a 2D or 3D drawing package.
GDF2GDF	Combines several GDF files into a single file, optionally sorting the top-level group.
GDF2HIS	Calculates histograms from the contents of a GDF file. The output is in GDF format and can be plotted or converted to ASCII.
GDF2SDDS	Converts a GDF file to an SDDS file.
GDFA	Removes one level of hierarchy in a GDF file by performing user-specified calculations. The calculations typically produce averages and standard deviations of arrays, reducing the total amount of data considerably.
GDFSOLVE	Multi-dimensional root finder and optimizer for GPT.
GDFTRANS	Transpose a GDF file to calculate trajectories as function of any parameter.
MR	Scan one or more parameters and concatenate the output in a single hierarchical file
MPIMR	MPI version of MR that distributes parameter scans over several PC's.
RAW2GDF	Converts a raw binary datafile to a GDF file.

3.1 Basics

A GDF file is a file containing binary data in a structured form. Basically, a GDF file consists of a general header followed by one or more data blocks. Every block has a name and contains data. Almost all native C/FORTRAN data types are supported. A simple GDF file is shown schematically in Figure 3–1.

General header	
string	"This is a string"
number	12
value	1234e56
array	{1.1,2.1,3.1,4.1,5.1}

Figure 3–1: Simple GDF file

Such a list of named blocks containing all kinds of data is impractical when the number of blocks becomes excessively large. Therefore the blocks are stored hierarchically.

As an example consider the output of the particle tracer GPT. It needs to store particle coordinates at different simulation times. The particle coordinates are stored as separate arrays containing the coordinates of the individual particles. Because the particle coordinates are calculated at different times, the blocks are grouped together by time. A (simplified) GPT outputfile is shown schematically in Figure 3–2.

General header		
time	1	
	x	{x1,x2,x3,...}
	y	{y1,y2,y3,...}
	z	{z1,z2,z3,...}
time	2	
	x	{x1,x2,x3,...}
	y	{y1,y2,y3,...}
	z	{z1,z2,z3,...}

Figure 3–2: GDF-output of the particle tracer GPT.

Groups can be nested. Suppose, for example, that the coordinates of the particles are calculated at two times with a different value for the variable **var**. The GDF file can then be represented as shown in Figure 3–3.

General header		
var	4	
	time	0
		particle coordinates
	time	1
		particle coordinates
var	5	
	time	0
		particle coordinates
	time	1
		particle coordinates

Figure 3–3: GDF-file containing nested groups.

The MR program, see section 3.13, is used to create groups based on parameter scans, while the GDFA program, see section 3.10, is used to flatten the hierarchy by one level to ease interpretation of the results.

3.2 ASCI2GDF

asci2gdf [-hHiv] [-c chars] [-s number] [-o outfile] [-a outfile]
** infile [arr1 fac1] ...**
ASCII file to GDF converter.

-h	Give short help and quit.
-H	Give long help and quit.
-i	Ignore error lines and continue.
-v	Verbose mode. Progress and file statistics are shown.
-c chars	Ignore lines starting with any of **chars**. Default "**#;**".
-s number	Skip the first **number** of lines.
-o outfile	Write output to **outfile**.
-a outfile	Append output to **outfile**.
infile	ASCII file to be converted.
arr*N* fac*N*	Name of the array(s) as listed in **infile**, directly followed by their multiplication factor.

ASCI2GDF converts a tabulated ASCII file to a GDF file. The resulting GDF file can be read by the field map elements and by all GPT utility programs capable of handling GDF files. ASCI2GDF writes all columns that are present in **infile** in the **outfile**, except for columns with a specified multiplication factor **fac** of 0. Non-zero multiplication factors can be used to convert from non-SI units to SI units. When the original column is already in SI units, neither the name nor its multiplication factor need to be specified on the commandline. All lines starting with a hash mark '**#**' and all blank lines are ignored. To test if the resulting file is correct, the GDF2A program can be used.

The ASCII file to be converted must have the following structure:

```
1. # Comment section
2. name1  name2  name3  name4
3. data   data   data   data
4. data   data   data   data
5. …      …      …      …
```

The following commandline converts the above-listed file **sample.txt** to GDF:

```
asci2gdf -o sample.gdf sample.txt
```

The conversion can be verified with GDF2A. Without the **-o** option the output appears on the screen:

```
gdf2a sample.gdf
```

The following commandline instructs ASCI2GDF to multiply array **name2** by 100 and to omit **name4** in the resulting GDF file. Because **name1** and **name3** are not specified, they are converted to the outputfile with multiplication factor 1.

```
asci2gdf -o sample.gdf sample.txt name2 100 name4 0
```

3.3 FISH2GDF

`fish2gdf [-hHv] [-o outfile] [-a outfile] infile`
Converts SUPERFISH output to a GDF file.

`-h`	Give short help and quit
`-H`	Give long help and quit
`-v`	Verbose mode
`-o outfile`	Write output to outfile
`-a outfile`	Append output to outfile
`infile`	Name of the SF7 file to convert to GDF

FISH2GDF is the first part of the interface between the SUPERFISH set of codes and GPT. Electrostatic, cylindrical symmetric magnetostatic field profiles and cylindrical symmetric rf cavities can be imported into GPT using FISH2GDF and the GPT elements listed in Table 3-A. The reference for the GPT field-map elements is given in section 4.8 and for a description of the SUPERFISH codes, we refer to [3F4].

Table 3-A: GPT Field map elements for various SUPERFISH solvers.

Problem	SuperFish Solvers	GPT elements
Electrostatic	POISSON/PANDIRA	**`map2D_E`**, **`map2Dr_E`**
Magnetostatic	POISSON/PANDIRA	**`map2D_B`**
RF Cavity	FISH	**`map25D_TM`**

The different SUPERFISH programs output the results in different units. For example electric field strength is calculated in V/cm for electrostatic problems, while MV/m is used for cavity calculations. FISH2GDF converts all units to SI using the conversion factors listed in Table 3-B.

Table 3-B: Conversion factors to SI.

SF unit	New (SI) unit	Conversion factor	Type
Cm	m	10^{-2}	All
G	T	10^{-4}	Magnetostatic
G-cm	Tm	10^{-6}	Magnetostatic
MV/m	V/m	10^{6}	Cavity
A/m	T	$\mu_0=1.25664\cdot10^{-6}$	Cavity
V/cm	V/m	10^{2}	Electrostatic
V	Volt	1	Electrostatic

Some columns in the SUPERFISH output contain characters in the upper 128 characters of the MS-Linedraw character set. FISH2GDF converts these names to regular ASCII using Table 3-C.

Table 3-C: Conversion from SUPERFISH to ASCII names

SF Name in MS-LineDraw	GDF Name
|E|	**`|E|`**
|B|	**`|B|`**
Bφ	**`Bphi`**
Aφ	**`Aphi`**

To be able to use FISH2GDF, first a problem-description file for SUPERFISH must be created. After running AUTOMESH and one of the SUPERFISH solvers, the results can be inspected using VGAPLOT. The calculated field profile must then be interpolated to a rectangular grid and converted to the GDF format using SF7 and FISH2GDF. The resulting GDF file can be imported into GPT using one of the corresponding field-map elements. The following batch file automates the whole process:

```
1. automesh infile
2. poisson / pandira / fish
3. vgaplot              (Optionally to inspect the results)
4. sfo                  (Optionally to normalize the fields)
5. sf7
6. fish2gdf  -o fieldmap.gdf outsf7
7. gdf2a fieldmap.gdf   (Optionally to inspect the results)
```

where the file **infile.in7** specifies the interpolation range.

```
1. rect noscreen              rect noscreen
2. rmin zmin rmax zmax        xmin ymin xmax ymax
3. Nr Nz                      Nx Ny
4. end                        end
```

The following line in the GPT inputfile positions a magentostatic field map at z=1m.

```
map2D_B("wcs","z",1,"fieldmap.gdf", "r","z", "Br","Bz", 1.0 ) ;
```

3.4 FISHFILE

fishfile [-hHvr] [-o outfile] [-t tolerance] [-g GPTfile] [-y height] [-z offset] fishfile [scatfile.gdf]

Calculates 'unrolled' scatter statistics or creates a GPT inputfile based on a Superfish file.

-o outfile	Write output to a file named **outfile** containing the 'unrolled' scatter statistics and other useful information
-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose output.
-t tolerance	The maximum tolerance [m] between the Superfish geometry and the scattered data. This option is useful when the GPT inputfile doesn't precisely match the superfish file. The default tolerance is 1 micron.
-g GPTfile	Writes a GPT scatter file based in the Superfish geometry as specified in **fishfile**.
-y height	Specifies the y-height [m] for a rectangular geometry. When a GPT inputfile is generated the boundary elements range from *y*=**-height**/2 to *y*=**height**/2. When not specified, the geometry is assumed to be cylindrical symmetric.
-z offset	Subtract the specified offset [m] from the data before starting the analysis.
fishfile	Superfish file describing the cylindrical symmetric geometry. The first region is assumed to be the boundary and is ignored.
scatfile.gdf	GPT outputfile with scatter statistics to be 'unrolled'.

The arrays written in **outfile.gdf** are written per plate number **nn**. They are:

scat_nnx	x-coordinate of primary particle [m].
scat_nny	y-coordinate of primary particle [m].
scat_nnz	z-coordinate of primary particle [m].
scat_nnE	Energy of primary particle minus the energy of all scattered particles [J].
scat_nnQ	Charge of primary particle minus the charge of all scattered particles [Q].
scat_nnd	Unrolled position [m].

For cylindrical symmetric geometries, the following arrays are also written:

scat_nnphi	Angle in xy-plane [rad], measured counterclockwise from the x-axis.
scat_nnr	Distance to z-axis [m].
scat_nnEor	Identical to scat_nnE divided by scat_nnr.
scat_nnQor	Identical to scat_nnQ divided by scat_nnr.
scat_nnrphi	Identical to scat_nnr times scat_nnphi.

Fishfile is the main program to analyze the results of a collector simulation using GPT. The raw output of GPT consists of the 3D collision point and the charge and energy of both the primary and the scatted particles, as described in section 1.11.2. Interpreting these results is time-consuming and difficult. The Fishfile program automates most analysis parts using the concept of 'unrolled' data. Furthermore, Fishfile is capable of generating a GPT file containing the boundary elements from a Superfish inputfile.

For cylindrical-symmetric geometries, the **x** and **y** coordinates in the Superfish file are converted to **r** and **z** respectively. When the **-y** option is used, the geometry is assumed to be cartesian and the **x** and **y** coordinates are converted to **z** and **x** respectively. In both cases, the units are converted from [cm] to [m].

3.5 GDF2A

gdf2a [-hHv] [-w width] [-o outfile] infile [arrays...]
Writes a GDF-file in ASCII form to the terminal or to a file.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Instructs the program to write additional information, such as data type.
-w width	Specify the width (in characters) of each field. Default 10.
-o outfile	Write output to a file named **outfile**. If not specified, output is written to **stdout**.
infile	Filename of the GDF file to convert to ASCII. If not specified, the program reads the GDF file from **stdin** unless **arrays** are listed. In the latter case, a dash ("**-**") must be specified to read from from **stdin**.
arrays	The names of the arrays to be written to **outfile**. If not specified, all arrays are written.

Note: To reduce the size of the ouputfile, the default number of digits written is set to single precision. The **-w 16** option can be used to instruct GDF2A to write output in double precision.

Because GDF2A can read from **stdin** and write to **stdout**, it can be piped after any command that writes a GDF file to **stdout**.

3.6 GDF2DXF

gdf2dxf [-hHvlp] [-f scaling] [-o outfile] inputfile x y [z] [fx fy [fz]]
Converts a GDF file to a point, line or vector drawing for further use in a 2D or 3D drawing package.

-h	Give short help and quit
-H	Give long help and quit
-v	Verbose mode
-l	Output lineplots
-p	Output scatterplots (default)
-o outfile	Write output to outfile
infile	Name of the file to perform the calculations on. When GDF2DXF has to read from **stdin**, a dash ("**-**") must be specified explicitly for **infile**.
x	The name of the array specifying the x-coordinates.
y	The name of the array specifying the y-coordinates.
z	Optional name of the array specifying the z-coordinates. If this array is specified, the drawing package must be able to handle 3D data.

For vector plots, the following additional options are available:

-f scaling	Vector plot only: Specify the scaling for the vector length.
fx	Optional name of the array specifying the x-direction for vector plots.
fy	Optional name of the array specifying the y-direction for vector plots.
fz	Optional name of the array specifying the z-direction for vector plots.

GDF2DXF can be used to export GDF data to the DXF format. The latter format is recognized by almost all drawing packages and DTP programs currently available. It allows you for example to plot simulated particle trajectories through a schematic drawing of your set-up. For example to import the zx-plane of a GDF file as a line drawing into AutoCAD, the following commands can be specified:

```
GDF2DXF -l -o outfile.dxf infile.gdf z x
```

Start AutoCAD

```
dxfin outfile.dxf
```

3.7 GDF2GDF

gdf2gdf [-hHvis] [-t group] [-o outfile] [-a outfile]
infile1 [value1] [infile2 [value2]] ...
Combines several GDF files into a single file, optionally sorting the top-level group.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Progress and file statistics are shown.
-i	Ignore recoverable errors and continue.
-s	Sort top-level group.
-t group	Top-level group.
-o outfile	Write output to **outfile**.
-a outfile	Append output to **outfile**.
infile[s]	Names of all the GDF files to be concatenated
value[s]	Optional values for the top-level group.

GDF2GDF combines several GDF files into a single large one. If **file1.gdf**, **file2.gdf** and **file3.gdf** have the exact same hierarchy, they can be combined into **all.gdf** with the following commandline:
gdf2gdf –o all.gdf file1.gdf file2.gdf file3.gdf.

Things get a little more complicated if the files to be combined do not have the same hierarchy. A typical scenario for this is when a file produced by MR needs to be combined with the result of a single additional run. Suppose that the MR file **var135.gdf** contains the simulation results for the variable **var** being 1, 3 and 5 and that the single run, **single4.gdf**, is created for **var** equal 4. These files can be combined, but only if the top-level group is specified with the **–t** option:
gdf2gdf –t var –o all.gdf var135.gdf single4.gdf

GDF2GDF will try to extract the value of **var** from **single4.gdf** to rearrange the hierarchy, but this only works if the file has been produced by MR. If the file has been produced by GPT, e.g. **gpt -o single4.gdf inputfile.in var=4**, this will not work. In that case, the value for the top-level group must be given directly after the filename on the commandline as follows:
gdf2gdf –t var –o all.gdf var135.gdf single4.gdf 4

Even this may not produce the desired result, as the ordering for **var** will become 1, 3, 5, 4. If sorted output is desired, please use the **–s** option to sort the top-level domain ascending order.
gdf2gdf -s –t var –o all.gdf var135.gdf single4.gdf 4

Technical note: When the inputfiles are huge, GDF2GDF first locks them, and subsequently reads them all storing only the hierarchy. The actual data is not stored but recorded as 64-bit file-offsets into the locked inputfiles. The next step builds the structure of the outputfile in memory by concatenating the hierarchy of the inputfiles. If needed, one level of hierarchy is added for single runs, and optionally the top-level domain is sorted. As a last step, the outputfile is written as one set of consecutive write operations, where each time the data is fetched from the appropriate inputfile at the stored location.

3.8 GDF2HIS

gdf2his [-hHvbls] [-n weight] [-o outfile] infile array width [offset]
Calculates histograms of the specified array. For every particle in this array GDF2HIS determines which interval of the histogram it belongs to and raises the count for that bar by one.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Displays the number of bars in the histogram for every group in the **infile**.
-b	Write the complete histogram. When this option is selected, the output can be plotted as a lineplot outlining the individual bars of the histogram.
-l	Write the outline of the histogram. When plotted as a lineplot only the outline of the histogram is shown, not the individual bars.
-s	Divide the height of the histograms by the specified **width**.
-n weight	Count all contributions as specified in the **weight** array. For GPT simulations, **nmacro** can be used to count the number of elementary particles.
-o outfile	Write output to a file named **outfile**. If not specified the output is written to **stdout**.
infile	Name of the file containing the array to calculate histograms of. When GDF2HIS has to read from **stdin**, a dash ("**-**") must be specified explicitly for **infile**.
array	Name of the array used to calculate the histogram. Typical examples are **z** and **G**.
width	Width of an interval within the histogram.
offset	The 'left-right' offset of the bars. If not specified, 0 is assumed, indicating one bar to the left side of zero and one to the right.

When GDF2HIS is run, a histogram is calculated for every group in the GDF file containing the specified array. In these groups, the following two arrays are appended:
array_his: Array of the positions of the centers of the intervals.
array_cnt: Array of the number of datapoints within one interval.
The string **array** is replaced by the name of the array specified on the commandline. The range of the histograms is obtained automatically from the minimum and maximum values of the data.

When the **-b** option is selected on the commandline, the arrays **array_hisbar** and **array_cntbar** are written in the ouputfile as well. When these two arrays are plotted as a xy-lineplot, the bars of the histogram are drawn.

When the **-l** option is selected, the arrays **array_hisline** and **array_cntline** are also written in the ouputfile. When these two arrays are plotted as a xy-lineplot, the outline of the histogram is drawn.

Example:
Suppose **infile.gdf** is a file containing all particle coordinates at different times. To calculate a histogram of the energy distribution in steps of 0.01, the following command can be specified:
gdf2his -b -o outfile.gdf infile.gdf G 0.01
This produces the file **outfile.gdf** where the arrays **G_cntbar** and **G_hisbar** are appended for every GDF group containing the particles. The outputfile should be plotted using a line plot, Figure 3–4.

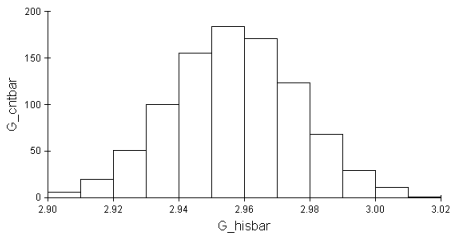


Figure 3–4: Example of GDF2HIS. The gamma's of the particles of a bunch are converted to a histogram to show the energy distribution.

3.9 GDF2SDDS

gdf2sdds [-hHva] [-o outfile] infile [arrays ...]
Converts a GDF file to the SDDS format.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode, instructs the program to write additional information during processing.
-a	Writes an ASCII SDDS file. When not specified, a binary file is written. Before use, please see the note below about precision.
-o outfile	Write output to a file named **outfile**. If not specified, output is written to **stdout**.
infile	Filename of the GDF file to convert to ASCII. If not specified, the program reads the GDF file from **stdin**. To read from **stdin** when **arrays** are specified, a dash ('**-**') must be specified.
arrays	The names of the arrays/parameters to convert. If not specified, all GDF information is converted to the SDDS format.

This program converts a GDF file to a file in the Self Describing Data Set (SDDS) format [4F5]. The SDDS format is supported by a toolkit consisting of a number of powerful and efficient programs for analyzing, manipulating and plotting SDDS data. The SDDS toolkit can be downloaded free of charge from Argonne National Laboratory.

The GDF2SDDS program writes GDF arrays as column data in SDDS format. Single values in GDF are converted to parameters in SDDS. When one or more **arrays** are specified, only the selected arrays are written to SDDS. GDF group names such as time, position and the values of one or more scanned parameters are always written to SDDS. As dictated by the SDDS column format, all specified arrays must have equal length.

To convert the x-, y- and z-coordinates in the GDF file **test.gdf** to the SDDS file **test.sdds**, the following command can be used:

```
gdf2sdds –o test.sdds test.gdf x y z
```

Because GDF2SDDS writes to **stdout** when no output filename is given, it can be piped before any command that reads an SDDS file from **stdin**. This allows efficient use of the **–pipe=in** option of many of the SDDS toolkit programs, as demonstrated in the following example:

```
gdf2sdds test.gdf | sddsquery –pipe=in
```

Note: The **-a** option writes an ASCII SDDS file. To reduce the size of the outputfile, double-precision floating-point numbers are written with only 8 digits precision. Because this causes loss of precision, it is recommended not to use this option except for diagnostic purposes.

3.10 GDFA

gdfa [-hHv] [-o outfile] [-a outfile] infile group_by prog ...
Writes a GDF file containing the results of performed calculations.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Displays nothing extra at the moment but is implemented for future compatibility.
-o outfile	Write output to a file named **outfile**. If not specified the output is written to **stdout**.
-a outfile	Append output to a file named **outfile**. This option is for internal use only.
infile	Name of the file to perform the calculations on. When GDFA has to read from **stdin**, a dash ("**-**") must be specified explicitly for **infile**.
group_by	The name of the variable used to group the data. Use **time** for **snapshot/tout** output and **position** for **screen** output. When **MR** is used, you can also use the name of the scanned parameter.
prog ...	The name(s) of one or more programs to run on the **infile**. See the following sections for a complete overview. The GPT programmer's reference describes the procedure how to create custom GDFA programs.

The following GDFA programs are currently available:

To explain GDFA we continue with the example of Figure 3–3. For convenience, we now represent this GDF file in the form shown in Figure 3–5.

	time=0	**time=1**
var=4	**Particle coordinates**	**Particle coordinates**
var=5	**Particle coordinates**	**Particle coordinates**

Figure 3–5: GDFA representation of a GDF-file containing two nested groups.

Firstly, GDFA runs the specified **prog**s on the data of the **infile**. A **prog** is a function of the data and returns a scalar. **stdz** for example returns the standard deviation of the array of the z-coordinates. After the first pass, the data can be represented as shown in Figure 3–6. For a complete list of the **prog**s currently available we refer to the following sections.

	time=0	**time=1**
var=4	**stdz**	**stdz**
var=5	**stdz**	**stdz**

Figure 3–6: GDFA representation of a GDF-file containing two nested groups.

Secondly, GDFA eliminates the **group_by** group by making arrays of the results returned by the **prog**s. Continuing with our example, we have two possible **group_by** variables: **time** and **var**. The results when grouped by time and var are both shown in Figure 3–7.

When **time** is chosen, the **var** hierarchy is maintained and the result is shown schematically in Figure 3–7. When **var** is chosen, the **time** hierarchy is maintained and the result is shown schematically in Figure 3–7.

General header		
var	**4**	
	stdz	**{.,.}**
	time	**{0,1}**

General header		
time	**0**	
	stdz	**{.,.}**
	var	**{4,5}**

```
var     5
        stdz    {.,.}
        time    {0,1}
```

```
time    1
        stdz    {.,.}
        var     {4,5}
```

Figure 3–7: Grouped by **time** (left) and grouped by **var** (right).

The **group_by** variable itself is always included as an array. If more than one **prog** is specified, GDFA makes separate arrays for the results of all the **prog**s. If one **prog** fails to calculate its result, the results of all the other **progs** are ignored too.

Custom GDFA programs can be created similar to custom GPT elements. The procedure is described in the separate GPT programmer's reference.

We would like to thank Dr A. Büchner for providing us with the initial versions of the emittance and Courant-Snyder routines and allowing us to share them with other GPT users.

3.10.1 Averages

All averages are weighted by the number of elementary particles each macro-particle represents and calculated using:

$$\text{avg}(x) = \bar{x} = \frac{\Sigma N_i x_i}{\Sigma N_i}$$

Name	Equation	Description
avgx	$\bar{x}$	Average *x*-coordinate [m].
avgy	$\bar{y}$	Average *y*-coordinate [m].
avgz	$\bar{z}$	Average *z*-coordinate [m].
avgr	$\Sigma N_i \sqrt{(\mathbf{x}_i - \bar{\mathbf{x}}_i)^2 + (\mathbf{y}_i - \bar{\mathbf{y}}_i)^2} \,/ \Sigma N_i$	Average *r*-coordinate [m].
avgBx	$\overline{\beta x}$	Average βx-coordinate
avgBy	$\overline{\beta y}$	Average βx-coordinate
avgBz	$\overline{\beta z}$	Average βx-coordinate
avgG	$\bar{\gamma}$	Average γ
avgt	$\bar{t}$	Average time [s] useful for position output.
avgfEx	$\overline{Ex}$	Average *Ex*-field [V/m].
avgfEy	$\overline{Ey}$	Average *Ey*-field [V/m].
avgfEz	$\overline{Ez}$	Average *Ez*-field [V/m].
avgfBx	$\overline{Bx}$	Average *Bx*-field [T].
avgfBy	$\overline{By}$	Average *By*-field [T].
avgfBz	$\overline{Bz}$	Average *Bz*-field [T].
avgp	$\frac{\beta_0}{\sqrt{1-\beta_0^2}} \frac{mc}{\lvert e \rvert}$ with $\beta_0 = \frac{\overline{\gamma\beta}}{\bar{\gamma}}$	Mean beam momentum [eV-s/m].

3.10.2 Standard deviations

All standard deviations are weighted by the number of elementary particles each macro-particle represents. Standard deviations are calculated by:

$$\text{std}(x) = \sqrt{\overline{(x - \bar{x})^2}}$$

Name	Equation	Description
stdx	$\mathrm{std}(x) = \sqrt{\overline{(x-\bar{x})^2}}$	Standard *x*-deviation [m].
stdy	std(*y*)	Standard *y*-deviation [m].
stdz	std(*z*)	Standard *z*-deviation [m].
stdBx	std(βx)	Standard βx-deviation
stdBy	std(βy)	Standard βy-deviation
stdBz	std(βz)	Standard βz-deviation
stdG	std(γ)	Standard γ-deviation
stdt	std(*t*)	Standard time deviation [s].

3.10.3 RMS Emittance routines

In our opinion, the normalized transverse emittance can best be calculated as the area in position and transverse momentum coordinates: *x* and $\gamma\beta_x$. However, to remain compatible with other codes, the emittance in the following GDFA programs is calculated in velocity space, *x* and β_x,. This value is multiplied by the average of the Lorentz factor γ for normalization. Without energy spread, these results are identical. However, when there is a relatively large energy spread, or when the phase space is far from elliptical, the results can be significantly different. See the GPT programmer's reference to adapt the routines to your own taste.

In all routines, the emittance is defined as the phase-space area divided by π. Therefore, the units of emittance are often presented as [π mm-mrad]. The factor π in the dimension is included to assure the reader that π has been factored out of the phase-space area.

The SI units for transverse and longitudinal emittance are [m-rad] and [J-s] respectively. The transverse emittance routines **nemixrms**, **nemiyrms**, **nemirrms**, **nemix90**, **nemiy90**, **nemix100** and **memiy100** calculate their results in [m-rad]. The longitudinal emittance is calculated in [eV-s] by **nemizrms**, **nemiz90** and **nemiz100** for convenience. To convert **nemixrms** from its SI result to [π mm-mrad] one should multiply by 1,000,000: One time by 1000 to convert from [m] to [mm] and one time by 1000 to convert from [rad] to [mrad].

To calculate the normalized RMS-emittances, first x_c, y_c, x'_c and y'_c are calculated by GDFA using:

$$x_c = x - \bar{x} \qquad y_c = y - \bar{y}$$

$$x'_c = \beta_x - \bar{\beta}_x \qquad y'_c = \beta_y - \bar{\beta}_y$$

The quantities *x'* and *y'* are directly multiplied by γ to provide correct normalization for beams with high energy spread and high divergence angles. All parameters are centered around zero.

The RMS emittance for the *x*-*x'* and *y*-*y'* phase spaces are defined by:

$$\mathbf{nemixrms} = \bar{\gamma}\sqrt{\overline{x_c^2}\cdot\overline{x'^2_c} - \overline{x_c x'_c}^2}$$

$$\mathbf{nemiyrms} = \bar{\gamma}\sqrt{\overline{y_c^2}\cdot\overline{y'^2_c} - \overline{y_c y'_c}^2}$$

When a cylindrical symmetric beam rotates around the z-axis, such as in a solenoidial field, strong x-y' and x'-y correlations result in very large values for the rms x- and y-emittances. This does not signal phase-space dilution or loss of beam quality in any way. It is just an artifact of the numerical method used to calculate the emittance. The emittance decreases again when the solenoidal field vanishes. This effect can be reduced by calculating the *r*-emittance, using the hypervolume in *x*-*x'*-*y*-*y'* subspace, to calculate the emittance using:

$$\mathbf{nemirrms} = \bar{\gamma}\sqrt{\varepsilon_{x,rms}\cdot\varepsilon_{y,rms} - \left|\overline{x_c y_c}\cdot\overline{x'_c\, y'_c} - \overline{x_c y'_c}\cdot\overline{x'_c\, y_c}\right|}$$

The units of **nemixrms**, **nemiyrms** and **nemirrms** are emittances in [m-rad]. However, the values are typically multiplied by 1,000,000 and quoted as [π mm-mrad]. Also please note that the results will be a factor of 4 less than the results produced by the now-obsolete **nemix** and **nemiy** programs.

The longitudinal emittance is calculated by:

$$t_c = t - \bar{t}$$
$$\gamma_c = \gamma - \bar{\gamma}$$
$$\mathbf{nemizrms} = \frac{mc^2}{|q_e|}\sqrt{\overline{t_c^2}\cdot\overline{\gamma_c^2} - \overline{t_c\gamma_c}^2}$$

The factor mc^2/q_e is used to convert the units to [eV-s].

When time output is used, all particle coordinates have the same time coordinate. In this case, the time coordinate is obtained by extrapolating the centered position using $t_i = -(z_i - \bar{z})/c\beta_{z,i}$.

3.10.4 90% and 100% Emittance routines

As an alternative to the RMS emittance, one can define 'per-particle' emittance values: Every particle is at the boundary of an ellipse with the same orientation and shape as the 'average ellipse', but with a different size. The area/π of this ellipse is the emittance of the particle.

The calculations are estimates, based on the overall ellipse parameters calculated by the Courant-Snyder parameters. As such, they do not necessarily produce the smallest possible ellipse enclosing 90% or 100% of the particles. When distributions are far from elliptical in phase space, these routines can not be used.

First the following parameters are calculated using all particles:

$$a_x = -\overline{x_c x'_c} \quad b_x = \overline{x_c^2} \quad c_x = \overline{x'^2_c} \quad e_x = \sqrt{b_x c_x - a_x^2}$$
$$a_y = -\overline{y_c y'_c} \quad b_y = \overline{x_c^2} \quad c_y = \overline{y'^2_c} \quad e_y = \sqrt{b_y c_y - a_y^2}$$
$$a_z = -\overline{t_c \gamma_c} \quad b_z = \overline{t_c^2} \quad c_z = \overline{\gamma_c^2} \quad e_z = \sqrt{b_z c_z - a_z^2}$$

The a, b, c parameters are very similar to the Courant-Snyder parameters α, β, γ. The 'per-particle' emittance is then given by:

$$\varepsilon_{x,i} = \left(c_x(x_i - \bar{x})^2 + 2\,a_x(x_i - \bar{x})(x'_i - \bar{x}') + b_x(x'_i - \bar{x}')^2\right)/e_x$$
$$\varepsilon_{y,i} = \left(c_y(y_i - \bar{y})^2 + 2\,a_y(y_i - \bar{y})(y'_i - \bar{y}') + b_y(y'_i - \bar{y}')^2\right)/e_y$$
$$\varepsilon_{z,i} = \left(c_z(t_i - \bar{t})^2 + 2\,a_z(t_i - \bar{t})(\gamma_i - \bar{\gamma}) + b_z(\gamma_i - \bar{\gamma})^2\right)\frac{mc^2}{|q_e|\,e_z}$$

The **nemi*100** programs calculate the emittance of the worst offender, equal to the area/π of the ellipse enclosing all particles:

$$\mathbf{nemix100} = \max(\varepsilon_x)$$
$$\mathbf{nemiy100} = \max(\varepsilon_y)$$
$$\mathbf{nemiz100} = \max(\varepsilon_z)$$

The **nemi*90** programs calculate the area/π of the smallest ellipse enclosing 90% of the particles. It provides more stable results than the 100% values, but it should be noted that all particles are used in the determination of the phase-space ellipse dimensions and orientation.

$$\mathbf{nemix90} = \varepsilon_{x,i} \mid 90\%\text{ of particles } j \text{ have} \varepsilon_{x,j} \le \varepsilon_{x,i}$$
$$\mathbf{nemiy90} = \varepsilon_{y,i} \mid 90\%\text{ of particles } j \text{ have} \varepsilon_{y,j} \le \varepsilon_{y,i}$$
$$\mathbf{nemiz90} = \varepsilon_{z,i} \mid 90\%\text{ of particles } j \text{ have} \varepsilon_{z,j} \le \varepsilon_{z,i}$$

3.10.5 Courant-Snyder parameters

The **CSalpha***, **CSbeta*** and **CSgamma*** routines calculate the Courant-Snyder parameters named α, β and γ. All Courant-Snyder parameters satisfy the equation: $\gamma\beta - \alpha^2 = 1$.

To avoid confusion, the symbols β and γ are always used for the normalized velocity and the Lorentz factor respectively when not specified otherwise. To obtain the Courant-Snyder parameters, first the following quantities are calculated:

$$x_c = x - \bar{x} \quad x'_c = \beta_x - \bar{\beta}_x \quad \varepsilon_x = \sqrt{\overline{x_c^2}\cdot\overline{x'^2_c} - \overline{x_c x'_c}^2}$$
$$y_c = y - \bar{y} \quad y'_c = \beta_y - \bar{\beta}_y \quad \varepsilon_y = \sqrt{\overline{y_c^2}\cdot\overline{y'^2_c} - \overline{y_c y'_c}^2}$$
$$t_c = t - \bar{t} \quad \gamma_c = \gamma - \bar{\gamma} \quad \varepsilon_z = \sqrt{\overline{t_c^2}\cdot\overline{\gamma_c^2} - \overline{t_c\gamma_c}^2}$$

Please note that the emittance values are not normalized. Furthermore, the quantities β_x and β_y are not divided by β_z, similar to the **nemixrms** and **nemiyrms** equations.

The Courant-Snyder parameters are then calculated by:

$$\mathbf{CSalphax} = -\overline{x_c x'_c}/\varepsilon_x \quad \mathbf{CSbetax} = \overline{\beta_z}\cdot\overline{x_c^2}/\varepsilon_x \quad \mathbf{CSgammax} = \overline{x'^2_c}/(\varepsilon_x\cdot\overline{\beta_z})$$

$$\mathbf{CSalphay} = -\overline{y_c y'_c}/\varepsilon_y \quad \mathbf{CSbetay} = \overline{\beta_z}\cdot\overline{y_c^2}/\varepsilon_y \quad \mathbf{CSgammay} = \overline{y'^2_c}/(\varepsilon_y\cdot\overline{\beta_z})$$

$$\mathbf{CSalphaz} = -\overline{t_c\gamma_c}/\varepsilon_z \quad \mathbf{CSbetaz} = \frac{|q_e|}{mc^2}\cdot\frac{\overline{t_c^2}}{\varepsilon_z} \quad \mathbf{CSgammaz} = \frac{mc^2}{|q_e|}\cdot\frac{\overline{\gamma_c^2}}{\varepsilon_z}$$

The average of β_z is a correction factor applied to remain consistent with other codes. The units of **CSalpha** are dimensionless. The units of **CSbetax** and **CSbetay** are [m/rad] and the units of **CSgammax** and **CSgammay** are [rad/m]. The units of **CSbetaz** and **CSgammaz** are [s/eV] and [eV/s] respectively.

3.10.6 Miscellaneous routines

Name	Equation	Description
cputime	cputime	Total CPU time used by the simulation [s].
rmax	$\max\left(\sqrt{x^2+y^2}\right)$	Maximum of radius [m].
numpar	length(x)	Number of particles 'alive'.
Q	$\Sigma N_i q_i$	Total charge [Q].

3.10.7 Scatter statistics

The following functions work in combination with the FISHFILE program, described in section 3.4, to obtain total current and power for each collector plate. Each **nn** must be replaced by 01, 02, …, 10 indicating the plate number. The programs convert total charge and energy to current and power respectively by dividing by the variable **tlen**. This variable must be present in the GPT outputfile, written using **outputvalue**, and must indicate the width of the uniform time distribution of the initial particles. An example is shown in tutorial section 2.11.8.

Name	Equation	Description
scatterInn	$\frac{\Sigma\,\mathbf{scat\_nnQ}}{\mathbf{tlen}}$	Total current on collector plate number **nn**.
ScatterPnn	$\frac{\Sigma\,\mathbf{scat\_nnE}}{\mathbf{tlen}}$	Total power on collector plate number **nn**.

3.10.8 Obsolete functions

The following functions are maintained for backward compatibility only. Please use the given alternatives instead.

Name	Equation	Alternative	Description
nemix	See below	**nemirrms**	Normalized rms emittance in (x,βx) subspace
nemiy	See below	**nemirrms**	Normalized rms emittance for (y,βy) subspace

The **nemix** and **nemiy** routines start by calculating

$$x_c = x - \bar{x} \qquad y_c = y - \bar{y}$$

$$x'_c = \beta_x - \bar{\beta}_x \quad y'_c = \beta_y - \bar{\beta}_y$$

Then the average rotation around the z-axis is subtracted:

$$\bar{\omega}/c = \frac{\Sigma_i\,\beta_{y,i}x_i - \beta_{x,i}y_i}{\Sigma_i\,x_i^2 + y_i^2}$$

$$\beta_{x,i} = \beta_{x,i} + \bar{\omega}y_i/c$$

$$\beta_{y,i} = \beta_{y,i} - \bar{\omega}x_i/c$$

The emittance for the x-x' and y-y' phase spaces are then defined by:

$$\mathbf{nemix} = 4\bar{\gamma}\sqrt{\overline{x_c^2}\cdot\overline{x'^2_c} - \overline{x_c x'_c}^2}$$

$$\mathbf{nemiy} = 4\bar{\gamma}\sqrt{\overline{y_c^2}\cdot\overline{y'^2_c} - \overline{y_c y'_c}^2}$$

It is important to note that the **nemix** and **nemiy** routines always produce a factor 4 higher emittance values. Furthermore, they do not calculate weighted averages.

3.11 GDFSOLVE

gdfsolve [-hHv] –o [outfile] infile.sol program [parameters]
Multidimensional solver for the specified program

-o outfile Write output to a file named **outfile**
-a outfile Append output to a file named **outfile**
-h Give short help and quit.
-H Give long help and quit.
-v Verbose mode. Display iteration output containing all variables and constraints.
infile.sol Name of the file containing information about the parameters that are to be varied and the constraints to be met. See Table 3-D for specific sections in this file.
program [parameters] Command line of a program that creates a GDF file and recognizes the **-o** option, for example GPT.

GDFsolve can be used to polish off a solution, based on a good initial estimate. The primary method used is based on multidimensional Newton-Raphson, providing quadratic convergence in the vicinity of a solution. GDFsolve can also be used as multidimensional optimizer, but this feature is less useful as it is far less efficient and sensitive to local minima and simulation noise.

Table 3-D: Sections in the **.sol** file and a GDFsolve output description

Section	Section	Description
[OPTIONS]	3.11.1	Sets various options and constants directly related to the algorithm.
[VARIABLES]	3.11.2	Defines the variables to be varied.
[CONSTRAINTS]	3.11.3	Defines the constraints to be met.
[OPTIMIZE]	3.11.4	Defines the constraints to be minimized or maximized.
	3.11.5	Output description of GDFsolve

Information about the internal algorithms of GDFsolve can be found in section 1.12. A sample **.sol** file without an optimize section is shown in Listing 3–1.

Listing 3–1: Sample .sol for GDFsolve: Find Isol resulting in an avgr of 1 mm.

```
1.  # Solve average beam radius (avgr) = 1 mm by varying Isol.
2.
3.  [OPTIONS]
4.  backtracking = 0 ;
5.
6.  [VARIABLES]
7.  Isol = 1000 ;              # Define variable to vary, starting value is 1000
8.  Isol.reldelta = 0.01 ;     # Use 1% difference to obtain derivative information
9.  Isol.min = 0 ;             # Isol must be positive
10.
11. [CONSTRAINTS]
12. avgr = 1.0e-3 ;            # avgr must be 1 mm
13. avgr.pos = 1 ;             # at z=1 m
14. avgr.abstol = 0.1e-3 ;     # with a tolerance of +/- 0.1 mm
```

3.11.1 Options

In the [OPTIONS] sections, the following options can be set. The syntax is shown in Table 3-E.

Table 3-E: Options syntax.

Syntax	Required	Default	Description
[OPTIONS]	No		Start of options section.
forcejacobian = 1 ;	No	Depends	Forces a full Jacobian matrix to be calculated after each successful step. When not specified, Broydens' update method is used when the number of variables is less or equal than the number of constraints, see section 1.12.1.5.
SVDtolerance = tol ;	No	0.001	Ratio between the largest and smallest Singular Value. When the ratio is smaller than **tol**, the corresponding direction is dropped.
minconvergence = a ;	No	0.01	Every step must bring the scaled constraints (1-**a**) closer to the solution. When this criterion can not be met, GDFsolve terminates with an error.
backtracking = 0 ;	No	1	Disable the backtracking algorithm and terminate with an error when |**F**| increases.
keepgoing = 1 ;	No	0	Disable all termination criteria and keep running until the executable is forced to stop externally. Using this option is not recommended for inexperienced users.
optimizetolerance = tol ;	No	0.05	The optimizer stops iterating when the target scaled rms value does not change more then **tol**.
ignoreerrors = 1 ;	No	0	Assume all error codes from GPT and GDFA to be a very bad simulation result. Please use with care.

3.11.2 Variables

The [VARIABLES] section defines all variables to be varied. The syntax is shown in Table 3-F.

Table 3-F: Variable syntax.

Syntax	Required	Description
[VARIABLES]	Yes	Start of variables section.
var = x ;	Yes	Defines a new variable named **var**, with initial estimate **x.**
var.absdelta = abs ; **var.reldelta = rel ;**	At least one	The absolute scaling of variable **var** is **abs**. The relative scaling of variable **var** is **rel**. The total scaling is defined as: $dx = \pm\sqrt{\mathbf{abs}^2 + x^2 \cdot \mathbf{rel}^2}$.
var.min = lb ; **var.max = ub ;**	No No	Variable lower bound for variable **var** is **lb**. Variable upper bound for variable **var** is **ub**.

The scaling **delta** is important for the internal algorithms of GDFsolve because the derivative of all constraints is approximated by taking steps larger or equal to **delta**. As a result, a too small **delta** is very sensitive to simulation noise caused by a too low accuracy setting or insufficient particles in the simulation. On the other hand, a too large **delta** could send the derivative approximation into the non-linear regime. Before GDFsolve is used, we recommended first making a scan over a number of **delta**'s and plotting every constraint.

3.11.3 Constraints

The [CONSTRAINTS] section defines all constraints to be met simultaneously. The syntax is shown in Table 3-G.

Table 3-G: Constraints syntax

Syntax	Required	Description
[CONSTRAINTS]	No	Start of constraints section.
con = ft ;	Yes	Define a new constraint named **con** to have target value **ft**.
con.time = t ; **con.position = p ;**	One of these	Specify the time **t** when constraint **con** must be met. Specify the position **p** where constraint **con** must be met.
con.abstol = abs ; **con.reltol = rel ;**	At least one	The absolute tolerance of constraint **con** is **abs**. The relative tolerance of constraint **con** is **rel**. The total tolerance is defined as: $df = \sqrt{\mathbf{abs}^2 + ft^2 \cdot \mathbf{rel}^2}$.

The geometric length of all scaled constraints $|\mathbf{F}|$ is given by:

$$|\mathbf{F}| = \sqrt{\sum \left(\frac{\mathbf{con} - \mathbf{ft}}{df} \right)^2}$$

A solution is found when $|\mathbf{F}| \le 1$. Apart from the termination criterion, the tolerance is also used in the internal algorithms of GDFsolve as the inverse weight of the constraints.

When an identical constraint must be used at different times or locations, the **con** name can be prepended with **group:** notation. For example, the following section makes the average beam radius 3 mm at z=1 m and 2 mm at z=2 m. The tolerance is set to 0.1 mm in both cases.

```
[CONSTRAINTS]
a:avgr = 3.0e-3 ;
a:avgr.position = 1.0 ;
a:avgr.abstol = 0.1e-3 ;

b:avgr = 2.0e-3 ;
b:avgr.position = 2.0 ;
b:avgr.abstol = 0.1e-3 ;
```

3.11.4 Optimize

The [OPTIMIZE] section defines the function(s) to be minimized. The syntax is shown in Table 3-H.

Table 3-H: Optimize syntax.

Syntax	Required	Description
[OPTIMIZE]	No	Start of optimize section.
opt = ft ;	Yes	Define a new function named **opt** with approximate minimum *ft*.
opt.time = t ;	One of	Specify the time **t** when function **opt** must be minimized.
opt.position = p ;	these	Specify the position **p** where constraint **opt** must be minimized.
opt.abstol = abs ; **opt.reltol = rel ;**	At least one	The absolute tolerance of function **opt** is **abs.** The relative tolerance of function **opt** is **rel.** The total tolerance is defined as: $df = \pm\sqrt{\mathbf{abs}^2 + ft^2 \cdot \mathbf{rel}^2}$.
opt.weight = m ;	No	Defines the weight of the constraint. When not specified 1 is assumed. A negative value must be specified to maximize a function.

When more than one function and/or additional constraints are specified, GDFsolve tries to minimize:

$$f = |\mathbf{F}| + \sum \mathbf{m} \frac{\mathbf{opt}}{df} \qquad [3.1]$$

A negative weight parameter can be used to maximize a function.

3.11.5 Output

GDFSolve writes output when all constraints are met and/or when a minimum has been found. Without a solution, there is no output. The outputfile itself contains the settings of all variables and the values of the corresponding constraints, followed by the complete output of the successful GPT run. All variables and constraints can be inspected using GDF2A.

The output of GDFsolve can also be analyzed with GDFA like the output of a normal GPT run. Furthermore, the MR utility can be used in combination with GDFsolve. This solves for identical constraints as function of a scanned parameter. This construction builds an outputfile of all successful GPT runs as function of the scanned parameter. GDFA can be used to analyze these results.

When the **-v** option is specified, GDFsolve writes iteration output to the messages window. It consists of the following columns:

N The iteration number.

varN Value of the N'th variable.

conN Value of the N'th constraint to be solved. This value is followed by the scaled F value, the number of tolerances off the target, between parentheses. When for example **con1** has a value of 10, a target of 4 and a delta of 1, the display will be 10(6) indicating that the constraint is 6 tolerances off its desired value.

optN Value of the N'th function to minimize. This value is followed by the scaled result, the number of tolerances off the estimated minimum times the weight, between parentheses. When for example opt1 has a value of 10, an estimated minimum of 4 and a delta of 1, the display will be 10(6) indicating that the value is expected to drop 6 tolerances.

rms When GDFsolve is used as root finder, the rms value is the vector length of F, the scaled distance from a solution. When GDFsolve is used as optimizer, the scaled results of the functions to minimize are added to this value.

Only the differences in variables and constraints are displayed when the root finder is calculating the Jacobian as indicated by a J following the iteration number.

3.12 GDFTRANS

gdftrans [-hHv] [-s array] [-o outfile] [-a outfile] infile group_by arrays ...
Transpose a GDF file to calculate trajectories as function of any parameter.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode.
-s array	Sort by **array**. Default set to array "**ID**".
-o outfile	Write output to outfile.
-a outfile	Append output to outfile.
infile	Name of the file to perform the calculations on. When GDFTRANS has to read from **stdin**, a dash ('**-**') must be specified explicitly for infile.
group_by	The name of the variable used to group the data.
arrays ...	The name(s) of one or more arrays to convert. Typicaly these are **x y z Bx By Bz** and **G**.

GDFTRANS can be used to obtain particle trajectories by transposing a GDF file. For example to obtain the arrays of the x- and z-coordinates for all individual particles as function of time, the following command can be specified:

```
GDFTRANS -o outfile.gdf infile.gdf time x z
```

By default the arrays are sorted by an additional array named ID, containing the actual numbering. GPT writes the particle number in array ID to allow trajectory plots to be created. An alternative array can be specified using the **-s** option and the array index is used when the ID array is not present.

Apart from the group_by variable, the hierarchy in the GDF File is preserved by GDFTRANS.

3.13 MR

mr [-hHvn] [-o outfile] [-a outfile] mrfile program [options]
Scan one or more parameters and concatenate the output in a single hierarchical file.

-o outfile	Write output to a file named **outfile**.
-a outfile	Append output to a file named **outfile**.
-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Echo the commands to the terminal as they are run.
-n	Display commands but do not actually run them.
mrfile	Name of the file containing information about the parameter(s) that are to be set or varied. It must be an ASCII-file with one or more lines each with the following syntax: **param value** or **param from to step** Lines beginning with "**#**" are ignored.
program [options]	Command line of a program to be run that creates a GDF file.

When the effect of one parameter in a simulation is to be studied, MR can be used to automate this process. The steps needed to use MR are:

- Give the parameter of interest a symbolic name in the inputfile instead of a value. For example the phase of a prebuncher could be named **phi**.
- Create a MR file containing information about the varying parameter. It consists of lines with the name of the varying parameter followed by either a **value** or a **from to step** specification. For example the following columns are equivalent and they can all be used to scan **phi** in the range from 0.4 to 0.7.

phi 0.4 **phi 0.5** **phi 0.6** **phi 0.7**	**phi 0.4 0.7 0.1**	**phi 0.4 0.6 0.1** **phi 0.7**

- Run MR using the syntax specified above. For example to run GPT
  mr -o result.gdf phi.mr gpt inputfile.in
  can be specified if **phi.mr** is one of the files above and **inputfile.in** is a valid simulation inputfile containing **phi**.

MR can be used to efficiently group all parameters of interest into a single file. If, for example, not only **phi** is a parameter of interest, but also **len**, **amp** and **theta**, the following **scan.mr** file sets these additional parameters for varying **phi**.

```
1. # MR file for varying phi
2. phi 0.4 0.7 0.1
3.
4. # Fixed len, amp and theta
5. len 10
6. amp 1
7. theta 5
```

Multidimensional scans are easily created by specifying a range of values for two (or more) parameters. To run the above scan for three different values of **len**, the following **scan.mr** file will start the required 12 runs.

```
1. # MR file for varying phi and len
2. phi 0.4 0.7 0.1
3. len 10
4. len 12
5. len 15
6.
7. # Fixed amp and theta
8. amp 1
9. theta 5
```

In the case of multidimensional scans, the **group_by** parameter of the GDFA program can be used to create plots with any varying parameter along the horizontal (or vertical) axis. For example:

1. **mr -o result.gdf scan.mr gpt infile.in**
2. **gdfa -o avgsasfunctionofphi.gdf result.gdf phi** …
3. **gdfa -o avgsasfunctionoflen.gdf result.gdf len** …

3.14 MPIMR

mpiexec [mpioptions] mpimr [-hHvn] [-o outfile] mrfile program [options]
MPI version of MR that distributes parameter scans over several PC's connected to a network.

The basic functionality of MPIMR is identical to the MR program described in section 3.13. However, where MR runs parameter scans sequentially, MPIMR runs them in parallel on any number of connected PC's or on multiprocessor machines. MPIMR balances the load by initially feeding all slave nodes one job and assigning new jobs only when the previous run is finished. The output of all jobs is stored in temporary files and subsequently concatenated into **outfile** in the correct order.

The use of MPIMR is identical to the use of MR, where **mpiexec** must be used to launch the master and slave processes. Assuming a properly installed MPI environment, the commandline that distributes a GPT parameter scan over ***N*** nodes in the network is:
mpiexec –n *N* mpimr –o result.gdf scan.mr gpt file.in

If one wants to distribute several GPT runs over the three computers named **apple**, **pear** and **berry**, the following: command may be used:
mpiexec –hosts 3 apple 2 pear 1 berry 1 mpimr –o result.gdf scan.mr gpt file.in

The reason for having host **apple** run **2** jobs is that the first process does not run any simulation; it only orchestrates spawning of new processes and it takes care of concatenating the temporary outputfiles. As these tasks consume virtually no CPU time, one usually also wants to run a simulation job on the first machine. Machines with multiple CPUs can run more than one simulation job in parallel, and this can be easily indicated by an (additional) increase in the number of parallel jobs specified after the hostname.

Before you can use MPIMR, the system administrator needs to install a working MPI environment on all machines. MPIMR is compiled and tested with MPICH2 that can be downloaded free of charge from Argonne National Laboratory.

To be able to use MPIMR all executables must be 'known' by all machines. The simplest way to accomplish this is to copy MPIMR.EXE and GPT.EXE into the same directory as the GPT inputfile(s) and the batch file. This implies that GPT.EXE must be copied every time a new custom GPT element is added or modified. If MPIMR is used in combination with GDFSOLVE, than both GDFSOLVE.EXE and GDFA.EXE must be copied to the same directory as the GPT inputfile(s). Analogous to the GPT.EXE case, this implies that GDFA.EXE must be copied every time a custom GDFA program is added or modified.
There are many alternatives to the scheme mentioned above. For example, it is possible to set the PATH environment variable on all machines to include the BIN directory of your working machine. This however requires administrative privileges and could conflict with other GPT users on the remote machines. If used properly, the best alternative is to make clever use of the large number of options of **mpiexec**. However, we can not provide technical support for these options as the implications require too detailed knowledge of the configuration of all machines.

Please note that the Stop button of GPTwin does not stop the running processes, neither on the local nor on remote machines. They all have to be terminated 'by hand' using the Windows Task Manager.

3.15 RAW2GDF

raw2gdf [-hHved] [-o outfile] [-a outfile] infile arr1 fac1 [arr2 fac2] ...
Raw binary file to GDF converter.

-h	Give short help and quit.
-H	Give long help and quit.
-v	Verbose mode. Progress and file statistics are shown.
-e	Convert endian format. This is typically needed when RAW2GDF is run on a non-intel UNIX machine and **infile** was generated on a PC, or vice versa.
-d	Specify when data is given in double-precision numbers.
-o outfile	Write output to **outfile**.
-a outfile	Append output to **outfile**.
infile	Name of the binary file to be converted.
arr1	Name of the array as it is to be specified in the outputfile.
fac1	Multiplication factor for **arr1**.
...	Specify all arrays present in **infile**.with their conversion factor.

RAW2GDF converts a binary file to a GDF file. The resulting GDF file can be read by the field map elements and all GPT utility programs capable of handling GDF files. For example, to test if the resulting file is correct GDF2A can be used.

RAW2GDF writes the columns in the outputfile specified by the **arr** parameters. These column names should be specified in all subsequent handling of the GDF file. To allow conversion to SI units, the arrays are multiplied by their corresponding multiplication factors: **fac**. A **fac** of **1** must be specified if the array is already in SI units. When **fac** is specified to be **0**, the corresponding column is not written in the outputfile.

When a raw data file is generated on a PC, but converted using RAW2GDF on a big-endian (typically a non-intel UNIX) machine, the **-e** option must be specified. This is also the case when a raw data file generated on a big-endian machine must be converted on a PC. Because GDF files are machine independent, a GDF file from any UNIX machine can be used without conversion on a PC and vice versa (if transferred in binary format).

Note: All arrays present in the binary file must be specified on the RAW2GDF commandline, whether they are relevant or not.

4 GPT Reference

This chapter serves as a reference of the simulation variables, keywords and elements. It is not recommended for first reading. The inputfile syntax, operator precedence and predefined constants are provided in section 4.1.

If necessary, you can also write your own element or modify an existing one. How this is done is explained in the GPT Programmer's Reference.

4.1 Inputfile syntax

A GPT inputfile is an ASCII file describing the set up that is to be simulated. This section is a reference for the inputfile language. For information on how to write your own GPT inputfiles, we refer to the tutorial (chapter 2).

A YACC generated parser is used to handle the expressions, if-statements, functions and variables that can be used in the GPT inputfile. YACC uses a specified grammar and operator precedence and generates code to parse the inputfile accordingly. The grammar is chosen to be very close to the standard C expression language. Tables of built-in constants, operator precedence and built-in functions are given at the end of this section.

A GPT inputfile is defined as follows:

inputfile An ASCII file containing a **statementlist**.
All blanks, tabs, new lines and everything following the character "**#**" till the end of the line are ignored.

statementlist One or or more **statement**s.

statement One of the following:
```
variable = value ;
keyword( parameters ) ;
if(condition) statement [else statement]
{ statementlist }
```

variable Sequence of letters and digits, not starting with a digit.
Built-in and previously assigned constants can be set to a new value. This is generally not advisable and a warning is printed.

value A floating point number, a built-in constant or a formula in standard notation. The formula may contain floating point numbers, built-in constants, user-defined variables and functions.

keyword Any of the keywords listed in chapter 3.

parameters One or more **value**s or **string**s separated by commas.

string Anything enclosed by double quotation marks.

Table 4-A: Operators in decreasing order of precedence. Their associativity indicates in what order operators of equal precedence in an expression are applied.

Symbol	Description	Associativity
`()`	Parenthesis (grouping)	Left-to-right
`!`	Factorial	Right-to-left
`+, -`	Unary plus, minus	
`^`	Exponential, right associative	Right-to-left
`*, /`	Multiplication, division	Left-to-right
`+, -`	Addition, subtraction	Left-to-right
`<, <=` `>, >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	Left-to-right
`!=, ==`	Relational is not equal to/is equal to	Left-to-right
`and, &&`	Logical AND	Left-to-right
`or, ||`	Logical OR	Left-to-right
`? :`	Ternary conditional	Right-to-left
`=`	Assignment	Right-to-left

Table 4-B: Built-in constants.

Name	Value	Description
c	299792458	Speed of light
deg	$180/\pi$	Conversion factor from radians to degrees
e	2.7182818284590452353 6	e
eps0	$1/\mu_0 c^2$	Permittivity of vacuum: ε_0
m	Default: **me**	Mass of an elementary particle
ma	$1.66057 \cdot 10^{-27}$	Unified mass constant
me	$9.10953 \cdot 10^{-31}$	Mass of an electron
mp	$1.67265 \cdot 10^{-27}$	Mass of a proton
mu0	$4\pi \cdot 10^{-7}$	Permeability of vacuum: μ_0
pi	3.1415926535897932384 6	π
q	Default: **qe**	Charge of an elementary particle
qe	$-1.6021892 \cdot 10^{-19}$	Charge of an electron

Table 4-C: Built-in functions.

Name	Description
defined(x)	TRUE if **x** is defined, FALSE otherwise
sin(x)	sine of **x**
cos(x)	cosine of **x**
tan(x)	tangent of **x**
asin(x)	$\sin^{-1}(\mathbf{x})$ in range $[-\pi/2, \pi/2]$, $\mathbf{x} \in [-1,1]$
acos(x)	$\cos^{-1}(\mathbf{x})$ in range $[0, \pi]$, $\mathbf{x} \in [-1, 1]$
atan(x)	$\tan^{-1}(\mathbf{x})$ in range $[-\pi/2, \pi/2]$
sinh(x)	hyperbolic sine of **x**
cosh(x)	hyperbolic cosine of **x**
tanh(x)	hyperbolic tangent of **x**
log(x)	natural logarithm $\ln(\mathbf{x})$, $\mathbf{x}>0$
log10(x)	base 10 logarithm $\log_{10}(\mathbf{x})$, $\mathbf{x}>0$
exp(x)	exponential function $e^{\mathbf{x}}$
sqrt(x)	$\sqrt{\mathbf{x}}$, $\mathbf{x} \geq 0$
abs(x), |x|	$\lvert\mathbf{x}\rvert$
ceil(x)	Smallest integer1F[2] not less than **x**
floor(x)	Largest integer[2] not greater than **x**
round(x)	Integer nearest to **x**

[2] Please note that **ceil(-pi)=-3** and **floor(-pi)=-4**

4.2 Accelerating structures

This section contains the build-in accelerating structures with time-varying electromagnetic fields.

4.2.1 TErectcavity

`TErectcavity(ECS,a,b,d,m,n,p,const,phi);`
Rectangular TE_{mnp} mode resonant cavity.

`ECS`	Element coordinate system.
`a`	Total cavity length in x-direction.
`b`	Total cavity length in y-direction.
`d`	Total cavity length in z-direction.
`m`	Mode index in x-direction.
`n`	Mode index in y-direction.
`p`	Mode index in z-direction.
`Bo`	Multiplication factor for all fields.
`phi`	Phase factor: φ in radians.

The fields generated by this element are given by [5F6]:

$$\mathbf{B} = B_0 \begin{pmatrix} -\frac{1}{k_c^2}\frac{p\pi}{d}\frac{m\pi}{a}\sin\left(\frac{m\pi x}{a}\right)\cos\left(\frac{n\pi y}{b}\right)\cos\left(\frac{p\pi z}{d}\right) \\ -\frac{1}{k_c^2}\frac{p\pi}{d}\frac{n\pi}{b}\cos\left(\frac{m\pi x}{a}\right)\sin\left(\frac{n\pi y}{b}\right)\cos\left(\frac{p\pi z}{d}\right) \\ \cos\left(\frac{m\pi x}{a}\right)\cos\left(\frac{n\pi y}{b}\right)\sin\left(\frac{p\pi z}{d}\right) \end{pmatrix} \sin(\omega t+\varphi)$$

$$\mathbf{E} = B_0 \frac{\omega}{k_c^2} \begin{pmatrix} \frac{n\pi}{b}\cos\left(\frac{m\pi x}{a}\right)\sin\left(\frac{n\pi y}{b}\right)\sin\left(\frac{p\pi z}{d}\right) \\ -\frac{m\pi}{a}\sin\left(\frac{m\pi x}{a}\right)\cos\left(\frac{n\pi y}{b}\right)\sin\left(\frac{p\pi z}{d}\right) \\ 0 \end{pmatrix} \cos(\omega t+\varphi)$$

where

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2$$

$$\beta = \left[\left(\frac{2\pi}{\lambda}\right)^2 - k_c^2\right]^{1/2} = \frac{p\pi}{d}$$

$$k = \frac{2\pi}{\lambda} = \left[\left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2 + \left(\frac{p\pi}{d}\right)^2\right]^{1/2} = \frac{\omega}{c}$$

4.2.2 TM010cylcavity

`TM010cylcavity(ECS,d,R,const,phi) ;`
Cylindrically symmetric pillbox cavity in TM_{010} mode.

`ECS`	Element coordinate system.
`d`	Total cavity length in z direction.
`R`	Radius of the cavity.

A	Multiplication factor for all fields.
phi	Phase factor: φ in radians.

The fields are calculated by [6]:

$$E_z = \mathbf{A}\ J_0\left(p01\frac{r}{\mathbf{R}}\right)\sin(\omega_{010}\cdot t+\varphi)$$

$$B_\varphi = \mathbf{A}\ J_1\left(p01\frac{r}{\mathbf{R}}\right)\cos(\omega_{010}\cdot t+\varphi)/c$$

where

$$p01 = 2.4048$$

$$\omega_{010} = kc = \frac{p01}{\mathbf{R}}\,c$$

and c is the speed of light. The resonant wavelength is given by:

$$\lambda = \frac{2\pi}{k} = 2.61\mathbf{R}$$

4.2.3 TM110cylcavity

TM110cylcavity(ECS,d,R,ffac,phi) ;
Cylindrically symmetric pillbox cavity in TM_{110} mode.

ECS	Element Coordinate System.
d	Thickness of the cavity. $-d/2<z<d/2$.
R	Outer radius of the cavity.
ffac	Multiplcations factor for all fields.
phi	Phase factor in radians.

The electromagnetic fields of this element are in cylindrical coordinates given by:

$$E_z = J_1\left(\frac{x_{110}\,r}{\mathbf{R}}\right)\cos(\varphi)\sin(\omega_{110}\,t+\mathbf{phi})$$

$$B_r = -\frac{J_1\left(\frac{x_{110}\,r}{\mathbf{R}}\right)}{\omega_{110}\,r}\sin(\varphi)\cos(\omega_{110}\,t+\mathbf{phi})$$

$$B_\varphi = -\frac{x_{110}\,J_0\left(\frac{x_{110}\,r}{\mathbf{R}}\right)/\mathbf{R} - J_1\left(\frac{x_{110}\,r}{\mathbf{R}}\right)/r}{\omega_{110}}\cos(\varphi)\sin(\omega_{110}\,t+\mathbf{phi})$$

where $x_{110}\approx 3.8317$ is the first zero of J_1 and $\omega_{110}=c\cdot x_{110}/\mathbf{R}$ with c the speed of light. To avoid the singularities at $r=0$, the following Taylor expansion of the fields is used for all $r<10^{-6}$:

$$E_z = \frac{x'}{2}\sin(\omega_{110}\,t+\mathbf{phi})$$

$$B_x = -\frac{x'\,y'}{8\,c}\cos(\omega_{110}\,t+\mathbf{phi})$$

$$B_y = -\frac{x'^2\,(y'^2-12)-4(y'^2-8)}{64\,c}\cos(\omega_{110}\,t+\mathbf{phi})$$

where $x'=x_{110}\,x/\mathbf{R}$ and $y'=x_{110}\,y/\mathbf{R}$ are the dimensionless position variables.

Example: Idealized cylindrically symmetric 1 mT, 5 cm, 3 Ghz TM_{110} cavity at z=10 cm.

```
1. freq=2998e6 ;                 # European 3 GHz
2. x110=3.8317 ;                 # First zero of J1
3. R   =c*x110 / (2*pi*freq) ;   # Calculate Radius based on frequency
4. Bmax=1e-3 ;                   # Maximum field: 1 mT
5. TM110cylcavity("wcs", "z",0.1, 0.05,R,2*c*Bmax,phi) ;
```

4.2.4 TMrectcavity

TMrectcavity(ECS,a,b,d,m,n,p,const,phi);
Rectangular TM_{mnp} mode resonant cavity.

ECS	Element coordinate system.

a	Total cavity length in x-direction.
b	Total cavity length in y-direction.
d	Total cavity length in z-direction.
m	Mode index in x-direction.
n	Mode index in y-direction.
p	Mode index in z-direction.
Eo	Multiplication factor for all fields.
phi	Phase factor: φ in radians.

The fields generated by this element are given by [6]:

$$\mathbf{E} = E_0 \begin{pmatrix} -\frac{1}{k_c^2}\frac{p\pi}{d}\frac{m\pi}{a}\cos\left(\frac{m\pi x}{a}\right)\sin\left(\frac{n\pi y}{b}\right)\sin\left(\frac{p\pi z}{d}\right) \\ -\frac{1}{k_c^2}\frac{p\pi}{d}\frac{n\pi}{b}\sin\left(\frac{m\pi x}{a}\right)\cos\left(\frac{n\pi y}{b}\right)\sin\left(\frac{p\pi z}{d}\right) \\ \sin\left(\frac{m\pi x}{a}\right)\sin\left(\frac{n\pi y}{b}\right)\cos\left(\frac{p\pi z}{d}\right) \end{pmatrix} \sin(\omega t+\varphi)$$

$$\mathbf{B} = E_0 \frac{\omega}{k_c^2 c^2} \begin{pmatrix} \frac{n\pi}{b}\sin\left(\frac{m\pi x}{a}\right)\cos\left(\frac{n\pi y}{b}\right)\cos\left(\frac{p\pi z}{d}\right) \\ -\frac{m\pi}{a}\cos\left(\frac{m\pi x}{a}\right)\sin\left(\frac{n\pi y}{b}\right)\cos\left(\frac{p\pi z}{d}\right) \\ 0 \end{pmatrix} \cos(\omega t+\varphi)$$

where

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2$$

$$\beta = \left[\left(\frac{2\pi}{\lambda}\right)^2 - k_c^2\right]^{1/2} = \frac{p\pi}{d}$$

$$k = \frac{2\pi}{\lambda} = \left[\left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2 + \left(\frac{p\pi}{d}\right)^2\right]^{1/2} = \frac{\omega}{c}$$

4.2.5 Trwcell

trwcell(ECS,Ezef,phi,w,L) ;

Resonant cavity of a $\frac{2}{3}\pi$ traveling wave buncher or linac.

ECS	Element Coordinate System.
Ezef	Effective **E** field [V/m].
phi	Phase factor: ϕ in radians.
w	Angular frequency: ω.in [s^{-1}].
L	Cell length [m].

The fields generated by this element are given in cylindrical coordinates by:

$$E_z = M\sin(\omega t+\phi-k_z z)\mathrm{I}_0(k_t r)$$

$$E_r = M\cos(\omega t+\phi-k_z z)\mathrm{I}_1(k_t r)\frac{k_z}{k_t}$$

$$B_\phi = E_r/c$$

where the following definitions are used:

$$k_z = \tfrac{2}{3}\pi/\mathrm{L}$$

$$k_t = \sqrt{k_z^2-\omega^2/c^2}$$

$$M = \tfrac{3\sqrt{3}}{2\pi}\,\mathrm{Ezef}$$

The factor in front of **Ezef** is the average of a sine in range $\frac{1}{6}\pi - \frac{5}{6}\pi$. It converts peak field to average field and is introduced for compatibility.

The formulas are proposed by G. Saxon [6F7].

4.2.6 Trwlinac

`trwlinac(ECS,ao,Rs,Po,P,Go,thetao,phi,w,L) ;`

Constant gradient $\frac{2}{3}\pi$ traveling wave buncher or linac without beamloading.

`ECS`	Element Coordinate System.
`ao`	Initial attenuation constant: α_0.
`Rs`	Shunt impedance: R_s.
`Po`	Input power, design value: P_0 [W].
`P`	Input power, actual power used: P [W].
`Go`	Design gamma at entrance: γ_0.
`thetao`	Bunch phase with respect to wave, design value: θ_0.
`phi`	RF phase offset: φ.
`w`	Angular frequency: ω.in [s^{-1}].
`L`	Length of the structure: L [m].

An rf–wave with constant amplitude travels through the structure. The phase velocity of the wave increases along the structure in such a way that an electron entering the linac with initial energy $\gamma_0 mc^2$ is accelerated but remains in the same position relative to the wave. For this to take place, the linac needs to be operated at its design power Po and the particle must be input at the design phase θ_0 with respect to the 'crest' of the wave. It is possible however to operate the linac at a different power than the design power and the particles can enter at any phase.

According to the design values the amplitude and phase of the field that accelerates the particles is:

$$E_0 = \sqrt{2\alpha_0 R_s P_0}\cos(\theta_0)$$

$$\varphi = \omega t - \int_0^z k_z(z')dz' + \theta_0$$

where z is the longitudinal position of the particle relative to the beginning of the structure.

To calculate the integral, we first calculate the particle velocity using the linear increase in energy from $\gamma_0 mc^2$ to $\gamma_0 mc^2 + E_0 q_e z$:

$$\beta(z)^2 = 1 - \frac{1}{(\gamma_0 + Fz)^2}$$

where F is the normalized design acceleration:

$$F = -\frac{E_0 q_e}{m_e c^2}$$

The longitudinal and transverse wavenumbers are given by:

$$k_z = \frac{k_0}{\beta(z)},\ k_t = \sqrt{k_z^{\,2} - k_0^{\,2}}$$

where $k_0 = \frac{\omega}{c}$

Yielding:

$$\int_0^z k_z(z')dz' = \frac{k_0}{F}\left[\sqrt{(Fz+\gamma_0)^2 - 1} - \sqrt{\gamma_0^{\,2} - 1}\right]$$

The amplitude and phase of the actual accelerating field are due to the rf input power P and the rf phase φ. They are given by:

$$E_1 = \sqrt{2\alpha_0 R_s P}$$

$$\theta = \omega t - \int_0^z k_z(z')dz' + \varphi$$

P may differ from Po and φ does not need to be chosen such that the particles enter the linac at phase θ_0.

The fields produced in cylindrical coordinates are:

$$E_z = E_1 \sin(\theta) I_0(k_t r)$$
$$E_r = E_1 \cos(\theta) I_1(k_t r) \frac{k_z}{k_t}$$
$$B_\varphi = \frac{E_r}{c}$$

This element is shaped using an erf function at the entrance and exit of the structure. The characteristic length is 1 mm. The shaping does not affect the numerical results but improves the simulation speed considerably because it reduces the sudden turn on and off of the fields.

4.2.7 Trwlinbm

`trwlinbm(ECS,ao,Rs,Po,P,Ib,Go,thetao,phi,w,L) ;`

Constant gradient $\frac{2}{3}\pi$ traveling wave buncher or linac with selfconsistent beamloading.

`ECS`	Element Coordinate System.
`ao`	Initial attenuation constant: α_0.
`Rs`	Shunt impedance: R_s.
`Po`	Input power, design value: P_0 [W].
`P`	Input power, actual power used: P [W].
`Ib`	Beam current: I_b.
`Go`	Design gamma at entrance: γ_0.
`thetao`	Bunch phase with respect to wave, design value: θ_0.
`phi`	RF phase offset: φ.
`w`	Angular frequency: ω.in [s^{-1}].
`L`	Length of the structure: L [m].

The electromagnetic fields generated by this element are separated into two independent waves:

- The accelerating rf-wave.
- The beam induced beamloading wave.

The accelerating wave is calculated in precisely the same way as in trwlinac, described in section 4.2.6. The beamloading wave has the same structure, i.e. the same wavenumbers, but different amplitude and phase. When the linac is run at its design values, the beamloading field opposes the accelerating field and the field's amplitude increases along the structure. When the linac is used far off its nominal settings, analytical expressions or correct approximations for the beamloading wave can not be derived. To allow this model to be used for all input settings, differential equations for the beamloading amplitude and phase are solved while tracing the particles.

The accelerating rf-wave has amplitude E_1 and φ as defined in section 4.2.6. The beamloading wave has amplitude E_b and phase $\varphi+\varphi_b$. Because the structure of both waves is similar, the total fields are:

$$E_z = \left(E_1 \sin(\theta) + E_b \sin(\theta+\theta_b)\right) I_0(k_t r)$$
$$E_r = \left(E_1 \cos(\theta) + E_b \cos(\theta+\theta_b)\right) I_1(k_t r) \frac{k_z}{k_t}$$
$$B_\varphi = \frac{E_r}{c}$$

Both the amplitude and phase of the beamloading wave are calculated from the particle trajectories. However, an amplitude and phase representation is not very efficient computationally. Therefore the beamloading wave is internally characterized by the amplitudes *u* and *v* of two independent waves, 90° out of phase.

$$\begin{cases} u = A\cos(\varphi_b) \\ v = A\sin(\varphi_b) \end{cases} \Rightarrow \begin{cases} A\sin(\theta+\varphi_b) \rightarrow u\sin(\theta) + v\cos(\theta) \\ A\cos(\theta+\varphi_b) \rightarrow u\cos(\theta) - v\sin(\theta) \end{cases}$$

The differential equations for *u* and *v* to be solved are then given by:

$$\frac{du}{dt} = \frac{du}{dz}\frac{dz}{dt} = \frac{\alpha_0 I_b R_s}{1-2\alpha_0 z}\frac{v_z}{N}\sin(\theta)$$
$$\frac{dv}{dt} = \frac{dv}{dz}\frac{dz}{dt} = \frac{\alpha_0 I_b R_s}{1-2\alpha_0 z}\frac{v_z}{N}\cos(\theta)$$

with boundary condition

$$u_{t=0} = v_{t=0} = 0$$

The *u* and *v* representation is transformed back using:

$$E_b = \sqrt{u^2 + v^2}$$
$$\varphi_b = \arctan(\frac{v}{u})$$

The amplitude and the phase of the beamloading wave are written in the outputfile as **`Eb`** and **`phib`**.

4.3 Timestep and output control

This section contains the elements controlling the Runge-Kutta solver and output generation. Using **accuracy**, the calculation accuracy can be specified. The **tout** and **screen** keywords specify the time and position when output is written. The other elements can be used to have more control over the timestep mechanism.

4.3.1 Acceptance

acceptance(ECS) ;
Write acceptance information to the GPT outputfile.

ECS All output is written relative to the Element Coordinate System specified here.

When beam-pipes and other obstacles are introduced in a GPT simulation not all particles necessarily make it through. To investigate which part of the initial phase space is (not) removed downstream the **acceptance** keyword can be used. Furthermore, this element can be very helpful in debugging GPT inputfiles when particles are lost by an unknown element. The following arrays are written to the GPT outputfile.

Particles that make it through:
OK_x_start *x*-position [m] where the particle is started
OK_y_start *y*-position [m] where the particle is started
OK_z_start *z*-position [m] where the particle is started
OK_Bx_start *x*-velocity/c when the particle is started
OK_By_start *y*-velocity/c when the particle is started
OK_Bz_start *z*-velocity/c when the particle is started
OK_G_start Lorentz factor when the particle is started
OK_t_start Simulation time [s] when the particle is started

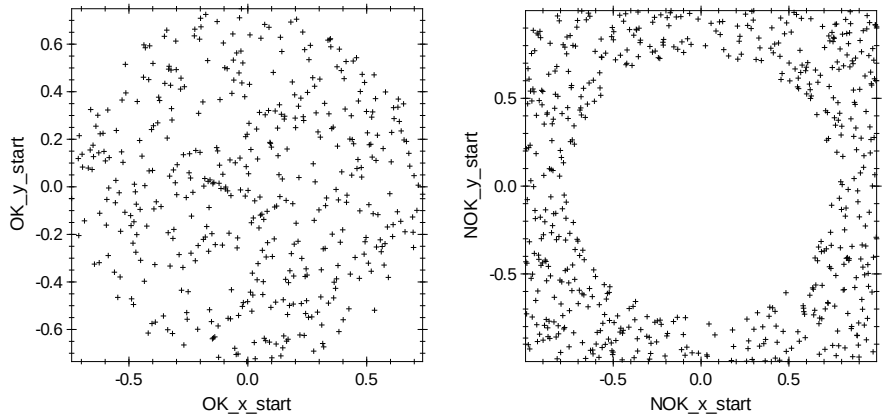
Particles that are removed from the simulation:
NOK_x_start *x*-position [m] where the particle is started
NOK_y_start *y*-position [m] where the particle is started
NOK_z_start *z*-position [m] where the particle is started
NOK_Bx_start *x*-velocity/c when the particle is started
NOK_By_start *y*-velocity/c when the particle is started
NOK_Bz_start *z*-velocity/c when the particle is started
NOK_G_start Lorentz factor when the particle is started
NOK_t_start Simulation time [s] when the particle is started

NOK_x_end *x*-position [m] where the particle is removed

NOK_y_end	*y*-position [m] where the particle is removed
NOK_z_end	*z*-position [m] where the particle is removed
NOK_Bx_end	*x*-velocity/c when the particle is removed
NOK_By_end	*y*-velocity/c when the particle is removed
NOK_Bz_end	*z*-velocity/c when the particle is removed
NOK_G_end	Lorentz factor when the particle is removed
NOK_t_end	Simulation time [s] when the particle is removed

A simple example is shown below: A rectangular beam is sent into a pipe.

```
1.  # Start 1000 particles, square distribution in xy-space
2.  setparticles("beam",1000,me,qe,0.0) ;
3.  setxdist("beam","u",0,2) ;
4.  setydist("beam","u",0,2) ;
5.  setGdist("beam","u",200,0) ;
6.
7.  # Beam pipe with R=0.75 m, between z=1 and z=2 m
8.  rmax("wcs","z",1.5, 0.75,1 ) ;
9.
10. # Acceptance output
11. acceptance("wcs","I") ;
12. tcontinue(4/c) ;
```



The accepted (left) and not accepted (right) part of the initial beam.

4.3.2 Accuracy

accuracy(GBacc,[xacc]) ;
Specifies the calculation accuracy.

GBacc	Negative base 10 logarithm of the simulation accuracy for γβ. If not specified, 4 is assumed.
xacc	Negative base 10 logarithm for the simulation accuracy for position. If not specified, 6 is assumed. It is rarely needed to change this value. Increasing **GBacc** usually increases the accuracy for the position simultaneously.

The simulation accuracy is calculated as follows:

$$\gamma\beta_{\max} = 10^{-\mathbf{GBacc}}$$
$$x_{\max} = 10^{-\mathbf{xacc}}$$

The requested accuracy is the minimal accuracy per step. For very long runs with over 10^4 steps, the final accuracy should be checked by increasing the accuracy and verifying that the result does not change significantly.

For more information about accuracy and timesteps see section 1.8.

4.3.3 dtmax

dtmax=expression ;
Specifies the maximum timestep size.

expression Maximum timestep. If the automatic timestep mechanism suggests a timestep larger than **dtmax**, **dtmax** is used. **dtmax** can be used to avoid particles 'jumping over' small global elements. Default ∞.

4.3.4 dtmaxt

dtmaxt(tstart,tend,dtmax) ;
Enforce a maximum timestep within a specified interval.

tstart Start time.
tend End time.
dtmax Maximum timestep when **tstart**<*t*<**tend**

Forces the timesteps to remain less than **dtmax** when the simulation time is within **tstart** and **tend**. This element should only be used when the adaptive stepsize control mechanism doesn't perform satisfactory.

Example: 100 timesteps are enforced between 85 and 95 ns.

```
1. startpar( "wcs","I", 0,0,0, 0,0,0.5 ) ;
2. tout( 0, 100e-9, 20e-9 ) ;
3. dtmaxt( 85e-9, 95e-9, 0.1e-9 ) ;
```

4.3.5 dtmin

dtmin=expression ;
Specifies the minimum timestep size.

expression Minimum timestep. If the automatic timestep mechanism suggests a timestep smaller than **dtmin**, **dtmin** is used. Please note that in this case the required accuracy cannot be guaranteed. Before using **dtmin** consider lowering your **accuracy**. Default 0.

4.3.6 dtstart

dtstart=expression ;
Specifies the initial timestep.

expression This is the initial value for *h* as described in section 1.8. Subsequent values for *h* are calculated automatically. Default: Twice the separation between the initial two particles, or $\frac{1}{1000c}$ if all particles are started simultaneously.

4.3.7 Outputvalue

outputvalue(name,expr) ;
Writes a value in the outputfile.

name Name of the value as it will be written in the outputfile.
expr Value to be written in the outputfile. Any expression can be entered.

Example:

```
1. w = 240e6 ;
2. outputvalue("freq", 2*pi*w) ;
```

4.3.8 Pp

pp(param1,param2,...) ;
Print parameters for diagnostic purposes.

This line instructs GPT to print the values of the parameters to **stderr**. Checking the results of intermediate calculations in an inputfile is a useful debug aid.

4.3.9 safety

safety=expression ;
Specifies the safety parameter.

expression The safety parameter *S* as described in section 1.8. Default 0.85.

Reducing the safety parameter can save a small amount of CPU time when tracking particle trajectories through fast changing fields.

4.3.10 Screen

```
screen(ECS,at,[CCSname]) ;
screen(ECS,from,to,step,[CCSname]) ;
```

Writes particle coordinates when the particles pass through a nondestructive screen to the outputfile.

ECS	Element Coordinate System of the screen. Using the ECS, the screen(s) can be located anywhere in space with any orientation.
at	The screen is positioned at z=**at** with respect to the specified **ECS**.
from,to,step	The screens are positioned at z=**from**+N**step** if **from**$\leq t \leq$**to**, where $N \in [0,1,2,...]$.
CCSname	The name of the CCS relative to which the coordinates of the particles will be written. If not specified, the screens ECS is used.

The typical use to position a screen at for example the z=5 meter plane is:

```
screen("wcs", "I", 5) ;
```

For every screen a separate group in the GPT outputfile, named **position** with value **at** is created. The coordinates written in the outputfile are listed in Table 4-D. When positioning multiple screens in combination with GDFA, you can group by **position** to create useful plots. Particles in rings can 'hit' the same screen several times. In that case multiple recordings of the same particle, with different t, will be present in the outputfile.

Table 4-D: Coordinates in the **position** group written by the **screen** element.

ID	Particle identification numbers.
x,**y**,**z**	Positions of the particles.
rxy	Particle distances from the z-axis.
Bx,**By**,**Bz**	Velocities of the particles normalized to the speed of light.
G	Lorentz factors of the particles.
t	Time when the particles pass the screen.

The equation for the screen is:

$$\mathbf{M}_{EW}^{-1}\left(\mathbf{r}_{WCS}-\mathbf{o}\right)\cdot\hat{\mathbf{z}}=0$$

where

$$\mathbf{o}=\mathbf{o}_{EW}+\mathbf{at}\mathbf{M}_{CW}\cdot\hat{\mathbf{z}}$$

For information about the used notation and coordinate systems in general, see section 1.4.

The coordinates are calculated via 'perfect' interpolation between two consecutive timesteps, where the interpolation error is always below or equal to the tracking error. Screens need to buffer particle coordinates and involve complex interpolations. They are slower and consume more resources compared to snapshots.

4.3.11 Snapshot

```
snapshot(time,[CCSname]) ;
snapshot(from,to,step,[CCSname]) ;
```

Writes the coordinates of the particles at the given simulation time to the outputfile.

time	Simulation time when the coordinates of the particles are to be written in the outputfile.
from,to,step	The coordinates of the particles are to be written in the outputfile at t=**from**+N**step** if **from** $\leq t \leq$ **to**, where $N \in [0,1,2,...]$.
CCSname	The name of the CCS relative to which the coordinates of the particles are to be written. If not specified, WCS is assumed.

The coordinates of the particles are written in the outputfile when the simulation time is the specified **time**. When **from to step** is specified, output is written periodically.

For every time output a separate group in the outputfile, named **time** with value **time**, is created containing the following coordinates:

Table 4-E: Coordinates in the **time** group written by the **tout** element.

ID	Particle identification numbers.
x,**y**,**z**	Positions of the particles.
rxy	Particle distances from the z-axis.
Bx,**By**,**Bz**	Velocities of the particles normalized to the speed of light.
G	Lorentz factors of the particles.

The automatic timestep algorithm is not affected by the snapshots. This allows the ODE solver to take the largest possible step, given the current accuracy setting, making snapshots the fastest output mechanism. Please see the **tout** keyword when the electromagnetic fiels output is required.

4.3.12 Tcontinue

tcontinue(t) ;
Continue the simulation till the specified time

t Minimum simulation time [s].

Normally, GPT stops execution after the last **tout** or **screen**. To overrule this behavior, the **tcontinue** element forces GPT to continue until the specified time **t**. This element is useful when solving additional differential equations with custom output routines. When scattering statistics are to be obtained it is typically used with all **tout** statements removed.

Stopping GPT takes precedence over **tcontinue**. In other words, when an element actively signals to stop GPT, the simulation will always be stopped.

4.3.13 time

time=expression ;
Specifies the simulation starting time.

expression Starting time of the simulation. Default: Release time of first particles to be started.

Usually used to continue a simulation using a different inputfile. See **setfile** for more documentation.

4.3.14 tmax

tmax=expression ;
Forces the simulation to stop at the specified simulation time.

expression Maximum simulation time [s]. Default: ∞.

A GPT simulation typically continues to run until all particles pass all screens. However, when particles go back due to for example a decelerating rf-phase, they never hit the last screen. This potentially results in a simulation that continues to run forever. Ideally this situation is prevented by removing all backstreaming particles with for example **zminmax** and **rmax**. A very effective alternative is setting a maximum simulation time with **tmax**.

Please note that **tmax** does not set the maximum CPU time.

4.3.15 Tout

tout(time,[CCSname]) ;
tout(from,to,step,[CCSname]) ;
Writes the electromagnetic fields at the position of all particle coordinates at the given simulation time to the GPT outputfile.

time	Simulation time when the coordinates of the particles are to be written in the outputfile.
from,to,step	The coordinates of the particles are to be written in the outputfile at t=**from**+N**step** if **from** $\le t \le$ **to**, where $N \in$[0,1,2,...].
CCSname	The name of the CCS relative to which the coordinates of the particles are to be written. If not specified, WCS is assumed.

The coordinates of the particles are written in the outputfile when the simulation time is the specified **time**. When **from to step** is specified, output is written periodically.

For every time output a separate group in the outputfile, named **time** with value **time**, is created containing the following coordinates:

Table 4-F: Coordinates in the **time** group written by the **tout** element.

ID	Particle identification numbers.
x,**y**,**z**	Positions of the particles.
rxy	Particle distances from the *z*-axis.
Bx,**By**,**Bz**	Velocities of the particles normalized to the speed of light.
G	Lorentz factors of the particles.
fEx,**fEy**,**fEz**	Electric fields at particle positions.
fBx,**fBy**,**fBz**	Magnetic fields at particle positions.

To obtain field information, the automatic timestep algorithm generates its timesteps in such a way that the particle coordinates are calculated at exactly the requested **time**. This slows down the ODE solver. If electromagnetic field information is not required, please use the snapshot keyword.

4.4 Initial particle distribution

The following elements can be used to specify the initial particle distribution. More information can be found in the corresponding tutorial sections 2.9 and 2.10.

4.4.1 Set manipulation

The following keywords perform an action on an entire particle set.

4.4.1.1 SetParticles

```
setparticles(set,N,m,q,Qtot);
```

Create a particle set with a specified number of particles.

`set`	Name of particle set to create.
`N`	Number of particles in this set.
`m`	Mass of elementary particle in this set [kg].
`q`	Charge of elementary particle in this set [C].
`Qtot`	Total charge of all particles in this set [C].

The total charge `Qtot` is uniformly distributed among the particles: The number of elementary particles every macro-particle represents is given by:

$$n = \frac{\mathbf{Qtot}}{\mathbf{Nq}}$$

One of the space-charge elements as described in section 4.5 must be used to include space-charge calculations. Otherwise, the **Qtot** parameter is ignored.

4.4.1.2 SetFile

setfile(set,filename) ;

Read a complete particle distribution from a GDF file.

set	Name of particle set to generate.
filename	Name of the GDF file containing the particle coordinates.

This element reads a macro-particle distribution from **filename** into the specified particle **set**. All particles are set relative to 'wcs'. The file can be created using **asci2gdf** or an outputfile of a previous GPT run can be specified. The GDF File must contain at least the following arrays:

x	Particle *x*-coordinates [m]
y	Particle *y*-coordinates [m]
z	Particle *z*-coordinates [m]
GBx	Particle normalized *x*-momentum: p_x/mc.
GBy	Particle normalized *y*-momentum: p_y/mc.
GBz	Particle normalized *z*-momentum: p_z/mc.

Instead of the momentum coordinates, it is also possible to specify the velocity components:

Bx	Particle normalized *x*-velocity: v_x/c.
By	Particle normalized *y*-velocity: v_y/c.
Bz	Particle normalized *z*-velocity: v_z/c.

Optionally, the following arrays can be present:

t	Release time of the particle [s]. Use this column to set the initial temporal distribution. If this column is not present, all particles start at t=0.
G	Particle Lorentz factor. If not specified, the Lorentz factor is calculated from: $\mathbf{G} = 1/\sqrt{1-\left(\mathbf{Bx}^2+\mathbf{By}^2+\mathbf{Bz}^2\right)}$. Explicitly specifying **G** slightly improves the numerical accuracy of the calculations, but this effect is only noticeable when **G** $> 10^6$.
m	Mass of the elementary particles this macro-particle represents. Default: See below.
q	Charge of the elementary particles this macro-particle represents. Default: See below.
nmacro	Number of elementary particles this macro-particle represents. Default: 0.
rmacro	Typical radius of this macro-particle. See the space-charge elements for a detailed description of this parameter. Default: 0.

The mass and charge of the elementary particles can be specified as separate arrays. When these are not present, the user-defined variables **m** and **q** are read from the inputfile. When also these are not specified, the mass and charge of an electron are used. To start protons, without specifying their mass and charge in **m** and **q** columns, the following lines could be used:

```
1. m = mp ;
2. q = -qe ;
3. setfile("protons","file.gdf") ;
```

If the file is a GPT outputfile created with a **tout(t);** statement, this group must be specified by a **time=t;** statement above **setfile**. Outputfiles created by MR can also be used. Either use MR again with exactly the same parameter sweep or specify **var=value** above **setfile** to specify which group has to be used. Listing 4–3 and Listing 4–2 show an example how to start particles from a previous GPT run.

Listing 4–1: **file1.in**: First inputfile writing output at t=1 ns to **file1.gdf**

```
1. setparticles("beam",1000,10*me,10*qe,Qtot) ;
2. setrxydist("beam","u",1,2) ;
3. setphidist("beam","u",0,2*pi) ;
4. ...
5. tout(1e-9) ;
```

Listing 4–2: **file2.in**: Second inputfile, reading the particle coordinates from **file1.gdf**

```
1. time=1e-9 ;
2. setfile( "beam", "file1.gdf" ) ;
3. ...
4. tout(2e-9) ;
```

4.4.1.3 SetScale

setscale(set,x,y,z,GBx,GBy,GBz,[t,[nmacro,rmacro]] ;
Multiplies all initial particle coordinates by a specified constant

set	Particle set
x	Scale factor for the *x*-coordinates
y	Scale factor for the *y*-coordinates
z	Scale factor for the *z*-coordinates
GBx	Scale factor for the $\gamma\beta_x$ -coordinates
GBy	Scale factor for the $\gamma\beta_y$ -coordinates
GBz	Scale factor for the $\gamma\beta_z$ -coordinates
t	Optional scale factor for the release times
nmacro	Optional scale factor for the number of elementary particles
rmacro	Optional scale factor for the macro particle radius

This element multiplies all initial particle coordinates in a set by a specified constant. It is particularly useful to scale the initial coordinates read from an external file with **setfile**.

Example: If, for example, the file **particles.gdf** has *x*, *y* and *z*-coordinates in [mm], they can be converted to [m] as follows:

```
1. mm=1e-3 ;
2. setfile("beam","particles.gdf") ;
3. setscale("beam",mm,mm,mm,1,1,1) ;
```

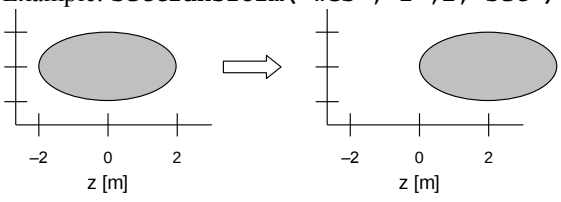
4.4.1.4 SetTransform

settransform(ecs,set);
Apply a coordinate transform on the particles in a set.

ECS	Transform specifiction.
set	Name of particle set.

The particle coordinates are modified according to the specified **ECS**. When the **ECS** specification transform does not start with **"wcs",** all particles are moved to the specified **CCS**. In that case only local and global elements specified relative to the same **CCS** will affect the particle trajectories.

Example: **settransform("wcs","z",2,"set") ;**



4.4.1.5 SetMove

setmove(setfrom,setto);
Move all particles in a set to a different set.

setfrom	Original particle set.
setto	Set to add particles to.

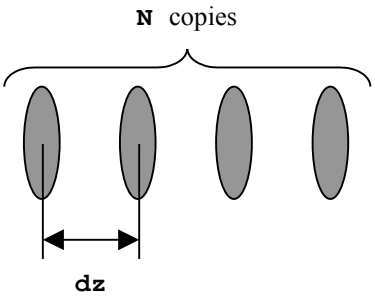
All particles in **setfrom** will be moved to the particles in **setto**.

4.4.1.6 SetCopy

setcopy(set,N,dz) ;
Multi-Copy a particle set in the z-direction

set	Particle set.
N	Number of copies (including original).
dz	Distance between copies in z-direction [m], may be negative.

The **setcopy** element makes **N** duplicates of all particles in a specific set in the z-direction. The spacing between the different bunches is **dz**, as shown below.



To copy a particle set in time domain, please use **settcopy**.

4.4.1.7 SetCurvature

setcurvature(set, Ra, Rc, Lc) ;
Model photo-emission from a curved cathode.

set	Particle set.
Ra	Aperture [m].
Rc	Cathode curvature radius [m]. Specify zero for a flat surface.
Lc	Laser curvature radius [m]. Specify zero for a flat laser front.

This element changes the position and release time coordinates of all particles in the specified set to model photo-emission from a curved cathode.

As illustrated in Figure 4–20, a hollow cathode is simulated with a negative value for **Rc**. This moves all initial particle coordinates in the negative z-direction; the ones nearest to the axis the most. Combined with a flat laser front, a hollow cathode releases particles furthest away from the axis first.

A curved laser front is simulated with a non-zero value for **Lc**. The sign of **Lc** is chosen identical to the sign of **Rc**. Combined with a flat cathode surface, **Rc**=0, a negative value for **Lc** releases particles near the axis first. When **Lc** is equal to **Rc** (with identical sign), the release time of the particles is not affected.

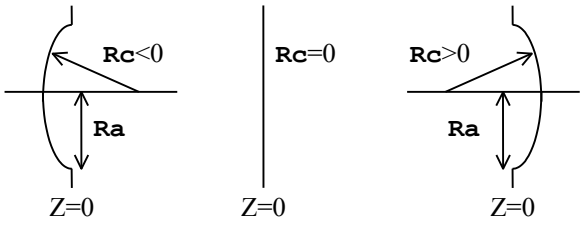


Figure 4–1: Spherical cathode for negative, zero and positive **Rc**.

All particles *i* with r_i>**Ra** or r_i>**Lc** are removed and a warning is printed.

The initial position coordinates are changed as function of particle radius r_i, cathode aperture **Ra** and cathode curvature **Rc**:

$$\Delta z_i = \begin{cases} -\left(\sqrt{\mathbf{Rc}^2 - r_i^2} - \sqrt{\mathbf{Rc}^2 - \mathbf{Ra}^2}\right) & \text{if } \mathbf{Rc} < 0 \\ 0 & \text{if } \mathbf{Rc} = 0 \\ +\left(\sqrt{\mathbf{Rc}^2 - r_i^2} - \sqrt{\mathbf{Rc}^2 - \mathbf{Ra}^2}\right) & \text{if } \mathbf{Rc} > 0 \end{cases} \quad [4.1]$$

The release time coordinates are changed as function of particle radius r_i, the initial position offset Δz_i and the laser curvature **Lc**.

$$\Delta t_i = \begin{cases} -\left(\sqrt{\mathbf{Lc}^2 - r_i^2} - \sqrt{\mathbf{Lc}^2 - La^2}\right)/c - \Delta z_i / c & \text{if } Lc < 0 \\ -\Delta z_i / c & \text{if } Lc = 0 \\ -\left(\sqrt{\mathbf{Lc}^2 - r_i^2} - \sqrt{\mathbf{Lc}^2 - La^2}\right)/c - \Delta z_i / c & \text{if } Lc > 0 \end{cases} \quad [4.2]$$

where *La* is the minimum of **Ra** and |**Lc**|.

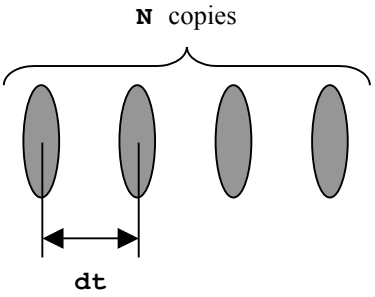
4.4.1.8 SetTCopy

settcopy(set,N,dt) ;

Multi-Copy a particle set in time domain.

set	Particle set.
N	Number of copies (including original).
dt	Distance between copies [s], may be negative.

The **settcopy** element makes **N** duplicates of all particles in a set, equidistantly spaced in time domain. The spacing between the different bunches is **dt**, as shown below.



settcopy is typically used to start a number of identical bunches as function of time. The example below starts five identical 1 ps bunches with 5 ps separation. Divergence was added to better show the effect of the element.

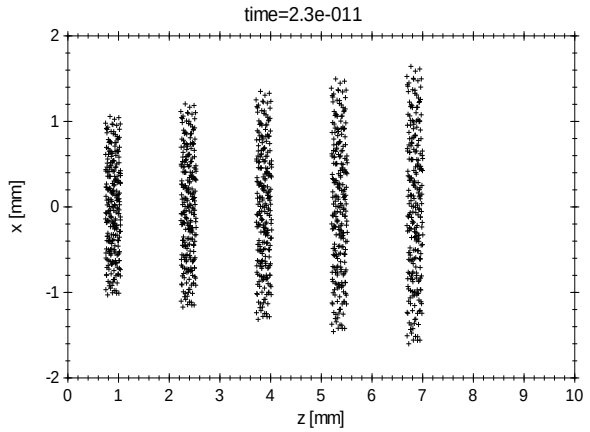


Figure: Five identical 1 ps bunches, started one after the other with, 5 ps separation. The plot shows a snapshot at *t*=23 ps.

```
# Bunch Parameters
G   =    10  ; # Lorentz factor Gamma=10
len =  1e-12 ; # Bunch length 1 [ps]
R   =  1e-3  ; # Initial bunch radius 1 [mm]
div =  1e+3  ; # Divergence of 100 [rad/m]

# Multi-bunch parameters
dt  =  5e-12 ; # Bunch spacing 5 [ns] ;
Nb  =  5     ; # 5 Bunches total

# Create initial cylindrical bunch
setparticles("beam",250,me,qe,0.0) ;
settdist("beam","u",0,len) ;
setrxydist("beam","u",R/2,R) ;
setphidist("beam","u",0,2*pi) ;
setGdist("beam","u",G,0) ;
addxdiv("beam",0,div) ;
addydiv("beam",0,div) ;

# Copy particle set
settcopy("beam",Nb,dt) ;

# Output from 0 to 30 [ps]
tout(0,30e-12,1e-12) ;
```

4.4.1.9 SetShuffle

`setshuffle(set);`
Randomize the order of all particles in the set.

The location of the particles in the specified **`set`** is randomized. Because this will not affect the actual particle coordinates, none of the distributions is affected.

4.4.1.10 SetReduce

`setreduce(set,N);`
Reduce the number of particles in a set.

`set`	Selected particle set
`N`	Final number of particles in the set.

This element randomly removes all but **`N`** particles in the specified set. As shown in Figure 4–2, this operation affects the smoothness and accuracy of the original distribution considerably. Therefore it should never be used without a verification that the resulting particle set is as desired.

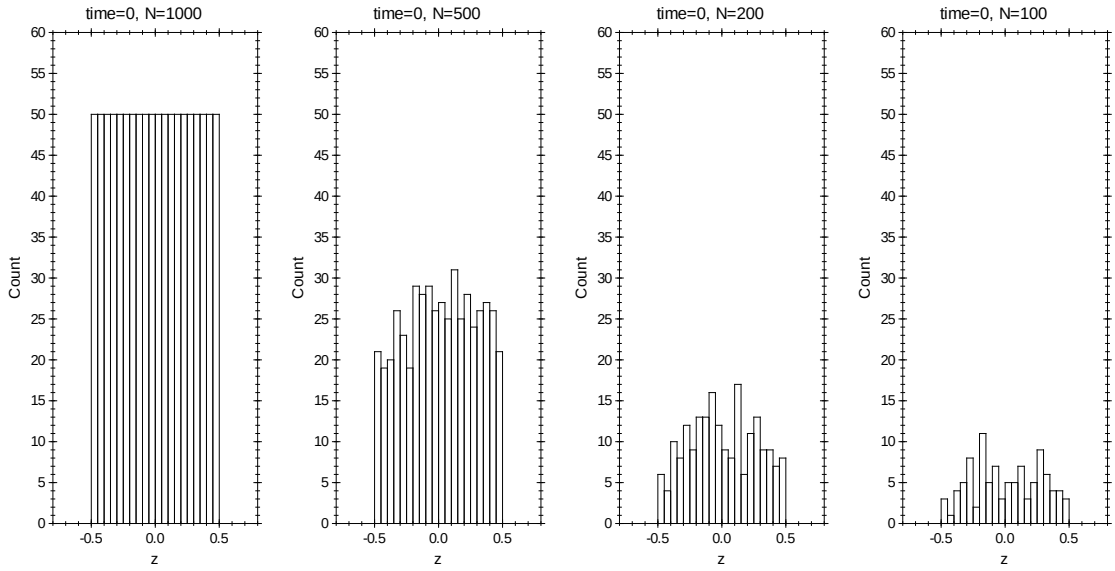


Figure 4–2: Uniform distribution containing 1000 particles (top left), reduced to 500, 200 and 100 particles.

4.4.1.11 SetRmacrodist

setrmacrodist(set,distribution);
Sets the distribution for the particle radii.

The radius of all particles in the **set** is modified according to the specified **distribution**. The distribution is interpreted as a 1D distribution.

This element can be used to specify the radius of the particles in a set. This radius is used by **spacecharge3D**, **spacecharge2Dline** and related elements to assign a volume to the individual macro-particles. **spacecharge2Dcircle** ignores **setrmacrodist** due to the complexity of the equations.

Example: The following line sets the radius of all particles in set **"beam"** to **R**:
setrmacrodist("beam","u",R,0) ;

4.4.1.12 SetStartPar

setstartpar(set,x,y,z,GBx,GBy,GBz,[m,q,n])
Add one macro-particle to a particle set.

set	Name of particle set.
x	Particle *x*-coordinate [m].
y	Particle *y*-coordinate [m].
z	Particle *z*-coordinate [m].
GBx	Particle normalized *x*-momentum: γvx/c.
GBy	Particle normalized *y*-momentum: γvy/c.
GBz	Particle normalized *z*-momentum: γvz/c.
m	Mass [kg] of the elementary particles this macro-particle represents, typically **me**.
q	Charge [C] of the elementary particles this macro-particle represents, typically **qe**.
n	Number of elementary particles this macro-particle represents.

Use this function to start one particle with a specified position and momentum. The mass and charge parameters are optional. When not specified, the inputfile variables **m** and **q** are used. An empty particle set is created before the particle is added if the specified **set** does not exist.

4.4.1.13 SetStartLine

setstartline(set,N,x1,y1,z1,x2,y2,z2,[GBx,Gby,GBz,[m,q,n]])
Start a number of particles on a line.

set	Name of particle set.
N	Number of particles to generate, minimum 2.
x1	Start *x*-position of line [m].
y1	Start *y*-position of line [m].
z1	Start *z*-position of line [m].
x2	End *x*-position of line [m].
y2	End *y*-position of line [m].
z2	End *z*-position of line [m].
GBx	Particle normalized *x*-momentum: γvx/c.
GBy	Particle normalized *y*-momentum: γvy/c.
GBz	Particle normalized *z*-momentum: γvz/c.
m	Mass [kg] of the elementary particles this macro-particle represents, typically **me**.
q	Charge [C] of the elementary particles this macro-particle represents, typically **qe**.
n	Number of elementary particles this macro-particle represents.

Use this function to start **N** particles equidistantly spaced on a straight line, optionally with a specified momentum. The endpoints are always included.

The mass and charge parameters are optional, but can only be specified when the momentum is given. When not specified, the inputfile variables **m** and **q** are used. An empty particle set is created before the particle is added if the specified **set** does not exist.

Example: Start 101 particles between 0 and 10 m on the z-axis.

```
1. setstartline("beam",101, 0,0,0, 0,0,10 ) ;
```

4.4.1.14 SetStartxyzGrid

setstartxyzgrid(set,Lx,dx,Ly,dy,Lz,dz,[GBx,GBy,GBz]) ;
Starts the macro-particles in a rectangular grid with equal velocity.

set	Name of particle set.
Lx	Length of array in x-direction. This length may be zero.
dx	Spacing in x-direction.
Ly	Length of array in y-direction. This length may be zero.
dy	Spacing in **y**-direction.
Lz	Length of array in z-direction. This length may be zero.
dz	Spacing in z-direction.
GBx	Particle normalized *x*-momentum: γvx/c.
GBy	Particle normalized *y*-momentum: γvy/c.
GBz	Particle normalized *z*-momentum: γvz/c.

The particles start in a rectangular grid centered around the origin. The grid extends from –**Lx**/2,–**Ly**/2, –**Lz**/2 to **Lx**/2,**Ly**/2,**Lz**/2 with spacing **dx**,**dy**,**dz**.

It is simple to instruct GPT to initialize billions of particles using this keyword. Since GPT has no built-in limit for the number of particles, it must rely on the operating system to indicate an 'out of memory' condition. On some virtual-memory machines, this error message may come too late, causing either GPT or the complete system to crash.

4.4.2 X, Y and Z manipulators

The following keywords modify the *x*-, *y*- and *z*-coordinates of the particles. The distribution syntax is listed in section 1.6.1.

4.4.2.1 SetXdist

setxdist(set,distribution);
Set *x* coordinate distribution.

Modifies *x* [m] such that $x_i = \mathbf{distribution}_i$

The *x*-coordinate of all particles in the **set** is modified according to the specified **distribution**. The distribution is interpreted as a 1D distribution. To specify a uniform distribution between –1 and 1 centered around zero, the following line can be used:

```
setxdist("beam","u",0,2) ;
```

4.4.2.2 SetXYdistbmp

setxydistbmp(set,filename,[xres,yres]) ;
Sets the xy-projection of the initial particle distribution based on a grayscale bitmap image

set	Particle set.
filename	Name of the grayscale bitmap file.
xres	Horizontal resolution [pixels/meter].
yres	Vertical resolution [pixels/meter].

This element sets the xy-projection of the initial particle distribution according to the specified grayscale bitmap image. Each pixel of the image is treated as a rectangular surface area with a uniformly distributed chance of emitting particles, given by the pixel color. Black pixels have zero chance of emitting particles, white pixels have maximum chance and grayscale values are linearly interpolated in between. This element is very convenient for the simulation of photoemission in for example an rf-photogun, where an actual image of the laser focus on the cathode plane can be used to set the initial particle distribution.

The **filename** must be a grayscale, 8-bit per pixel, uncompressed MS-Windows bitmap file, typically having extension bmp or dib. Reverse row order is supported, but RLE compression is not. A variety of other image formats such as TIFF can easily be converted to grayscale MS-Windows bitmaps with image processing software such as Adobe® Photoshop®.

The pixel size, as stored in the bitmap file, can be overruled with the **xres** and **yres** parameters. The bitmap is placed with its center in the origin. The horizontal and vertical axes of the image are aligned with the *x*- and *y*-axis respectively. To move the center and/or change the alignment, please use the **settransform** element after **setxydistbmp**.

Example:
We used Adobe® Photoshop® to create a 1000x1000 test bitmap of an elliptical hard-edge bunch with checkerboard modulation, as shown in the left plot of Figure 4–3: The following GPT inputfile sets the xy-projection of the initial particle distribution with 100k electrons based on this bitmap.

```
4. setparticles("beam",100000,me,qe,0) ;
5. setxydistbmp("beam","test.bmp") ;
6. tout(0) ;
```

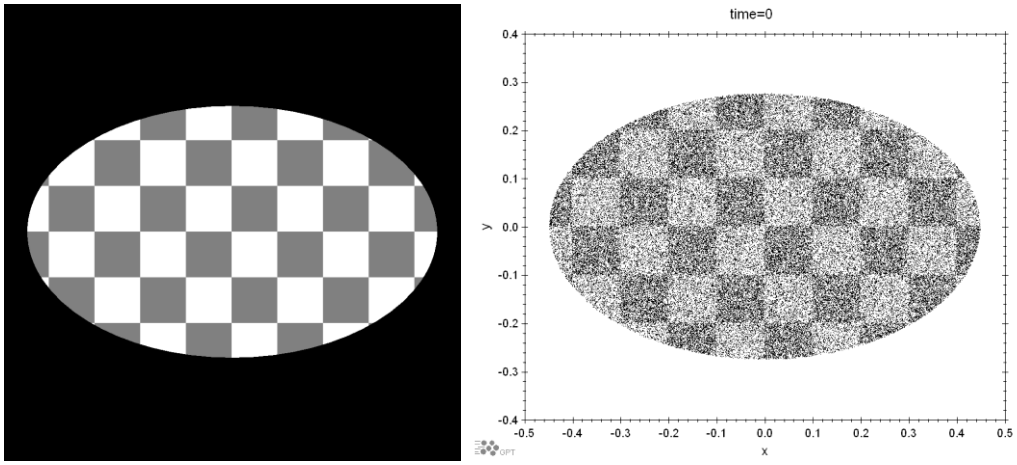


Figure 4–3: Left: Original test.bmp grayscale bitmap of an elliptical bunch with checkerboard modulation. Right: Corresponding GPTwin plot of the initial particle distribution with 100k particles. Because white pixels emit most particles while particles are shown in black in the right plot, the plots look like color-reversed versions of each other.

Technical notes:
Grayscale bitmap files are essentially indexed-color images with 256 or less grayscale color entries. **setxydistbmp** does not use the 8-bit color-index to determine the particle density, but actually looks up the corresponding RGB pixel color from the color table. This ensures correct results even in case the color table is not filled with 256 monotonically increasing grayscale colors. An error message is produced if the red, green and blue values of any pixel in the image are not identical.

The newer V4 and V5 BITMAPINFOHEADER structures are recognized, but the added functionality is ignored because this only concerns color information.

4.4.2.3 SetYdist

setydist(set,distribution);
Set *y*-coordinate distribution.

Modifies *y* [m] such that $y_i = \mathbf{distribution}_i$

The *y*-coordinate of all particles in the **set** is modified according to the specified **distribution**. The distribution is interpreted as a 1D distribution.

4.4.2.4 SetZdist

setzdist(set,distribution);
Set longitudinal particle distribution.

Modifies *z* [m] such that $z_i = \mathbf{distribution}_i$

The *z*-coordinate of all particles in the **set** is modified according to the specified **distribution**. The distribution is interpreted as a 1D distribution.

4.4.2.5 SetRxydist (2D)

setrxydist(set,distribution);
Set radial distribution in *x*-coordinate.

Modifies *x* [m] such that $x_i = \mathbf{distribution}_i \mid 2D$

The *x*-coordinate of all particles in the **set** is modified according to the specified **distribution**. The distribution is interpreted as a 2D distribution. This keyword is typically followed by:
setphidist(set,"u",0,2*pi) ;

4.4.2.6 SetPhidist

setphidist(set,distribution);
Set phi distribution.

Modifies: *x* and, *y* such that:

$$\begin{aligned} rxy_i &= \sqrt{x_i^2 + y_i^2} \\ x_i &= rxy_i \cos(\mathbf{distribution}_i) \\ y_i &= rxy_i \sin(\mathbf{distribution}_i) \end{aligned} \qquad [4.3]$$

The angle in the xy–plane of all particles in the **set** is modified according to the specified distribution. The distribution is interpreted as a 1D distribution. The *z*-distribution of the particles will not be affected.

4.4.2.7 SetEllipse

setellipse(set,a,b,c);
Set the xyz distribution as a uniformly filled ellipsoid.

set	Name of particle set.
a	Size in x-dimension [m].
b	Size in y-dimension [m].
c	Size in z-dimension [m].

The ellipsoid is aligned with the primary axes where **a**, **b** and **c** specify the maximum *x*, *y* and *z* coordinate respectively.

4.4.3 GBx, GBy and GBz manipulators

The following keywords modify the $\gamma\beta_x$-, $\gamma\beta_y$- and $\gamma\beta_z$-coordinates of the particles. In many cases, the Lorentz factor γ is modified simultaneously. The distribution syntax is listed in section 1.6.1.

4.4.3.1 SetGBxdist

setGBxdist(set,distribution);
Sets the momentum distribution in *x*-direction.

Modifies: $\gamma\beta x$ such that $\gamma\beta x_i = \mathbf{distribution}_i$

The $\gamma\beta$x-coordinates of the particles in the **set** are modified according to the specified distribution. The distribution is interpreted as a 1D distribution.

4.4.3.2 SetGBydist

setGBydist(set,distribution);
Sets the momentum distribution in *y*-direction.

Modifies γβ*y* such that $\gamma\beta y_i = \mathbf{distribution}_i$

The γβy-coordinates of the particles in the **set** are modified according to the specified distribution. The distribution is interpreted as a 1D distribution.

4.4.3.3 SetGBzdist

setGBzdist(set,distribution);
Sets the momentum distribution in *z*-direction.

Modifies γβ*z* such that $\gamma\beta z_i = \mathbf{distribution}_i$

The γβz-coordinates of the particles in the **set** are modified according to the specified distribution. The distribution is interpreted as a 1D distribution.

4.4.3.4 SetGBrxydist (2D)

setGBrxydist(set,distribution);
Sets radial distribution in γβ*x*-coordinate.

Modifies γβ*x* such that $\gamma\beta x_i = \mathbf{distribution}_i \mid 2D$

The γβ*x*-coordinate of all particles in the **set** is modified according to the specified **distribution**. Although the distribution is interpreted as a 2D distribution, the γβ*x* coordinate is left unchanged. This keyword is therefore typically followed by:

```
setGBphidist(set,"u",0,2*pi) ;
```

4.4.3.5 SetGBphidist

setGBphidist(set,distribution);
Set γβphi distribution.

Modifies γβ*x* and γβ*y* such that:

$$\begin{aligned} \gamma\beta rxy_i &= \sqrt{\gamma^2\beta x_i^2 + \gamma^2\beta y_i^2} \\ \gamma\beta x_i &= \gamma\beta rxy_i \cos(\mathbf{distribution}_i) \\ \gamma\beta y_i &= \gamma\beta rxy_i \sin(\mathbf{distribution}_i) \end{aligned} \qquad [4.4]$$

The angle in the γβ*x*γβ*y*–plane of all particles in the **set** is modified according to the specified distribution. The distribution is interpreted as a 1D distribution. The theta distribution of the particles will not be affected.

4.4.3.6 SetGBthetadist (2D spherical)

setGBthetadist(set,distribution);
Set the γβ-θ distribution.

Modifies γβx, γβy and γβz such that:

$$\begin{aligned} \gamma\beta rxy_i &= \sqrt{\gamma^2\beta x_i^2 + \gamma^2\beta y_i^2} \\ \gamma\beta rxyz_i &= \sqrt{\gamma^2\beta x_i^2 + \gamma^2\beta y_i^2 + \gamma^2\beta z_i^2} \\ \gamma\beta x_i &= \gamma\beta rxyz_i \sin(\mathbf{distribution}_i)\, \gamma\beta x_i / \gamma\beta rxy_i \\ \gamma\beta y_i &= \gamma\beta rxyz_i \sin(\mathbf{distribution}_i)\, \gamma\beta y_i / \gamma\beta rxy_i \\ \gamma\beta z_i &= \gamma\beta rxyz_i \cos(\mathbf{distribution}_i) \end{aligned} \qquad [4.5]$$

The γβx-, γβy- and γβz-coordinates of the particles in the **set** are modified to create the specified γβ-θ distribution. The distribution is interpreted as a 2D spherical distribution. The γ- and γβ-φ distribution of the particles will not be affected. When γβx and γβy are both zero, a phi angle of zero is assumed.

4.4.3.7 SetGdist

`setGdist(set,distribution);`
Set the γ distribution.

Modifies γβz such that:

$$\begin{aligned} \gamma &= \texttt{distribution} \\ \gamma\beta z &= \sqrt{\gamma^2 - \gamma^2\beta_x^2 - \gamma^2\beta_y^2 - 1} \end{aligned} \qquad [4.6]$$

The γβz-coordinates of the particles in the **`set`** are modified to create the specified γ-distribution. The distribution is interpreted as a 1D distribution.

Useful equations:

$$\begin{aligned} &\text{Lorentz factor} && \gamma = 1/\sqrt{1-\beta^2} \\ &\text{Kinetic energy} && E\,[eV] = (\gamma - 1)\, mc^2 / |q_e| \end{aligned} \qquad [4.7]$$

4.4.3.8 SetGBxemittance

`setGBxemittance(set,emittance);`
Sets the emittance in xβx-space. Modifies: γβx.

`set`	Selected particle set
`emittance`	Requested emittance in xβx-space in [m-rad]

This element sets the emittance in xβx-space by scaling all γβx-coordinates linearly using:

$$\begin{aligned} &x_c = x - \bar{x} \quad x'_c = \gamma\left(\beta_x - \bar{\beta}_x\right) \\ &\gamma\beta x = \gamma\beta x \frac{\texttt{emittance}}{\sqrt{\overline{x_c^2}\cdot\overline{x'^2_c} - \overline{x_c x'_c}^2}} \end{aligned} \qquad [4.8]$$

Because this element modifies the γβx-coordinates, it should not be used after **`setGdist`**. A warning is printed when over one order of magnitude or more needs to be scaled.

This element sets the emittance in transverse position and momentum space. Therefore, the results can be slightly different from the results calculated by the **`nemixrms`** routines, especially in the presence of a large energy spread.

The element assumes all particles in the set to represent an equal amount of elementary particles.

4.4.3.9 SetGByemittance

`setGByemittance(set,emittance);`
Sets the emittance in yβy-space. Modifies: γβy.

`set`	Selected particle set
`emittance`	Requested emittance in yβy-space in [m-rad]

This element sets the emittance in yβy-space by scaling all γβy-coordinates linearly using:

$$\begin{aligned} &y_c = y - \bar{y} \quad y'_c = \gamma\left(\beta_y - \bar{\beta}_y\right) \\ &\gamma\beta y = \gamma\beta y \frac{\texttt{emittance}}{\sqrt{\overline{y_c^2}\cdot\overline{y'^2_c} - \overline{y_c y'_c}^2}} \end{aligned} \qquad [4.9]$$

Because this element modifies the γβy-coordinates, it should not be used after **`setGdist`**. A warning is printed when over one order of magnitude or more needs to be scaled.

This element sets the emittance in transverse position and momentum space. Therefore, the results can be slightly different from the results calculated by the **nemiyrms** routines, especially in the presence of a large energy spread.

The element assumes all particles in the set to represent an equal amount of elementary particles.

4.4.4 X–GBx, Y–GBy and Z–G manipulators

The following keywords skew the phase-space areas. They will not affect the transverse or longitudinal emittance, nor will they affect the *x*-, *y*- and *z*-distributions. The distribution syntax is listed in section 1.6.1.

4.4.4.1 AddXDiv

addxdiv(set,xc,div);
Add divergence in the x-βx plane.

set	Selected particle set.
xc	Center of *x*-distribution [m], typically 0.
div	Divergence in [rad/m]

The γβx-coordinate is modified by:

$$\begin{aligned} \gamma\beta_x &= \gamma\beta_x + \mathbf{div}\cdot(x-\mathbf{xc}) \\ \gamma\beta_z &= \sqrt{\gamma^2-\gamma^2\beta_x^2-\gamma^2\beta_y^2-1} \end{aligned} \qquad [4.10]$$

A linear x-γβx-dependence is added, keeping the transverse emittance constant. To keep the particle energy and the longitudinal emittance constant, the γβz-coordinte is modified accordingly.

A bunch is typically characterized by a radius *R* and a divergence angle θ. However, in this element it is not possible to account for the beam radius because it is differently defined for uniform and Gaussian transverse distributions. Therefore the divergence must be specified in [rad/m], resulting in $\theta = \mathbf{div}\,R/\gamma$.

4.4.4.2 AddYDiv

addydiv(set,yc,div);
Add divergence in the y-βy-plane

set	Selected particle set.
yc	Center of *y*-distribution [m], typically 0.
div	Divergence in [rad/m]

The γβ*y*-coordinate is modified by:

$$\begin{aligned} \gamma\beta_y &= \gamma\beta_y + \mathbf{div}\cdot(y-\mathbf{yc}) \\ \gamma\beta_z &= \sqrt{\gamma^2-\gamma^2\beta_x^2-\gamma^2\beta_y^2-1} \end{aligned} \qquad [4.11]$$

A linear y-γβy dependence is added, keeping the transverse emittance constant. To keep the particle energy and the longitudinal emittance constant, the GBz-coordinte is modified accordingly.

4.4.4.3 AddZDiv

addzdiv(set,zc,div);
Add divergence in the z-γ-plane.

set	Selected particle set.
zc	Center of *z*-distribution [m].
div	Divergence in [eV/m].

The γβz-coordinate is modified by:

$$\gamma_{new} = \gamma_{old} + \frac{\mathbf{div} \cdot |q_e|}{mc^2} \cdot (z - \mathbf{zc})$$
$$\gamma\beta_z = \sqrt{\gamma_{new}^2 - \gamma^2\beta_x^2 - \gamma^2\beta_y^2 - 1} \qquad [4.12]$$

A linear z-γ-dependance is added, keeping the longitudinal emittance constant. This has no effect on the transverse emittance or particle distribution.

4.4.5 Time-dependent particle distributions

Default, all particles are emitted at once at t=0 in the simulation. This can be changed using the elements presented in this section to start particles at any time in the simulation.

4.4.5.1 SetTdist

`settdist(set,distribution);`
Set particle-release time distribution.

Modifies **`t`** [s] such that $t_i = \mathbf{distribution}_i$

The release time of the particles in the **`set`** is modified according to the specified **`distribution`**. The distribution is interpreted as a 1D distribution. To specify a uniform distribution between –1 ns and 1 ns centered around zero, the following line can be used:

```
settdist("beam","u",0,2e-9) ;
```

4.5 Spacecharge

This section describes the built-in space-charge routines of GPT. Please see section 1.5 for a brief comparison between the different routines.

It is possible to write new and/or more efficient space-charge models for specific situations. The procedure to develop such custom space-charge models is given in the GPT Programmer's Reference.

GPT uses the concept of macro-particles: Every particle in the simulations represents a (large) number of elementary particles. In the case of simulations of granularity/stochastic effects, every particle in the simulations must represent exactly one elementary particle. In other words, $Q_{tot}=N\cdot q$, where Q_{tot} is the total charge, N the number of particles and q the charge of an elementary particle. The simplest method is to set the total charge based on the number of particles (assuming electrons, and assuming **`N`** is defined elsewhere):

```
Qtot = N*qe ;
setparticles("beam",N,me,qe,Qtot)
```

Alternatively, it is possible to calculate the number of particles based on the total charge **`Qtot`** (assuming electrons, and assuming **`Qtot`** is defined elsewhere):

```
N = round(|Qtot/q|) ;
setparticles("beam",N,me,qe,Qtot)
```

where the **`round()`** function rounds towards the nearest integer and where **`||`** indicates the absolute value.

4.5.1 Spacecharge3Dmesh

```
spacecharge3Dmesh([option,parameter(s)],...) ;
```

Mesh-based 3D space-charge routine.

We would like to express our thanks to Dr Gisela Pöplau from Rostock University, Germany for the development of the non-equidistant 3D multi-grid Poisson solver, tailor-made for this application.

4.5.1.1 Syntax and options

The **`sc3dmesh`** element calculates 3D space-charge fields by solving Poisson's equations on a non-equidistant mesh adapted to the beam geometry. The model is very fast, allowing over a million particles to be tracked on a normal PC. The element scales almost linear with the number of particles in terms of CPU time. This element is not suitable for the simulation of the effect of short-range Coulomb interactions such as disorder induced heating. Please see section 4.5.2 for the correct approach for calculating point-to-point interactions.

The charge density is determined on a mesh in the rest frame of the bunch where no self-magnetic fields are present. A non-equidistant multi-grid technique is used to solve Poisson's equation to obtain the electrostatic potential. Interpolations and Lorentz transformations are used to transform the particle coordinates to the rest frame and to transform the calculated fields back to the laboratory frame.

Note: This element gives a poor estimate of the space-charge forces in bunches with a large energy spread, because particles still have relatively large velocities in the rest frame of the bunch. A warning is printed to inform the users when this is the case.

Option	Parameter(s)	Default	Description
`"RestMaxGamma"`	**`Gmax`**	2.0	Triggers a warning when particles have a Lorentz

Option	Parameter(s)	Default	Description
			factor larger than **Gmax** in the zero-momentum frame of the bunch.
"MeshNtotal"	**[Nx,Ny,Nz]**		Use a total of **Nx**, **Ny** and **Nz** meshlines in the x-, y- and z-direction respectively to fill the bounding box.
"MeshNbunch"	**Nx,Ny,Nz**		Slice the particle bunch in **Nx**, **Ny** and **Nz** meshlines in the x-, y- and z-direction respectively. Additional meshlines are added in all directions to fill the bounding box.
"MeshNfac"	**Alpha**	1	Automatically determine the total number of meshlines in the bunch using the following equation: **Nx**=**Ny**=**Nz**=**alpha***$N_{\text{particles}}$^**mpow**. Can be used in combination with **MeshNtotal** without parameters to specify the total number of meshlines in the bounding box instead of in the bunch.
"MeshNpow"	**mpow**	1/3	
"MeshAdapt"	**Fn**	0.5	Sets the mesh adaptivity. The meshing algorithm ensures that the maximum difference in distance between neighboring meshlines is always kept less than the specified **fn**.
"MemOvershoot"	**n**	10	Percentage of memory overhead consumed by the memory management scheme. If set to 0, all block memory management is disabled resulting in the lowest memory consumption at the cost of a significant increase in page faults and CPU time.
"MeshBoxSize"	**Nstd**	5	Sets the minimum size of the bounding box in units of standard deviation for each dimension.
"MeshBoxAccuracy"	**macc**	0.3	Sets the minimum size of the bounding box, as a fraction of the effect of image charges in the walls of the box.
"MeshMaxAspect"	**A**	∞	Enforces the maximum ratio between the largest and smallest mesh-spacing in each dimension to be less than **A**. Because this restriction is enforced by reducing the mesh adaptivity, using this option can result in a dramatic increase in the number of meshlines.
"SolverAcc"	**Verror**	0.01	Sets the accuracy of the multigrid Poisson solver.
"SolverMethod"		**"MGCG"**	Sets the method of the Poisson solver to any of the options listed below.
	"MG"		Multigrid algorithm.
	"MGCG"		Multigrid preconditioned cg-algorithm.
	"CG"		Preconditioned cg-algorithm.
	"SOR"		Use SOR procedure, for comparisons only.
"SolverStartZero"			Start from zero potential. If not specified, the solver starts from the previously obtained solution if the mesh size equal to mesh size of previous step.
"Cathode"			Simulates a flat cathode plane by setting a dirichlet boundary condition at z=0. Truncates the bounding box at all negative z-coordinates.
"BoundaryOpen"			Use open boundary conditions.
"BoundaryXperiodic"	**Xsize**		Use periodic boundary conditions in X. Not

Option	Parameter(s)	Default	Description
			implemented yet.
"BoundaryYperiodic"	**Ysize**		Idem in Y.
"BoundaryZperiodic"	**Zsize**		Idem in Z.
"BoundaryXYpipe"	**Xsize,Ysize**		Use Dirichlet boundary conditions on a beam-pipe with elliptical cross section in the xy-plane. Work in progress…
"Boundaries"	**String**	**"DDD DDD"**	Specify boundary conditions for each boundary individually, as a string in the following order: x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}. Options are several combinations of 'DAOPEN', see section 4.5.1.8.
"BeamScale"	**small, large**	10^{-4} 10^{+4}	Enforces the bunch aspect-ratio between **small** and **large** using a linear coordinate transform. The electromagnetic fields are adapted accordingly, but results will be incorrect when feature sizes in the bunch with an aspect ratio outside the specified range are present.

The typical use of the **sc3dmesh** element is to use the default settings:

```
sc3dmesh() ;
```

To overwrite any of the default settings, the name of the option must be specified followed by one or more parameters. For example, to set a fixed number of 33 meshlines in all directions, the following line must be used:

```
sc3dmesh("MeshNtotal",33,33,33) ;
```

The order of independent options is irrelevant. As a result, to combine a fixed number of meshlines with the simulation of a cathode plane at z=0, either one of the following lines can be used:

```
sc3dmesh("Cathode","MeshNtotal",33,33,33) ;
sc3dmesh("MeshNtotal",33,33,33,"Cathode") ;
```

4.5.1.2 Introduction to the algorithms

The chosen algorithm for this space-charge model in GPT is a mesh-based Poisson solver in the rest frame of the bunch. First the bunch is transformed to the frame with no average momentum. In this rest frame, only electrostatic forces are present as long as the velocities in this frame are non-relativistic. The bunch length in the rest frame is longer by the Lorentz factor γ.

Subsequently, a Cartesian mesh is generated in a box around the bunch. The distribution of the meshlines is non-equidistant and adapted to the projected charge density. To reduce the number of meshlines needed, an adaptive meshing technique is used. The purpose of this scheme is to both reduce 'wasted' CPU time in empty volume and reduce numerical errors by choosing more meshlines in regions with higher charge density.

The non-equidistant mesh is used to store an approximation of the charge density on the corners of the mesh. This information is obtained using a linear distribution of the charge of every particle over the eight corners of its enclosing meshbox.

The charge density on the mesh is fed into a Poisson solver to obtain the potential. A state-of-the-art multigrid Poisson solver has been constructed for the non-equidistant meshes described in subsection 4.5.1.8. It scales linearly in CPU time with the number of meshnodes. Selectable Dirichlet or open boundary conditions allow the simulation of bunches within pipes with rectangular cross section, bunches near a cathode and a bunch in open space.

The resulting potential is interpolated and differentiated using a 2^{nd} order interpolation scheme to obtain the electric field in the rest frame of the bunch at all particle coordinates. This electric field is transformed back to the laboratory frame to obtain the electric and magnetic fields. The GPT kernel combines these fields with the external fields in the tracking engine.

The chosen algorithm implies that the velocity in the rest frame of the bunch is non-relativistic. This assumption is not always true for long bunches in the first stages of acceleration. Therefore, it is possible that new electrons are emitted with an energy in the eV level while the front of the bunch has already been accelerated to relativistic velocities. This is a known limitation of the routine reducing the applicability.

4.5.1.3 Rest frame

The first step in calculating the space-charge forces is the determination of a rest-frame velocity. This is the frame in which all particle velocities are (nearly) zero, resulting in an electrostatic field only. Because not all particles have equal velocity in a bunch with energy spread, a shared rest frame typically does not exist. The rest frame is then defined as the frame in which the total momentum is zero. The velocity $\boldsymbol{\beta}_o$ of this frame is defined by the average of all particle velocities, weighted by the Lorentz factor γ:

$$\boldsymbol{\beta}_o = \frac{\overline{\gamma\boldsymbol{\beta}}}{\bar{\gamma}} = \frac{\sum n_i \gamma_i \boldsymbol{\beta}_i}{\sum n_i \gamma_i} \qquad [4.13]$$

where n_i is the number of elementary particles represented by the macro-particle i. The normalized momentum $\gamma_o\boldsymbol{\beta}_o$ and the Lorentz factor γ_o of the rest frame are then given by:

$$\begin{aligned} \gamma_0\boldsymbol{\beta}_o &= \frac{\boldsymbol{\beta}_o}{\sqrt{1-\boldsymbol{\beta}_0^2}} \\ \gamma_o &= \sqrt{1+\gamma_0^2\boldsymbol{\beta}_0^2} \end{aligned} \qquad [4.14]$$

When there is energy spread, the velocity of the rest frame is identical to the average velocity only when all velocities are non-relativistic.

The particle coordinates $\mathbf{r}_i$ are different in the rest frame, denoted with a prime. They are a factor γ_0 expanded in the direction of the velocity of the frame:

$$\begin{cases} \mathbf{r}_i{}'_{\perp} = \mathbf{r}_{i\perp} \\ \mathbf{r}_i{}'_{//} = \gamma_0 \mathbf{r}_{i//} \end{cases} \Rightarrow \mathbf{r}_i' = \mathbf{r}_i + \frac{\mathbf{r}_i \cdot \gamma_0\boldsymbol{\beta}_0}{\gamma_0 + 1}\,\gamma_0\boldsymbol{\beta}_0 \qquad [4.15]$$

The particle velocity $\mathbf{u}_i$ hereby changes also:

$$\begin{cases} \mathbf{u}_i{}'_{\perp} = \dfrac{\mathbf{u}_{i\perp}}{\gamma_0\left(1-\boldsymbol{\beta}_u\cdot\boldsymbol{\beta}_0\right)} \\ \mathbf{u}_i{}'_{//} = \dfrac{\mathbf{u}_{i//}}{\left(1-\boldsymbol{\beta}_u\cdot\boldsymbol{\beta}_0\right)} \end{cases} \Rightarrow \mathbf{u}' = \frac{c}{\gamma_u\gamma_0 - \gamma_u\boldsymbol{\beta}_u\cdot\gamma_0\boldsymbol{\beta}_0}\left[\left(\frac{\gamma_u\boldsymbol{\beta}_u\cdot\gamma_0\boldsymbol{\beta}_0}{\gamma_0+1} - \gamma_u\right)\gamma_0\boldsymbol{\beta}_0 + \gamma_u\boldsymbol{\beta}_u\right] \qquad [4.16]$$

It's more convenient to use the Lorentz factor given by:

$$\gamma_{u'} = \gamma_0\gamma_u\,(1-\beta_0\beta_u) \qquad [4.17]$$

Ideally, all velocities in the rest frame are well below the speed of light. If this is not the case, a warning is printed counting the number of particles with a Lorentz factor $\gamma_{u'}$ larger than **`Gmax`**. The parameter **`Gmax`** can be set with the **`"RestMaxGamma"`** option.

4.5.1.4 Bounding box

To calculate the electromagnetic potential, a Cartesian 3D mesh is used to store an estimate of the charge in the rest frame of the bunch. To solve the potential in free space, an infinite mesh is required. Because this is impractical, the actual mesh is truncated at a bounding box where the potential is assumed to be zero. The center of this bounding box is chosen as the average particle coordinate in the rest frame of the bunch. The effect of a bounding box (with Dirichlet boundary conditions) is similar to the effect of image charges.

Choosing a too large box is a waste of CPU time and memory, while choosing a too small box violates the assumption that the potential at the edges of the bounding box is zero. A combination of two options is available to set the correct size of the bounding box. The minimum size of the bounding box is always **`nstd`** times the standard deviation in each dimension, as can be set with the **`"MeshBoxSize"`** option.

Unfortunately, for low and high aspect ratio bunches this sets a too small box for the transverse and longitudinal dimensions respectively. The **`"MeshBoxAccuracy"`** option can be used to define the

maximum ratio of the actual fields divided by the fields produces by the image charges. The calculation is based on analytical expressions for a uniformly charged brick, with equal RMS properties as the actual bunch.

4.5.1.5 Number of meshlines

The bounding box is divided by a number of Carthesian meshlines. The meshlines form the edges of rectangular boxes filling the calculation domain. As explained in subsequent sections, the corners of the boxes, the crossings of the meshlines, are used to store an estimate of the charge density and to calculate the electrostatic fields.

Figure 4–4 shows a schematic of a typical meshline distribution in one dimension. A number of meshlines within the bunch dimension, N_{bunch}, are positioned more closely together where more charge is present. The lower and upper parts of the bounding box are filled with N_{low} and N_{upp} meshlines repectively, where the spacing between the meshlines gradually increases near the edge of the bounding box. The total number of meshlines is N_{total}.

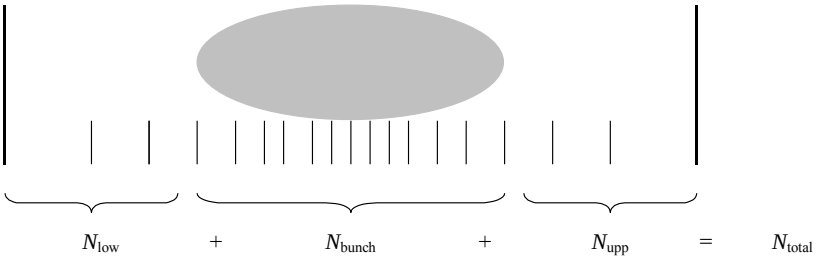


Figure 4–4: One-dimensional schematic of a typical meshline distribution.

By default, the number of meshlines in the bunch, N_{bunch}, is equal in all directions and chosen as a function of the number of particles N:

$$N_{bunch} = \mathbf{alpha}\, N^{\mathbf{mpow}}$$

where **alpha** and **mpow** can be set with the **"MeshNfac"** and **"MeshNpow"** options.

Alternatively, the number of meshlines slicing the bunch can be specified with the **"MeshNbunch"** option. Then, the number of meshlines in the bunch is fixed at **Nx**, **Ny** and **Nz** in the x-, y- and z-direction respectively and set independent on the number of particles in the bunch. In both cases, the space between the bunch and the edges of the bounding box is filled with as few additional meshlines as possible to save CPU and memory usage, see section 4.5.1.6.

A disadvantage of the options above is that an a-priori unknown number of meshlines are added to fill the bounding box. This can result in unpredictable memory usage and CPU time. To avoid this, the **"MeshNtotal"** option sets the total number of meshlines in the bounding box. When **"MeshNtotal"** is specified without further parameters, the total number of meshlines, N_{total}, is equal in all directions and given by:

$$N_{total} = \mathbf{alpha}\, N^{\mathbf{mpow}} \qquad [4.18]$$

With parameters, the total number of meshlines is **Nx**, **Ny** and **Nz** in the x-, y- and z-direction respectively.

4.5.1.6 Meshline positions

The positioning of the meshlines is an iterative process. To determine the position of the x'-lines, planes in the yz-plane, first a histogram of the x'-coordinates of the particles is created. To obtain a good trade-off between bin width and noise, the number of bins is taken equal to the square root of the number of particles, see Figure 4–5.

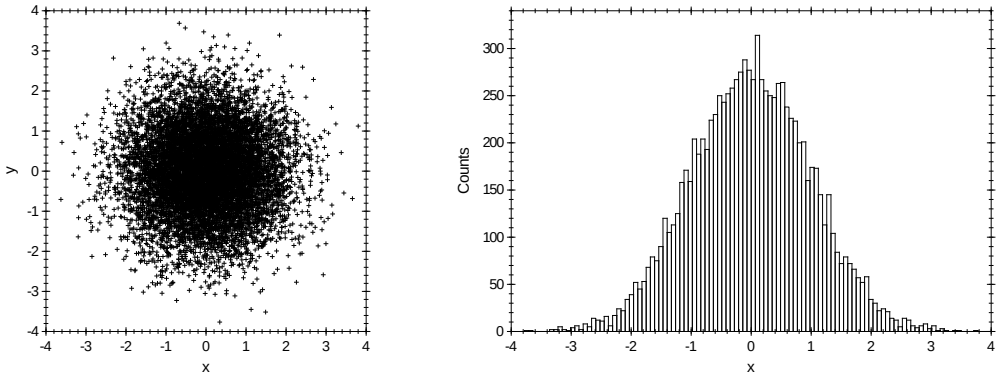


Figure 4–5: Particle positions in the xy-plane and a corresponding x-histogram for a Gaussian charge-density distribution. 10,000 sample particles are used for a Gaussian distribution with σ=1.

The next step approximates the Cumulative Density Function (CDF) by integrating the histograms. To compensate for the fact that we use three projections to adapt the meshline spacing to the charge density, the cubic root of the height of the histograms is used. This typically broadens the distribution. The inverse of this CDF is used to map Nx equidistantly spaced points to Nx meshline positions following the beam density.

The above procedure, however, cannot be used directly because it allows very large differences in neighboring mesh intervals, creating a non-optimal meshline distribution for the Poisson solver described in section 4.5.1.8. Therefore, the **fn** parameter is introduced to restrict the difference in spacing between neighboring meshlines. When **fn** is 0.25, this means that the difference in spacing between neighboring meshlines cannot vary by more than 25%. As an example, any mesh spacing of 1 mm with **fn**=0.25 enforces a spacing between 0.8 and 1.25 mm for its neighbours. The parameter **fn** can be set with the **"MeshAdapt"** option.

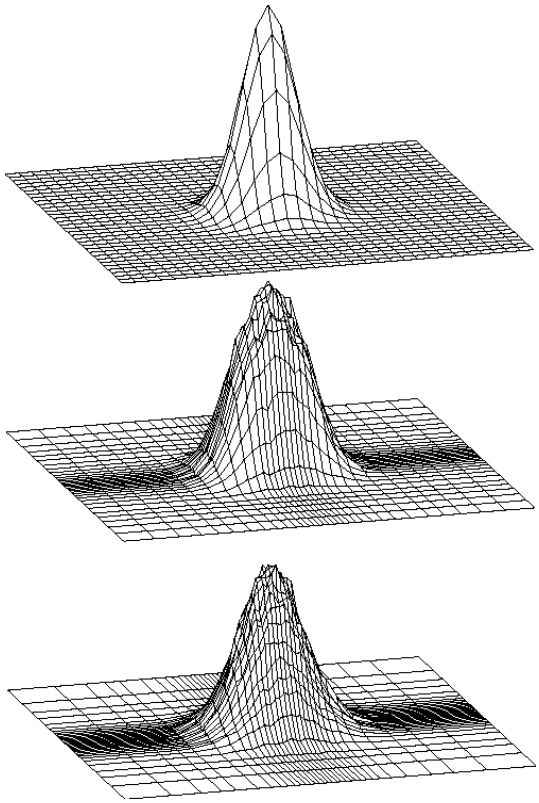


Figure 4–6: xy-meshline positions for a Gaussian charge density with **fn**=0 (top), **fn**=0.2 (middle) and **fn**=0.5 (bottom), with a bounding box of 5 sigma. The vertical axis shows the total charge in each meshbox, where the height of the top has been normalized in all plots individually.

To span the mesh over the bounding box defined in section 4.5.1.4, additional meshlines are added left and right of the bunch. The spacing of these additional meshlines is increased towards the boundary, restricted to **fn**, to add as few lines as possible. If a fixed number of meshlines is chosen, the number of meshlines in the bunch decreases proportionally with the increase of meshlines between the edge of the bunch and the edge of the bounding box.

The addition of meshlines and the effect of **fn** is shown in Figure 4–6H. When **fn**=0, the spacing between all neighboring meshlines is allowed to vary by 0%, creating an equidistant mesh. Such a mesh is most stable to solve for the Poisson solver, but it will create many empty meshboxes. On the other extreme, setting **fn**=0.5 results in dense sampling of the electron bunch and sparse sampling of the surrounding area.

4.5.1.7 Charge on mesh

After the meshlines are positioned, the charge of all particles is distributed over the corners of the meshboxes using a trilinear interpolation technique. For every particle, the corresponding meshbox index (n_x,n_y,n_z) is found by three separate bisection searches for the three dimensions. Then the relative distances t, u and v to the lower, left, bottom corner of the box are calculated from the meshlines positions l:

$$\begin{aligned} t &= \left(x_i - l_x(n_x)\right)/\left(l_x(n_x+1) - l_x(n_x)\right) \\ u &= \left(y_i - l_y(n_y)\right)/\left(l_y(n_y+1) - l_y(n_y)\right) \\ v &= \left(z_i - l_z(n_z)\right)/\left(l_z(n_z+1) - l_z(n_z)\right) \end{aligned} \qquad [4.19]$$

The charge Q_i of each particle i is distributed over all eight corners $Q_{mesh}(n_x,n_y,n_z)$ of the enclosing meshbox as follows:

$$\begin{aligned} Q_{mesh}(n_x,n_y,n_z) &= Q_i\ (t-1)\,(1-u)\,(1-v) \\ Q_{mesh}(n_x,n_y,n_z+1) &= Q_i\ (t-1)\,(1-u)\,v \\ Q_{mesh}(n_x,n_y+1,n_z) &= Q_i\ (t-1)\,u\,(1-v) \\ Q_{mesh}(n_x,n_y+1,n_z+1) &= Q_i\ (t-1)\,u\,v \\ Q_{mesh}(n_x+1,n_y,n_z) &= Q_i\ t\,(1-u)\,(1-v) \\ Q_{mesh}(n_x+1,n_y,n_z+1) &= Q_i\ t\,(1-u)\,v \\ Q_{mesh}(n_x+1,n_y+1,n_z) &= Q_i\ t\,u\,(1-v) \\ Q_{mesh}(n_x+1,n_y+1,n_z+1) &= Q_i\ t\,u\,v \end{aligned} \qquad [4.20]$$

4.5.1.8 Solve potential

The electrostatic potential V in the rest frame of the bunch satisfies Poisson's equation:

$$-\nabla^2 V = \rho/\varepsilon_0 \text{ in } \Omega, \qquad [4.21]$$

where ρ is the charge density given in units of [C/m^3] and Ω denotes the bounding box explained in section 4.5.1.4.

Possible boundary conditions are:

'D': Dirichlet boundary conditions with V=0 on $\partial\Omega$.

'A': Dirichlet boundary conditions with an approximate value for the potential at the boundary based on the analytical expressions of a uniformly charged brick with equal RMS sizes.

'O': Open boundary conditions, which describe the decay of the potential of the bunch and ensure that $\lim_{r\to\infty} V = 0$.

'P': Periodic boundary condition. Not implemented yet.

'E': Elliptical boundary condition. Work in progress…

'N': Neumann boundary condition with $\partial V/\partial n = 0$ on $\partial\Omega$.

Poisson's equation is discretized by means of second-order finite differences. The generation of the underlying mesh is described in section 4.5.1.6. The resulting linear system of equations $AV_{mesh} = Q_{mesh}$ is solved by means of a multigrid (MG) technique adapted to non-equidistant grids. A more detailed description of this solver can be found in [7F8, 8F9].

The input of the Poisson solver are the meshline positions and the charge on all meshnodes $Q_{mesh}(n_x,n_y,n_z)$. The output is the potential on all meshnodes $V_{mesh}(n_x,n_y,n_z)$.

Two multigrid schemes are implemented: The most stable algorithm is the multigrid preconditioned conjugate gradient method (MGCG) that is set as default. For saving CPU time it is possible to choose the plain MG algorithm for bunches with 'good' behavior:

```
sc3dmesh("SolverMethod","MG") ;
```

The number of multigrid iterations is determined by the option `"SolverAcc"` with the default setting 10^{-2}. That means for the relative residual

$$\frac{|\,AV_{mesh}^{(n)} - Q_{mesh}\,|_{\max}}{|\,Q_{mesh}\,|_{\max}} \le 10^{-2}, \qquad [4.22]$$

where $V_{mesh}^{(n)}$ denotes the solution after n multigrid iterations. In our experience this solver accuracy suffices for most applications. A smaller number usually does not improve the result because all discretizations occurring in the simulation (this also applies to the tracking itself) restrict the accuracy that can be achieved. Furthermore the achievable accuracy is restricted by the noise that the charge density contains.

In cases the multigrid Poisson solver does not provide acceptable results, or does not even converge, several parameters can be changed:

Change the size of the bounding box
In several cases enlarging the bounding box helps. This leads to a better approximation of the field near the surface of this bounding box (see section 4.5.1.4) in cases where the field of the bunch decays.
If the potential of the bunch has a decay of at least as 1/r, the difference between simulations with Dirichlet or open boundaries is not essential. In case the stability of the algorithm turns out to be a problem Dirichlet boundary conditions should be preferred.

Change the Poisson solver algoritnm
Beside the multigrid algorithms two other Poisson solvers are implemented: A preconditioned conjugate gradient (CG) method with the diagonal of the matrix *A* as preconditioner and the SOR relaxation. The related commands are:
`sc3dmesh("SolverMethod","CG") ;` and `sc3dmesh("SolverMethod","SOR") ;` respectively.
Although both algorithms are much slower than the multigrid method (CG faster that SOR) they can be applied in order to verify that the problem is due to the solver (for instance no convergence).

Change the discretization:
Problems with the Poisson solver are mostly caused by the discretization. As already mentioned in section 4.5.1.6 large differences in neighboring step sizes can be the underlying reason. Furthermore, the global aspect ratio h_{max}/h_{min} must not be too large. A smaller value for `fn`, which controls the step sizes of the mesh, therefore improves the discretization of Poisson's equation, e.g. `sc3dmesh("MeshAdapt",0.2);`
Another possibility to verify the mesh is to vary the number of meshlines (see section 4.5.1.5). A general advice cannot be given because an appropriate choice very much depends on the characteristics of the application. Note that a larger number of meshlines specified with `"MeshNtotal"` usually provides an undesired larger aspect ratio!

4.5.1.9 Interpolate E' from V'

The electric field in the rest frame of the bunch is given by the derivative of the potential:

$$\mathbf{E}' = -\nabla V \qquad [4.23]$$

Centered differencing and a trilinear interpolation scheme are used to calculate this derivative at all particle positions. First, for every particle, the procedure described in 4.5.1.7 is again used to find the enclosing meshbox index (n_x,n_y,n_z). To obtain the x-component of the electric field, first the electric fields interpolated at the left and at the right sides of the box, trilinear interpolated to the yz-position of the particle, are calculated using centered differencing:

$$E_{x,left} = \frac{\begin{array}{l}\big((1-u)(1-v)\,V(-1,0,0) + (1-u)v\,V(-1,0,1) + u(1-v)\,V(-1,1,0) + uv\,V(-1,1,1)\big) - \\ \big((1-u)(1-v)\,V(+1,0,0) + (1-u)v\,V(+1,0,1) + u(1-v)\,V(+1,1,0) + uv\,V(+1,1,1)\big)\end{array}}{l_x(n_x+1) - l_x(n_x-1)}$$

$$E_{x,right} = \frac{\begin{array}{l}\big((1-u)(1-v)\,V(0,0,0) + (1-u)v\,V(0,0,1) + u(1-v)\,V(0,1,0) + uv\,V(0,1,1)\big) - \\ \big((1-u)(1-v)\,V(2,0,0) + (1-u)v\,V(2,0,1) + u(1-v)\,V(2,1,0) + uv\,V(2,1,1)\big)\end{array}}{l_x(n_x+2) - l_x(n_x)} \qquad [4.24]$$

The notation $V(a,b,c)$ is used to represent the potential V at index (n_x+a,n_y+b,n_z+c).

Linear interpolation is used to obtain the final x-component of the electric field at the particle position:

$$E_x = (1-t)\,E_{x,left} - left + t\,E_{x,right} \qquad [4.25]$$

The equations for the y- and z-components of the electric field are fully analogous.

4.5.1.10 Aspect-ratio scaling

Very low and very high aspect ratio bunches are difficult to simulate with the described mesh-based space-charge routine. For relatively uniform bunches, an option to increase the accuracy is to multiply the particle coordinates in one dimension such that the aspect ratio of the bunch falls in the range between 0.01 and 100.

Very high aspect ratio bunches (A>100) have nearly identical fields when the total charge is constant. From a physical point of view this is clear, because in the 'pancake' regime the bunch behaves as a surface charge and the precise thickness of this surface does not affect the fields. As a result, it is safe to scale the longitudinal coordinate of all particles in a pancake bunch as long as it stays sufficiently flat. Symbolically:

$$\text{If } \begin{cases} stdz < stdx/S_{\text{large}} \\ stdz < stdy/S_{\text{large}} \end{cases} \Rightarrow \begin{cases} S = \dfrac{\min(stdx, stdy)}{S_{\text{large}}\ stdz} \\ z_i \leftarrow S\ z_i \end{cases} \text{ where } S_{\text{large}} \approx 100 \qquad [4.26]$$

Clearly, this scaling can only be used when the features of interest in the bunch are 'pancake'. If a 'hotspot' near a cathode has a radius of one-tenth of the radius of the bunch, the S_{large} parameter needs to be in the order of 1000 to allow scaling.

Scaling is also possible for very low aspect ratio bunches (A<0.01). These 'cigar'-shaped bunches have equal transverse fields as long as the charge density remains constant. The longitudinal fields, except at the very edges, however scale inverse linear with the length. Therefore, the scaling recipe becomes:

$$\text{If } \begin{cases} stdz > stdx/S_{small} \\ stdz > stdy/S_{small} \end{cases} \Rightarrow \begin{cases} S = \dfrac{\max(stdx, stdy)}{S_{small}\ stdz} \\ z_i \leftarrow S\ z_i \\ E_x \leftarrow S\ E_x \\ E_y \leftarrow S\ E_y \\ E_z \leftarrow S^2\ E_z \end{cases} \text{ where } S_{\text{small}} \approx 0.01 \qquad [4.27]$$

The parameters S_{small} and S_{large} can be set with the **"BeamScale"** option.

4.5.1.11 Transform to rest frame

To obtain the actual electromagnetic 3D space-charge fields, the electric field $\mathbf{E}'_i$ is transformed back to the laboratory frame. This is accomplished with the following Lorentz transformation:

$$\begin{aligned} \mathbf{E}_i &= \gamma_0 \left[\mathbf{E}'_i - \frac{\gamma_0}{\gamma_0 + 1} (\boldsymbol{\beta}_0 \cdot \mathbf{E}'_i) \boldsymbol{\beta}_0 \right] \\ \mathbf{B}_i &= \frac{\gamma_0 \boldsymbol{\beta}_0 \times \mathbf{E}'_i}{c} \end{aligned} \qquad [4.28]$$

4.5.2 spacecharge3Dtree

spacecharge3Dtree(theta) ;
3D space-charge routine based on the Barnes-Hut hierarchical tree algorithm [10,11].

theta Overall accuracy parameter. The typical range is from 0.3 (publication quality) to 1.0 (quick and dirty).

The main use of **spacecharge3Dtree** is to calculate stochastic/granularity effects such as disorder-induced heating and Boersh effect efficiently: **spacecharge3Dtree** covers all Coulomb interactions from first principles, and therefore includes both space charge and stochastic effects.

The fields of relativistic bunches are calculated in a co-moving frame, fully analogous to the **spcecharge3Dmesh** element. This element is however substantially slower compared to the PIC method employed in **spacecharge3Dmesh**. Consequently, it is wise to use this element only when granularity effects are important.

A Barnes-Hut algorithm is used to efficiently compute the fields of many interacting particles. This is an $O(N \log N)$-scheme based on hierarchical grouping of distant particles. The interaction between near neighbours is calculated individually, but distant charges are grouped together into a single-charge or multipole expression, leading to substantial savings in the number of interactions necessary to calculate the fields compared to the point-to-point method employed in **spacecharge3D**. Only in the case of excessive energy spread, when there is no co-moving frame with all velocities far below the speed of light, the fully relativistic **spacecharge3D** element should be considdered.

To avoid field-discontinuities that slow down the integration of the equations of motion, each particle is represented as a Plummer sphere with smoothing length **R**. The current implementation requires this radius to be identical for all particles. It can be set with the **setrmacrodist** keyword as described in section 4.4.1.11:

```
setrmacrodist(set,"u",R,0) ;
```

where **set** is the particle set, for example **"beam".**

If each particle in the simulation represents one elementary particle, the smoothing length **R** must be chosen much smaller than the typical distance between the particles. In the case where each particle represents a large number of elementary particles, **R** is a tradeoff between unphysical disorder-induced heating and underestimation of space-charge forces. The 'correct' value for this case is in the order of the typical distance between the macro-particles. However, in that case we strongly recommend benchmarking against the PIC model of **spacecharge3Dmesh**.

The charge density, electrostatic field and potential of the particles are respectively given by:

$$\rho(r) = \frac{q}{4\pi}\frac{3R^2}{\left(r^2+R^2\right)^{5/2}}$$

$$\mathbf{E}_r(r) = \frac{q}{4\pi\varepsilon_0}\frac{\mathbf{r}}{\left(r^2+R^2\right)^{3/2}}$$

$$V(r) = \frac{q}{4\pi\varepsilon_0}\frac{1}{\sqrt{r^2+R^2}}$$

The integrated charge density equals the total charge q. The limiting cases for $r \gg R$ are the well-known equations for point charges:

$$\mathbf{E}_r(r) = \frac{q}{4\pi\varepsilon_0}\frac{\mathbf{r}}{r^3}$$

$$V(r) = \frac{q}{4\pi\varepsilon_0 r}$$

The algorithm calculates near-neighbour interactions individually, and all other particles are grouped in increasingly larger clusters as function of distance. The dimensionless parameter **theta** sets the accuracy of the calculations by indirectly defining the number of particles that are grouped together. It is defined as the threshold for the ratio between the size of a distant cluster and its distance. For a detailed explanation of **theta**, see [12]. A smaller value for **theta** results in a higher accuracy at the expense of CPU time. Typical values are in the range 0.3 to 2.0, with a recommended range between 0.5 and 1.0. In the extreme case **theta**=0, the algorithm does not use any grouping and calculates all N^2 interactions individually.

Quadrupole moments of the charge distribution are taken into account in the calculation as it increases the accuracy at fairly modest computational costs. If the particle distribution contains both negative and positive charge also the dipole moment is taken into account.

4.5.2.1 Lorentz transformations for relativistic bunches

All particle interactions are calculated in the rest frame of the bunch, fully analogous to the **spacecharge3Dmesh** method. The rest frame is defined as the frame with zero total momentum. It has a velocity $\boldsymbol{\beta}_0$ and accompanying Lorentz factor γ_0 of respectively:

$$\boldsymbol{\beta}_0 = \frac{\overline{\gamma\boldsymbol{\beta}}}{\bar{\gamma}} = \frac{\sum n_i \gamma_i \boldsymbol{\beta}_i}{\sum n_i \gamma_i}$$

$$\gamma_0 = 1/\sqrt{1-\beta_0^2}$$

The particle coordinates $\mathbf{r}_i$ are different in the rest frame, denoted with a prime. They are a factor γ_0 expanded in the direction of the velocity of the frame:

$$\begin{cases} \mathbf{r}_i{}'_{\perp} = \mathbf{r}_{i\perp} \\ \mathbf{r}_i{}'_{//} = \gamma_0 \mathbf{r}_{i//} \end{cases} \Rightarrow \mathbf{r}_i{}' = \mathbf{r}_i + \frac{\mathbf{r}_i \cdot \gamma_0 \boldsymbol{\beta}_0}{\gamma_0 + 1}\,\gamma_0 \boldsymbol{\beta}_0$$

Ideally, all velocities in the rest frame are well below the speed of light. A warning is printed if the Lorentz factor of a particle *as measured in the rest frame* exceeds the value of 2. The calculated electric field $\mathbf{E}'_i$ is Lorentz transformed back to the laboratory frame to get the required **E** and **B** fields for the tracking:

$$\mathbf{E}_i = \gamma_0 \left[\mathbf{E}'_i - \frac{\gamma_0}{\gamma_0 + 1} (\boldsymbol{\beta}_0 \cdot \mathbf{E}'_i) \boldsymbol{\beta}_0 \right]$$
$$\mathbf{B}_i = \frac{\gamma_0 \boldsymbol{\beta}_0 \times \mathbf{E}'_i}{c}$$

4.5.3 Spacecharge3D

`spacecharge3D([zmin,zmax]) ;`
Instructs GPT to take 3D space-charge effects into account using a point-to-point method.

`zmin` *z*-position [m] from where to calculate the space-charge fields. If not specified, –∞ is assumed.

`zmax` *z*-position [m] unto where to calculate the space-charge forces. If not specified, ∞ is assumed.

The fields generated by the **`spacecharge3D`** routine are calculated directly from relativistic particle-particle interaction. For each particle-particle contribution both particles have to be within the range **`zmin`** < z < **`zmax`**. Retardation and radiation effects are not taken into account. The advantage of this method is that no approximations are made; the disadvantage is the price in terms of CPU time. Particle-particle interactions are N^2 processes, where N is the number of traced macro-particles. In our experience however relatively few particles already give good statistics because, apart from discretisation, no approximations are made. A few hundred particles are usually sufficient.

To calculate the fields generated by particle *j* at the position of particle *i*, first both particle coordinates are transformed to the rest frame of particle *j*. Due to Lorentz contraction, the distance between the two particles thereby changes:

$$\mathbf{r}_{ji} = \mathbf{r}_i - \mathbf{r}_j$$
$$\mathbf{r}'_{ji} = \mathbf{r}_{ji} + \frac{\gamma_j^{\,2}}{\gamma_j + 1} \left(\mathbf{r}_{ji} \cdot \boldsymbol{\beta}_j \right) \boldsymbol{\beta}_j \qquad [4.29]$$

where $\mathbf{r}_{ji}$ is the distance measured in the lab frame and $\mathbf{r}'_{ji}$ is the distance in the rest frame.

Within the rest frame of particle *j*, only an electric field is present. This coulomb field is given by

$$\mathbf{E}'_{j \to i} = \frac{1}{4\pi\varepsilon_0} \frac{Q \mathbf{r}'_{ji}}{\left(\mathbf{r}'^2_{ji} + R^2 \right)^{3/2}} \qquad [4.30]$$

where R is the Plummer sphere radius. This extra term in the denominator is to prevent rare divisions by zero, and it slightly increases tracking speed. It can be set with the keyword **`setrmacrodist`** described in section 4.4.1.11. To prevent an unrealistic drop in field amplitude, the radius R should be set smaller than the typical distance between two particles.

Transforming this electric field back to the lab frame and summing over all particles yields the electromagnetic fields at the position of particle *i*:

$$\mathbf{E}_i = \sum_{j \neq i} \gamma_j \left[\mathbf{E}'_{j \to i} - \frac{\gamma_j}{\gamma_j + 1} \left(\boldsymbol{\beta}_j \cdot \mathbf{E}'_{j \to i} \right) \boldsymbol{\beta}_j \right] \qquad [4.31]$$
$$\mathbf{B}_i = \sum_{j \neq i} \frac{\gamma_j \boldsymbol{\beta}_j \times \mathbf{E}'_{j \to i}}{c}$$

4.5.4 Spacecharge3Dclassic

`spacecharge3Dclassic([zmin,zmax]) ;`
Instructs GPT to take 3D non-relativistic space-charge effects into account using a point-to-point method.

`zmin` *z*-position [m] from where to calculate the space-charge fields. If not specified, –∞ is assumed.

zmax *z*-position [m] unto where to calculate the space-charge forces. If not specified, ∞ is assumed.

The fields generated by the **spacecharge3Dclassic** routine are calculated directly from non-relativistic particle-particle interaction. For each particle-particle contribution both particles have to be within the range **zmin** < z < **zmax**. Retardation, relativistic and radiation effects are not taken into account.

The coulomb field generated by particle *j* at the position of particle *i* is given by

$$\mathbf{r}_{ji} = \mathbf{r}_i - \mathbf{r}_j$$
$$\mathbf{E}_{j\to i} = \frac{1}{4\pi\varepsilon_0}\frac{Q\,\mathbf{r}_{ji}}{\left(\mathbf{r}_{ji}^2 + R^2\right)^{3/2}} \qquad [4.32]$$

where *R* is the Plummer sphere radius. This extra term in the denominator is to prevent rare divisions by zero, and it slightly increases tracking speed. It can be set with the keyword **setrmacrodist** described in section 4.4.1.11. To prevent an unrealistic drop in field amplitude, the radius *R* should be set smaller than the typical distance between two particles.

4.5.5 SetTotalCharge

settotalcharge(set,Q) ;
Sets the total charge of all particles in a set.

set Particle set.
Q Total charge [C] of the set.

This element can be used to specify the total charge of a set, analogue to the **Q** parameter of the **setparticles** function. It can only be used after all particles are added to the specified **set**.

4.5.6 Spacecharge2Dcircle

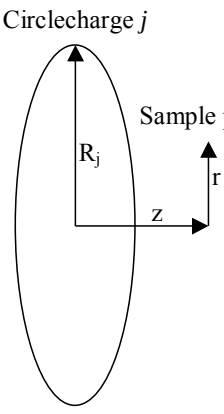
spacecharge2Dcircle([zmin,zmax]) ;
Calculates 2D space-charge fields using charged circles.

zmin *z*-position [m] from where to calculate the space-charge forces. If not specified, –∞ is assumed.
zmax *z*-position [m] upto where to calculate the space-charge forces. If not specified, ∞ is assumed.

The **spacecharge2Dcircle** element can be used to calculate the space-charge effect of cylinder-symmetric beams by representing every particle as a homogeneously charged circle. Because every particle represents a complete circle, the amount of particles needed to obtain correct statistics is much less when compared to the point-to-point calculations. The influence of the transverse velocity on the space-charge field is neglected.

For the calculation of the total fields working on particle *i* due to the space charge, first the parameters of the circle representing particle *j* are obtained. The radius R_j, charge density λ_j and center position $\mathbf{C}_j$ are defined by:

$$R_j = \sqrt{x_j^2 + y_j^2}$$
$$\lambda_j = Q_j / 2\pi R_j$$
$$\mathbf{C}_j = (0,0,z_j) \qquad [4.33]$$



The velocity of the circle is the longitudinal velocity of particle *j*. Therefore, the Lorentz factor γ_j of the circle must be:

$$\gamma_j = 1/\sqrt{1-\beta_{z,j}^2} \qquad [4.34]$$

In the rest frame of the circle, there is only an electrostatic field2F[3]. This field is derived from the electrostatic potential (in polar coordinates):

$$V'_j(r,\theta) = \frac{\lambda_j R}{4\pi\varepsilon_0}\int_0^{2\pi}\frac{1}{\sqrt{R^2+r^2-2Rr\sin(\theta)\cos(\phi)}}d\phi = \frac{Q_j}{2\pi^2\varepsilon_0}\frac{K\left(\frac{4Rr\sin(\theta)}{a^2+r^2+2Rr\sin(\theta)}\right)}{\sqrt{R^2+r^2+2Rr\sin(\theta)}} \quad [4.35]$$

Using the definition3F[4]:

$$\mathrm{K}(k) = \int_0^{\pi/2}\frac{1}{\sqrt{1-k\sin(\theta)}}d\theta \quad [4.36]$$

The resulting electric field in the rest frame is calculated from $\mathbf{E}' = -\nabla V'$. Calculating the gradient and converting from polar to cylindrical coordinates yields 4F[5]:

$$E'_r = \frac{Q}{4\pi^2\varepsilon_0 r\sqrt{d^2+4Rr}}\left(K(\alpha) - \frac{R^2-r^2+z^2}{d^2}E(\alpha)\right)$$
$$E'_z = \frac{QzE(\alpha)}{2\pi^2\varepsilon_0 d^2\sqrt{d^2+4Rr}} \quad [4.37]$$

where $\mathrm{E}(k) = \int_0^{\pi/2}\sqrt{1-k\sin(\theta)}d\theta$, $d^2 = (R-r)^2+z^2$ and $\alpha = 4Rr/(d^2+4Rr)$. Due to Lorentz contraction in the laboratory frame, $z = \gamma_j(z_i - z_j)$. No relativistic effects occur in the transverse direction because the circle is assumed to have a longitudinal velocity component only.

Transforming this field back to the lab frame and adding the contributions of all particles *j* yields:

$$\begin{aligned} E_x &= \sum_{j\neq i}\gamma_j E'_{x,j} & B_x &= \sum_{j\neq i} -\gamma_j\beta_{z,j}E'_{y,j}/c \\ E_y &= \sum_{j\neq i}\gamma_j E'_{y,j} & B_y &= \sum_{j\neq i} +\gamma_j\beta_{z,j}E'_{x,j}/c \\ E_z &= \sum_{j\neq i}E'_{z,j} & B_z &= 0 \end{aligned} \quad [4.38]$$

Currently, it is not allowed to start particles at *x*=*y*=0 while using the **spacecharge2Dcircle** element. Because this element assumes cylindrical symmetry anyway, it is best to fill only the positive x-axis as shown in the following example.

Example: Start a 100 pC beam with a radius of 1 mm, optimally distributed for spacecharge2Dcircle. The optional **setphidist** can be specified after **setxdist** to fill the entire xy-plane.

```
17. nps  = 1000 ;      # Start 1000 particles
18. Qtot = -100e-12 ;  # Total charge 100 pC
19. R    = 1e-3 ;      # Beam radius of 1 mm
20. setparticles( "beam", nps, me, qe, Qtot ) ;
21. setxdist("beam","u",R/2,R) ;
22. setphidist("beam","u",0,2*pi) ;
23. setcharge2Dcircle("beam",Qtot) ;
24. spacecharge2Dcircle() ;
```

4.5.7 Setcharge2Dcircle

setcharge2Dcircle(set,Q) ;

Sets the total charge of a particle set for the rz space-charge models.

---

[3] Actually, a $\beta_{\varphi,j}$ term could be interpreted as a rotation of the circle, and a d*R*/d*t* term as expansion and contraction. Both terms result in a magnetic field in the rest frame of the circle, but their contributions are neglected here.

[4] Please note that the argument *k* of the elliptic functions is sometimes written as k^2 in the literature.

[5] Actually, we use the Carlson elliptic functions: $\mathrm{K}(k) = \mathrm{R_F}(1-k^2)$ and $\mathrm{E}(k) = \mathrm{R_F}(1-k^2) - \frac{1}{3}k^2\mathrm{R_D}(1-k^2)$ to avoid the singularity at *r*=0.

set	Particle set.
Q	Total charge of the particle set [C].

The number of elementary particles represented by each macro-particle, n_i, is calculated. In contrast to **setparticles** and **settotalcharge**, the n_i's are not constant but a linear function of the distance of the particle to the z-axis. The total charge of all the particles in the specified **set** sums to **Q**.

When a point-to-point space-charge model is used, the optimal initial particle distribution is uniform in the xy-cross section for a uniform beam. This results in more particles at a larger radius, each representing the same number of elementary particles. However, when an rz space-charge model like **spacecharge2Dcircle** is used, it is better to distribute the particles equidistantly in *r*. This implies that particles further off-axis should carry more charge, linear with this distance. Thus each particle should represent *n* elementary particles as specified by:

$$n_i = \frac{\mathbf{Q}\sqrt{x_i^2 + y_i^2}}{\sum_i q_i \sqrt{x_i^2 + y_i^2}} \qquad [4.39]$$

where q_i is the elementary charge of particle *i*. For a uniform distribution in *x* of identical particles, the equation reduces to:

$$n_i = \frac{2\sqrt{x_i^2 + y_i^2}\ \mathbf{Q}}{N\ R\ q} \qquad [4.40]$$

4.5.8 Spacecharge2Dline

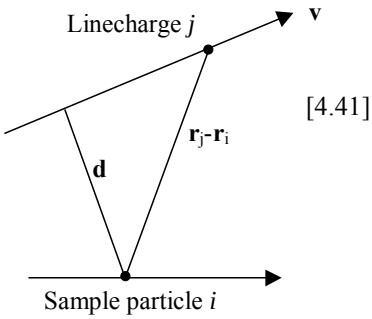
spacecharge2Dline([zmin,zmax]) ;
Calculates 2D space-charge fields using infinitely long line charges [9F13].

zmin	*z*-position [m] from where to calculate the space-charge forces. If not specified, –∞ is assumed.
zmax	*z*-position [m] unto where to calculate the space-charge forces. If not specified, ∞ is assumed.

The **spacecharge2Dline** element can be used to provide a quick estimate for space-charge forces in continuous beams or very long bunches. Every particle is represented as a moving line charge, directed in the particle's velocity. Because every particle represents a complete line, the amount of particles needed to obtain correct statistics is much less when compared to the point-to-point calculations in **spacecharge3D**.

To calculate the fields at the position of particle *i*, generated by the moving line charge representing particle *j*, first the distance **d** between the point *i* and the line *j* is calculated by:

$$\mathbf{d}_{ji} = (\mathbf{r}_j - \mathbf{r}_i) - \left[(\mathbf{r}_j - \mathbf{r}_i)\cdot\hat{\mathbf{v}}_j\right]\cdot\hat{\mathbf{v}}_j \qquad [4.41]$$



The total charge (**Qtot**) parameter of **setparticles** is interpreted as the beam current [A] when **spacecharge2Dline** is used. Therefore, each particle *j* represents a current i_j [A] and charge density λ_j [C/m] given respectively by:

$$i_j = \frac{\mathbf{Qtot}}{\mathbf{N}},\ \lambda_j = \frac{i_j}{v_j} \qquad [4.42]$$

The total electromagnetic field, at the position of particle *i*, is the sum of the contributions of the individual line charges *j*. The magnetic contribution is due to the current i_j, the electrical contribution due to the charge density λ_j.

$$\mathbf{E}_i = \sum_{j \neq i} \frac{i_j}{|v_j|} \frac{1}{2\pi \varepsilon_0 d_{ji}^2} \cdot -\mathbf{d}_{ji}$$
$$\mathbf{B}_i = \sum_{j \neq i} \frac{i_j}{|v_j|} \frac{\mu_0}{2\pi d_{ji}^2} \cdot (\mathbf{d}_{ji} \times \mathbf{v}) \qquad [4.43]$$

These equations are relativistically correct, even though no Lorentz transformation is applied. The transverse distribution is not restricted in any way.

When **`setrmacrodist`** is used, the line charges have a radius *R*. Within this radius, the equations reduce to:

$$\mathbf{E}_i = \sum_{j \neq i} \frac{i_j}{|v_j|} \frac{1}{2\pi \varepsilon_0 R^2} \cdot -\mathbf{d}_{ji}$$
$$\mathbf{B}_i = \sum_{j \neq i} \frac{i_j}{|v_j|} \frac{\mu_0}{2\pi R^2} \cdot (\mathbf{d}_{ji} \times \mathbf{v}) \qquad [4.44]$$

Because every particle represents a line, the simulation must be started with a thin disk of particles. When the longitudinal dimensions of the disk become too large due to energy spread, the accuracy of the model diminishes rapidly.

4.6 Static electric fields

This section contains the elements generating static electric fields.

4.6.1 Circlecharge

`circlecharge(ECS,R,lambda) ;`
Circle with homogeneous charge density.

`ECS`	Element Coordinate System.
`R`	Radius of the ring [m].
`lambda`	Charge density [C/m].

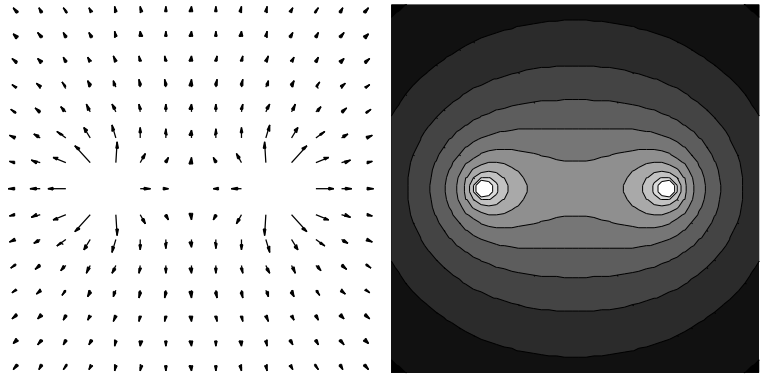


Figure 4–7: Electric field in the xz-plane and corresponding potential.

The circle with radius **`R`** having a homogeneous charge density **`lambda`** in [C/m] is located in the xy-plane centered around the origin. The potential of the electric field is given in spherical coordinates by:

$$V(r,\theta)=\frac{\lambda R}{4\pi\varepsilon_0}\int_0^{2\pi}\frac{1}{\sqrt{R^2+r^2-2Rr\sin(\theta)\cos(\phi)}}d\phi \qquad [4.45]$$

The electric field is calculated from $\mathbf{E}=-\nabla V$. The integral and gradient are calculated analytically.

4.6.2 Ecyl

`ecyl(ECS,R,L,E) ;`
Constant **E** field bounded by cylinder.

`ECS`	Element Coordinate System.
`R`	Radius of the cylinder [m].
`L`	Cylinder length [m].
`E`	Electric field strength [V/m].

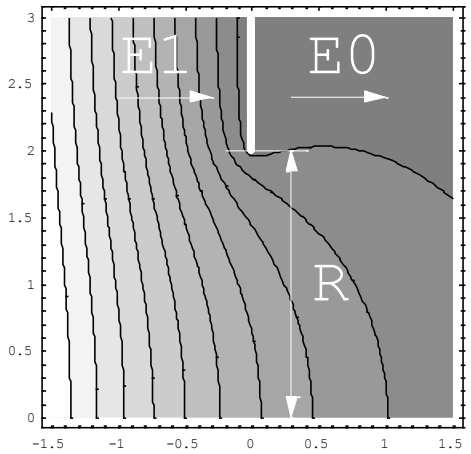
The **E** field generated by this element is given by:

$$\mathbf{E}=\begin{cases}\mathbf{E}\hat{z} & \text{if } x^2+y^2<\mathbf{R}^2 \text{ and } |z|<\frac{1}{2}\mathbf{L}\\ 0 & \text{otherwise}\end{cases} \qquad [4.46]$$

4.6.3 Ehole

ehole(ECS,R,E1,E0) ;
Two regions with uniform electric field, separated by a conducting plate with circular hole.

ECS	Element Coordinate System.
R	Radius of the hole.
E1	Uniform field strength at negative *z*.
E0	Uniform field strength at positive *z*.



This element models two regions of uniform electric field in the *z*-direction, separated by an infinitely thin conducting plate with a circular hole. The electromagnetic field is fully analytic, where the effect of the hole is given by [10F14]:

$$V(z,r)=\frac{(\mathbf{E0}-\mathbf{E1})\,\mathbf{R}}{\pi}\left(\sqrt{\frac{\rho-\lambda}{2}}-\frac{|z|}{\mathbf{R}}\arctan\left(\sqrt{\frac{2}{\rho+\lambda}}\right)\right) \qquad [4.47]$$

with $\lambda=\left(z^2+r^2\right)/\mathbf{R}^2-1$ and $\rho=\sqrt{\lambda^2+4\,z^2/\mathbf{R}^2}$. In case **R** is specified as zero, this element reduces to two regions of uniform electric field in the *z*-direction.

To remove particles crossing the plate, **ehole** can be combined with **scatteriris** as shown below:

```
; Plate at z=1 m, with 1 mm hole.
zplate = 1 ;
radius = 1e-3 ;
ehole("wcs","z",zplate, radius, E1, E0) ;
; Remove particles crossing the plate
forwardscatter("wcs","I","remove", 0) ;
scatteriris("wcs","z",zplate, radius,1 ) scatter="remove" ;
```

Example: Einzel lens
The **ehole** element is not particularly useful itself, as it stretches from –∞ to +∞. However, it can be used as building block to construct all kinds of electrostatic focusing systems. For example an Einzel lens can easily be constructed by combining three **ehole** elements:

```
zcenter = 1 ;
separation = 0.1 ;
radius = 10e-3
ehole("wcs","z",zcenter-separation, radius, -Efield/2, Efield/2 ) ;
ehole("wcs","z",zcenter           , radius,  Efield  ,-Efield ) ;
ehole("wcs","z",zcenter+separation, radius, -Efield/2, Efield/2 ) ;
rmax("wcs","z",zcenter, radius,2*separation) ;
```

Note: The above summation of three **ehole** elements is only correct if **radius** is much smaller than **separation**. If this assumption is violated an external Poisson solver, such as present in the Superfish set of codes, must be used to obtain the electrostatic fields. The resulting field can subsequently be imported into GPT with the **map2D_E** element.

4.6.4 Erect

erect(ECS,a,b,L,E) ;
Constant **E** field bounded by a rectangular box.

ECS	Element Coordinate System.
a	Total box length in x-direction [m].
b	Total box length in y-direction [m].
L	Total box length in z-direction [m].
E	Electric field strength [V/m].

The **E** field generated by this element is given by:

$$\mathbf{E} = \begin{cases} \mathbf{E}\hat{z} & \text{if } |x| < \tfrac{1}{2}\mathbf{a}, |y| < \tfrac{1}{2}\mathbf{b}, |z| < \tfrac{1}{2}\mathbf{L} \\ 0 & \text{otherwise} \end{cases} \qquad [4.48]$$

4.6.5 Linecharge

linecharge(ECS,L,lambda) ;
Line with homogeneous charge density.

ECS	Element Coordinate System.
L	Total line length [m].
lambda	Line charge density [C/m].

The equation for the line is: $x = y = 0$, $|z| < \tfrac{1}{2}\mathbf{L}$

The electric field generated by this element is given by:

$$\mathbf{E} = -\operatorname{grad}\left(\frac{\lambda}{4\pi\varepsilon_0} \int_{-\mathbf{L}/2}^{\mathbf{L}/2} \frac{1}{\sqrt{x^2 + y^2 + (z - z')^2}}\, dz' \right) \qquad [4.49]$$

where the integral and gradient are calculated analytically.

4.6.6 Platecharge

platecharge(ECS,a,b,sigma) ;
Rectangular plate with homogeneous charge density.

ECS	Element Coordinate System.
a	Total plate length in x-direction [m].
b	Total plate length in y-direction [m].
sigma	Surface charge density [C/m^2].

The equation for the plate is: $z = 0$, $|x| < \tfrac{1}{2}\mathbf{a}$, $|y| < \tfrac{1}{2}\mathbf{b}$

The electric field generated by this element is given by:

$$\mathbf{E} = -\operatorname{grad}\left(\frac{\sigma}{4\pi\varepsilon_0} \int_{-\mathbf{a}/2}^{\mathbf{a}/2} \int_{-\mathbf{b}/2}^{\mathbf{b}/2} \frac{1}{\sqrt{(x - x')^2 + (y - y')^2 + z^2}}\, dy'\, dx' \right) \qquad [4.50]$$

where the integral and gradient are calculated analytically.

4.6.7 Pointcharge

pointcharge(ECS,Q) ;
Pointcharge at the origin.

ECS	Element Coordinate System.
Q	Charge in Coulomb.

The electric field generated by this element in spherical coordinates is given by:

$$\mathbf{E} = \frac{Q}{4\pi\varepsilon_0}\frac{\mathbf{r}}{|\mathbf{r}^3|} \qquad [4.51]$$

4.6.8 pointchargeset

`pointchargeset(set,theta) ;`
Electrostatic field of a point charge distribution, calculated with the Barnes-Hut tree algorithm [15,16].

`set`	Particle set
`theta`	Overall accuracy parameter. The typical range is from 0.3 (publication quality) to 1.0 (quick and dirty).

The main use of **`pointchargeset`** is to specify a fixed particle set, such as an ion background while tracking electrons. All particles in the specified set are removed from the tracking and internally stored as a fixed electrostatic field.

For efficiency reasons, the Barnes&Hut hierarchical tree method is used to calculate the combined field. This method allows a tradeoff between speed and accuracy, governed by the **`theta`** parameter. Please see the analogous element **`spacecharge3Dtree`** in section 4.5.2 for a more detailed description.

To avoid singularities, it is important to set the smoothing length **`R`** of the particles. The current implementation only allows identical radii:
`setrmacrodist(set,"u",R,0)`

4.7 Static magnetic fields

This section contains elements that model static magnetic fields.

The elements **magpoint**, **magline** and **magplate** calculate a **B** field as if magnetic charge were present. They can be convenient in a simulation, but should be used with caution because magnetic charge does not exist. The **B** field generated by a monopole is assumed to be [11F17]:

$$\mathbf{B} = \frac{\mu_0 P}{4\pi} \frac{\mathbf{r}}{|\mathbf{r}^3|} \qquad [4.52]$$

with P the magnetic pole strength. The fields of the magnetic line charge λ_m and surface charge σ_m are derived analogue to the electrical case.

The magnetic scalar potential Φ is defined such that **B**=–grad(Φ) analogous to the electrical case.

4.7.1 Barmagnet

barmagnet(ECS,a,b,L,B) ;

Rectangular magnet modeled as two plates with opposite magnetic charge density.

ECS	Element Coordinate System.
a	Magnet length in x-direction [m].
b	Magnet length in y-direction [m].
L	Magnet length in z-direction [m].
B	Nominal magnetic field [T].

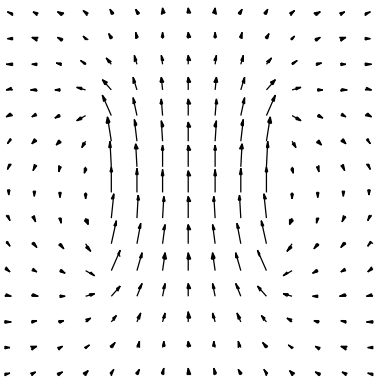


Figure 4–8: Typical **B** field in the x-z- and y-z-plane.

The **B** field generated by this element can be written as the field of two identical plates with opposite magnetic surface charge density. The plates are located at:
$|x| < \frac{1}{2}\mathbf{a}$, $|y| < \frac{1}{2}\mathbf{b}$, $z = \pm\frac{1}{2}\mathbf{L}$.

with corresponding surface charge density:

$$\sigma_m = \pm \frac{\mathbf{B}}{\mu_0} \qquad [4.53]$$

Within the magnet {0,0,**B**} is added to the magnetic field to convert the **H** field to a **B** field.

For the magnetic field generated by a rectangular plate with homogeneous magnetic surface charge density see **magplate**, section 4.7.5.

4.7.2 Bzsolenoid

bzsolenoid(ECS,R,L,nI) ;
A homogeneous solenoid without thickness approximated as circular current sheet.

ECS	Element Coordinate System.
R	Radius of the solenoid [m].
L	Solenoid length [m].
nI	Ampere turns per meter [A].

The fields generated by this element in cylindrical coordinates are:

$$\begin{aligned} B_z &= \frac{1}{2}\mu_0 \left[\frac{z+\frac{1}{2}\mathbf{L}}{\left\{\left(z+\frac{1}{2}\mathbf{L}\right)^2+\mathbf{R}^2\right\}^{\frac{1}{2}}} - \frac{z-\frac{1}{2}\mathbf{L}}{\left\{\left(z-\frac{1}{2}\mathbf{L}\right)^2+\mathbf{R}^2\right\}^{\frac{1}{2}}} \right] \mathbf{nI} \\ B_r &= -\frac{r}{4}\mu_0 \left[\frac{1}{\left\{\left(z+\frac{1}{2}\mathbf{L}\right)^2+\mathbf{R}^2\right\}^{\frac{3}{2}}} - \frac{1}{\left\{\left(z-\frac{1}{2}\mathbf{L}\right)^2+\mathbf{R}^2\right\}^{\frac{3}{2}}} \right] \mathbf{nIR^2} \\ B_\phi &= 0 \end{aligned} \qquad [4.54]$$

The fields generated are only correct near the **z**-axis of the element.

4.7.3 Linecurrent

linecurrent(ECS, x1,y1,z1, x2,y2,z2, I) ;
Straight line segment.

ECS	Element Coordinate System.
x1,y1,z1	Start position of the line [m].
x2,y2,z2	End position of the line [m].
I	Current through the line [A].

The line runs from (x_1,y_1,z_1) to (x_2,y_2,z_2). The current flows from the starting point of the line to the endpoint.

The field of this element is calculated in a frame in which the **z**-direction is parallel to the line segment and the origin is chosen to be the center point of the line. In this new coordinate system, the magnetic field generated by this element is given by:

$$\begin{aligned} \mathbf{B} &= \nabla \times \frac{\mu_0 \mathbf{I}}{\mathbf{4}\pi} \left(0,0, \int_{z-L/2}^{z+L/2} \frac{1}{\sqrt{x^2+y^2+z^2}}\, dz \right) \\ L &= \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2} \end{aligned} \qquad [4.55]$$

where the integral and curl are calculated analytically.

4.7.4 Magline

magline(ECS,L,lambda) ;
Line with homogeneous magnetic charge density.

ECS	Element Coordinate System.

L	Total line length [m].
lambda	Magnetic line charge density [A].

The equation for the line is: $x = y = 0$, $|z| < \frac{1}{2}\mathbf{L}$

The magnetic field generated by this element is given by:

$$\mathbf{B} = -\text{grad}\left(\frac{\mu_0 \lambda}{4\pi} \int_{-\mathbf{L}/2}^{\mathbf{L}/2} \frac{1}{\sqrt{x^2 + y^2 + (z - z')^2}} dz' \right) \quad [4.56]$$

where the integral and gradient are calculated analytically.

4.7.5 Magplate

magplate(ECS,a,b,sigma) ;
Rectangular plate with homogeneous magnetic charge density.

ECS	Element Coordinate System.
a	Total plate length in x-direction [m].
b	Total plate length in y-direction [m].
sigma	Magnetic surface charge density [A/m].

The equation for the plate is: $z = 0$, $|x| < \frac{1}{2}\mathbf{a}$, $|y| < \frac{1}{2}\mathbf{b}$

The magnetic field generated by this element is given by:

$$\mathbf{B} = -\text{grad}\left(\frac{\mu_0 \sigma}{4\pi} \int_{-\mathbf{a}/2}^{\mathbf{a}/2} \int_{-\mathbf{b}/2}^{\mathbf{b}/2} \frac{1}{\sqrt{(x - x')^2 + (y - y')^2 + z^2}} dy'\, dx' \right) \quad [4.57]$$

where the integrals and gradient are calculated analytically.

4.7.6 Magdipole

magdipole(ECS, m) ;
Magnetic mathematical dipole.

ECS	Element Coordinate System.
m	Magnetic dipole moment [A m^2].

This element models a mathematical magnetic dipole, located at the origin and aligned with the z-axis. The magnetostatic field is given by:

$$B = \left(\frac{\mu_0 \, \mathbf{m}}{4\pi r^3} \right) \{ 3(\hat{\mathbf{m}} \cdot \hat{r}) \hat{r} - \hat{\mathbf{m}} \} \quad [4.58]$$

4.7.7 Magpoint

magpoint(ECS,P) ;
Magnetic pointcharge at the origin.

ECS	Element Coordinate System
P	Magnetic charge [Am].

The magnetic field generated by this element in spherical coordinates is given by:

$$\mathbf{B} = \frac{\mu_0 \mathbf{P}}{4\pi} \frac{\mathbf{r}}{|\mathbf{r}^3|} \quad [4.59]$$

4.7.8 Quadrupole

quadrupole(ECS,L,G) ;
Quadrupole lens without fringe fields.

ECS	Element Coordinate System.
L	Length of the quadrupole [m]
G	Magnetic field gradient in [T/m]. Usually B_{pool}/radius.

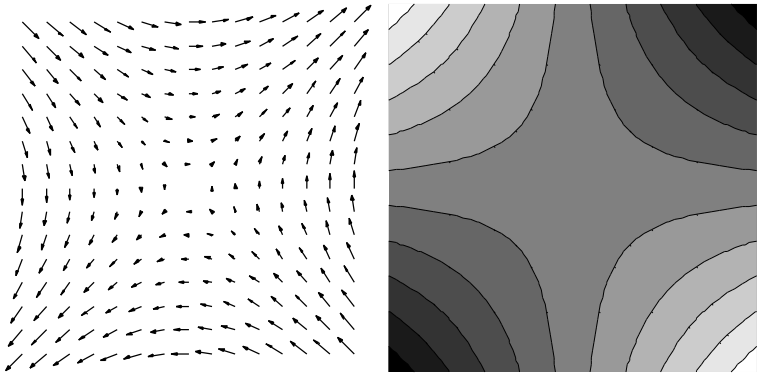


Figure 4–9: Magnetic field in the xy-plane and corresponding scalar potential.

The **B** field and scalar potential generated by this element are given by [18, pp. 39]:

$$\mathbf{B} = \begin{cases} (\mathbf{G}y, \mathbf{G}x, 0) & \text{if } |z| < \frac{1}{2}\mathbf{L} \\ 0 & \text{otherwise} \end{cases} \qquad [4.60]$$
$$\Phi = -Gxy$$

4.7.9 Rectcoil

rectcoil(ECS,r1,r2,L,I) ;
Homogeneous solenoid with length and thickness.

ECS	Element Coordinate System.
r1	Inner radius of the coil [m].
r2	Outer radius of the coil [m].
L	Length of the coil [m].
I	Total current through the coil [A].

To calculate the field on-axis the field of a single current loop is integrated in both the *z*- and *r*-direction:

$$B(z,r) = \int_{\mathbf{r1}}^{\mathbf{r2}} \int_{z-a}^{z+a} \frac{\mu_0 I r^2}{2(r^2+z^2)^{3/2}} dz dr \qquad [4.61]$$

where a=**L**/2.

The field off-axis is approximated using a 4th-order power series expansion [13F19].

$$B_z(z,r) = B(z) - \frac{1}{4}B''(z)r^2 + \frac{1}{64}B''''(z)r^4$$
$$B_r(z,r) = -\frac{1}{2}B'(z)r + \frac{1}{16}B'''(z)r^3 \qquad [4.62]$$

The fields are only correct near the **z**-axis of the element.

4.7.10 Rectmagnet

rectmagnet(ECS,a,b,Bfield,dl,b1,b2);
Rectangular magnet with fringe fields.

ECS	Element Coordinate System
a	Length of the magnet in *x*-direction [m].
b	Length of the magnet in *z*-direction [m].
Bfield	Maximum magnetic field [T].

dl	Fringe field offset for the magnet's entry and exit planes [m] to change the effective length.
b1	First coefficient of the fringe fields Enge function [m^{-1}]. A value of 0 disables fringe fields.
b2	Second order coefficient of the fringe fields Enge function [m^{-2}]. Typically 0.

This element models a rectangular magnet with a magnetic field-component **Bfield** in the *y*-direction in the region $-½\mathbf{a}<x<½\mathbf{a}$ and $-½\mathbf{b}<z<-½\mathbf{b}$, see Figure 4–10.

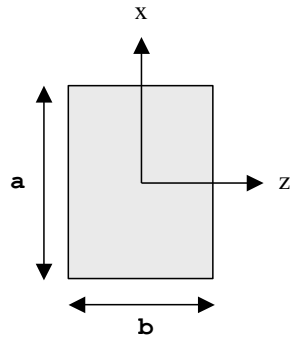


Figure 4–10: Geometry of rectmagnet.

The entrance and exit sides, assuming the *z*-direction as propagation direction of the particles, have fringe fields analogous to the fringe fields of the **sectormagnet** element. A reasonable estimate for $\mathbf{b_1}$ is $2/gap$, and we recommend a zero value for both **dl** and $\mathbf{b_2}$. Please see the documentation of the **sectormagnet** element for a detailed description of the fringe fields.

4.7.11 Sectormagnet

sectormagnet(fromCCS,toCCS,R,Bfield,[phiin,phiout,dl,b1,b2]);
Sector magnet with options for rotated pole faces and fringe fields.

fromCCS	Name of the CCS the particles start on.
toCCS	Name of the CCS the particles are bended to.
R	Radius of the bend [m].
Bfield	Maximum magnetic field [T]. Normally this is $\pm\mathbf{m}c\gamma\beta/\mathbf{qR}$.
phiin	Angle of incident pole face [rad]. Specify 0 for normal incidence.
phiout	Angle of exit pole face [rad]. Specify 0 for normal incidence.
dl	Fringe field offset for the magnet's entry and exit planes [m] to change the effective length.
b1	First coefficient of the fringe fields Enge function [m^{-1}]. A value of 0 disables fringe fields.
b2	Second order coefficient of the fringe fields Enge function [m^{-2}]. Typically 0.

This element models a bending magnet with a **B** field perpendicular to the **z**-directions of both the **fromCCS** and **toCCS** coordinate systems. The geometry is that of a wedge with the two straight sides perpendicular to the **z**-directions of the specified coordinate systems. When the **z**-axes do not intersect, or when they are parallel, this element can not be used.

The optional **phiin** and **phiout** parameters can be used to specify the angle (with respect to normal incidence) of the input and output pole faces in the bending plane. A positive angle means that the total angle of the 'wedge' will decrease. This is shown schematically in Figure 4–11.

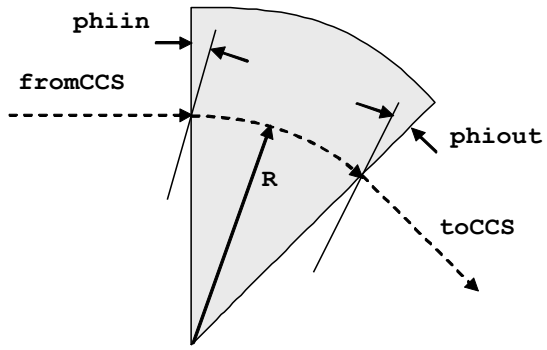


Figure 4–11: Geometry of the sectormagnet, with positive values for **phiin** and **phiout**.

The fringe fields are modeled as a second order Enge function, given on-axis by [14F20]

$$B_y(z, y=0) = \frac{1}{1+\exp\left(\mathbf{b_1}\ (z-\mathbf{dl}) + \mathbf{b_2}\ (z-\mathbf{dl})^2\right)} \qquad [4.63]$$

in the coordinate system shown in Figure 4–12. The parameter **dl** changes the effective length of the magnet, while $\mathbf{b_1}$ affects the steepness of the drop-off of the magnetic field. The second order correction, given by parameter $\mathbf{b_2}$, is implemented but should be used with care as it can reverse the field drop-off at a distance far away from the magnet.

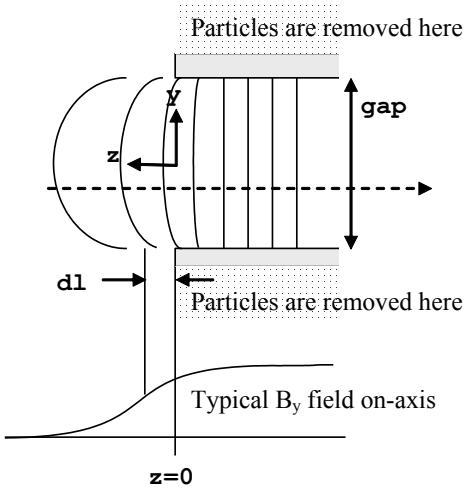


Figure 4–12: Fringe fields of the sectormagnet.

GPT removes all particles outside the gap region from the simulation. This is at a distance $|y| \geq \pi / \mathbf{b_1}$.

There are a number of ways to obtain the parameters **dl**, $\mathbf{b_1}$ and $\mathbf{b_2}$. Arguably the best method is to fit equation (1) to a measured field-profile of the magnet to be simulated. Second best is to use the 'field integral' FINT, such as used in MAD, and given by:

$$\frac{1}{\mathbf{b}_1} = g \cdot \mathrm{FINT} = \int \frac{B_y(s)\left(B_0 - B_y(s)\right)}{B_0^2} ds \qquad [4.64]$$

where B_0 is the nominal field inside the magnet in [T] and g the gap of the dipole in [m]. If everything else fails, a reasonable average value for $\mathbf{b_1}$ is $2/gap$. As mentioned before, we recommend a zero value for $\mathbf{b_2}$.

The normal use of the **sectormagnet** element is that the specified **radius** of the bend is the 'effective radius', based on the magnetic properties of the magnet. In that case, the **dl** parameter is (almost) zero and in any case far smaller than the size of the gap. To ease interpretation of the simulation results, **dl** can be used to fine-tune the magnet such that the particles exist the magnet exactly on-axis.
If on the other hand the **radius** parameter is based on the physical dimensions of the magnet, than the **dl** parameter should be used to specify how far the fringe fields extend out of the magnet. This is typically in the order of the size of the gap. In that case, the correct value for **dl** is approximated by:

$$\mathbf{dl} = \int \frac{B_y(s)}{B_0} - \mathrm{H}(s)\, ds \qquad [4.65]$$

where H(s) is the Heaviside unit step function.

To disable fringe-field calculations, both **b** parameters must be specified as 0. This is not recommended, as it slows down simulation speed considerably due to the resulting discontinuous field, especially in combination with space-chare calculations.

The full expression for the magnetic field used in this element are given by [15F21].

$$B_y = \frac{B\left(1+e^f\cos(h)\right)}{1+2e^f\cos(h)+e^{2f}} \qquad B_z = -\frac{B\left(e^f\sin(h)\right)}{1+2e^f\cos(h)+e^{2f}} \qquad [4.66]$$

with $f = \mathbf{b_1}\,z + \mathbf{b_2}(z^2-y^2)$ and $h = y(\mathbf{b_1}+2\mathbf{b_2}\,z)$.

We realize that fringe fields exist in all kinds of shapes and forms, and that these expressions model only a limited number of actual fields. On the other hand, it is our experience that the exact shape of the fringe fields does in many cases not affect the overall simulation results significantly. If needed, accurate modeling of all different shapes is possible with the **map3D_B** element in combination with an external field solver. In that case, the **CCSflip** element might be needed to 'manually' move the particles from one coordinate system to the other.

Advanced note: In reality half a bending magnet is constructed on the **fromCCS** and the other half on the **toCCS**. When a particle leaves one half of the bending magnet it is placed on the other coordinate system to go through the remaining part of the bend. Both half magnets are local elements and are not allowed to overlap other local elements.

4.7.12 Sextupole

sextupole(ECS,L,G) ;
Sextupole lens.

ECS	Element Coordinate System.
L	Length of the sextupole [m].
G	Magnetic field gradient in [T/m^2]. Usually B_{pool}/radius^2.

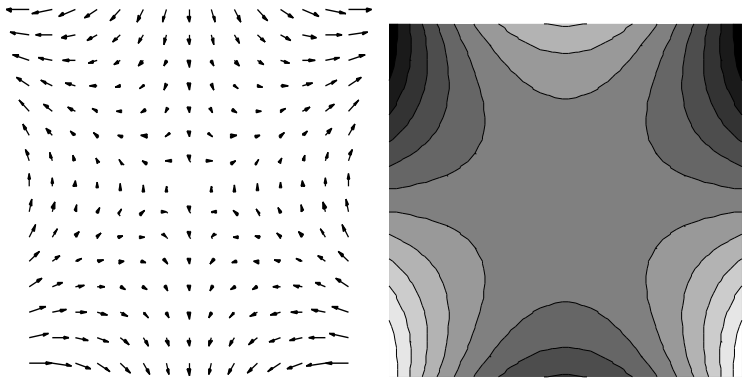


Figure 4–13: Magnetic field in the xy-plane and corresponding scalar potential.

The **B** field and scalar potential generated by this element are given by [18, pp. 158]:

$$\mathbf{B} = \begin{cases} \mathbf{G}\begin{pmatrix} 2xy \\ x^2-y^2 \\ 0 \end{pmatrix} & \text{if } |z| < \tfrac{1}{2}\mathbf{L} \\ \mathbf{0} & \text{otherwise} \end{cases} \qquad [4.67]$$

$$\Phi = \frac{G}{3}\left(y^3 - 3x^2y\right)$$

4.7.13 Solenoid

`solenoid(ECS,R,I) ;`
Single turn solenoid.

`ECS`	Element Coordinate System.
`R`	Radius of the solenoid [m]
`I`	Current through the solenoid [A].

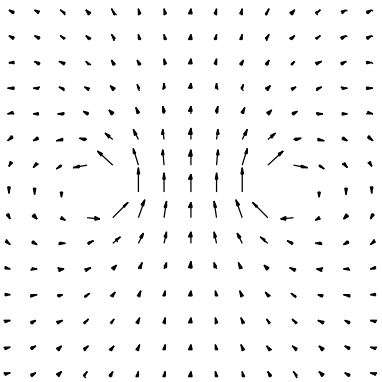


Figure 4–14: Magnetic field in the xz-plane.

The solenoid is modeled as a circle centered around the origin in the xy-plane with radius **`R`** carrying a current **`I`**. The resulting magnetic field is directed in the positive z-direction, within the current loop. The vector potential of the magnetic field is given in spherical coordinates by [16F22, pp. 117]:

$$A_\phi(r,\theta) = \frac{\mu_0 I R}{4\pi} \int_0^{2\pi} \frac{\cos(\phi)}{\sqrt{R^2 + r^2 - 2Rr\sin(\theta)\cos(\phi)}} d\phi \qquad [4.68]$$

The magnetic field is calculated from $\mathbf{B} = \nabla \times \mathbf{A}$. The integral and rotation are calculated analytically.

As the field is defined everywhere and no approximations are made, this element is very useful to study aberrations in solenoid lens systems.

4.8 Field maps

This section describes the elements capable of handling external field maps. These elements interpolate the electromagnetic fields and the electric potential on a rectangular grid. They are able to read GDF datafiles created by the utility programs ASCI2GDF and RAW2GDF, see sections 3.2 and 3.15 respectively. Output created by one of the SUPERFISH [4] set of codes can be converted using FISH2GDF, see section 3.3. When used in combination, ASCI2GDF, RAW2GDF or FISH2GDF and the field map elements, enable GPT to calculate the particle trajectories in electromagnetic fields calculated by Poisson solver codes.

The zero position for *z* in the GDF file containing the field map is always placed at the ECS specified in the field map element syntax. Therefore, **`map3D_E("wcs","z",position, ... )`** sets the *z*=0 position in the field map at *z*=**`position`** in GPT.

If a binary file named **`outbin`** consists of four arrays, the r- and z- coordinates and the radial and longitudinal electric field, and is located in the current directory then
`raw2gdf -o outbin.gdf outbin r 1 z 1 Er 1 Ez 1`
converts the file to the GDF format. Because GPT uses SI units, any non-SI parameter must be converted using the multiplication factors. For example, if the coordinates are specified in cm and the electric field is in V/cm, the following line can be used to convert all units to SI:
`raw2gdf -o outbin.gdf outbin r 0.01 z 0.01 Er 100 Ez 100`

Analogously, an ASCII file named **`outfile.txt`** can be converted using:
`asci2gdf -o outbin.gdf outfile.txt r 0.01 z 0.01 Er 100 Ez 100`

A file created by SF7 of the SUPERFISH codes can be directly converted using:
`fish2gdf –o outbin.gdf outsf7`

Once the file has been converted, the field map can be used in a GPT inputfile as demonstrated in the following example:

```
1.  # GPT Inputfile demonstrating the use of map2D_E
2.
3.  # Start a particle at 0,0,0 with v=0.5 c
4.  startpar( "wcs", "I", 0,0,0, 0,0,0.5) ;
5.
6.  position = 1 ;
7.
8.  map2D_E("wcs","z",position,  # Position field map around z=1
9.                 "outbin.gdf", # Read data from file outbin.gdf
10.                "r","z",      # r and z coordinates
11.                "Er","Ez",1   # Er and Ez are multiplied by 1
12.         ) ;
13.
14. tout(0,4/c,0.1/c) ;          # Output every 0.1/c till t=4/c
```

4.8.1 map1D_B

`map1D_B(ECS,filename,z,Bz,Bfac);`

Reads a 1D table of Bz samples on-axis from the specified GDF file and extrapolates these to a cylindrical symmetric field map for the magnetic field.

ECS	Element Coordinate System.
filename	Filename of GDF file containing the Bz-samples along the z-axis.
z	Name of array in the file specifying the z-coordinates.
Bz	Name of array in the file specifying the longitudinal magnetic field on-axis.
Bfac	Multiplication factor for the magnetic field.

The fields generated by this element are derived from [19]:

$$B_r(r,z) = -\tfrac{1}{2} r B_z'(z)$$
$$B_z(r,z) = B_z(z) - \tfrac{1}{4} r^2 B_z''(z) \qquad [4.69]$$

These fields are only correct near the **z**-axis of the element. The map25D_B and map3D_B elements can be used to specify off-axis field information.

A so-called 'natural cubic spline' is calculated from the given (**z**,**Bz**) points. It is recommended to specify more points at those positions where the first derivative fluctuates significantly.

To keep map1D_B as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF-file have thus to be specified on the commandline.

The data points specified in the GDF file **filename** need to be sorted in ascending z-order. Specifically for this 1D element the z-samples do not need to be equidistant.

Example:

```
map1D_B("wcs","z",1, "Bfield.gdf","z","Bz",5) ;
```

Where the file **bfield.gdf** is created from the ASCII file **bfield.txt**:

```
1.  # Example Bfield. z in [mm] and Bz in [Gauss]
2.    z    Bz
3.    0     0
4.    1    10
5.    2    21
6.    3    33
7.    4    46
8.    5    60
9.    6    75
10.   7    91
11.   8   108
12.  …     …
```

Conversion from ASCII to GDF format and conversion of the units to the MKS system used by GPT are done simultaneously by ASCI2GDF with the following command:

```
asci2gdf -o bfield.gdf bfield.txt z 1e-3 Bz 1e-4
```

4.8.2 map1D_E

```
map1D_E(ECS,filename,z,Ez,Efac);
```

Reads a 1D table of Ez samples on-axis from the specified GDF file and extrapolates these to a cylindrical symmetric field map for the electric field.

ECS	Element Coordinate System.
filename	Filename of GDF file containing the Ez samples along the z-axis.
z	Name of array in the file specifying the z-coordinates.
Ez	Name of array in the file specifying the longitudinal electric field on-axis.
Efac	Multiplication factor for the electric field.

The fields generated by this element are derived from [19]:

$$E_r(r,z) = -\tfrac{1}{2} r E_z'(z)$$
$$E_z(r,z) = E_z(z) - \tfrac{1}{4} r^2 E_z''(z) \qquad [4.70]$$

These fields are only correct near the **z**-axis of the element. The map25D_E and map3D_E elements can be used to specify off-axis field information.

A so-called 'natural cubic spline' is calculated from the given (**z**,**Ez**) points. It is recommended to specify more points at those positions where the first derivative fluctuates significantly.

To keep map1D_E as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have thus to be specified on the commandline.

The data points specified in the GDF file **filename** need to be sorted in ascending z-order, but do not need to be equidistant.

4.8.3 map1D_TM

map1D_TM(ECS,filename,z,Ez,fac,phi,w);
Reads a 1D table of Ez samples on-axis from the specified GDF file and extrapolates these to a cylindrical symmetric field map for a TM mode cavity.

ECS	Element Coordinate System.
filename	Filename of GDF file containing the Ez-samples along the z-axis.
z	Name of array in the file specifying the z-coordinates.
Ez	Name of array in the file specifying the longitudinal electric field on-axis.
fac	Multiplication factor for all fields.
phi	Phase factor: ϕ in radians.
w	Angular frequency: ω.

The fields generated by this element are derived from:

$$\begin{aligned} E_z(r,z) &= E_z(z)\cos(\omega t+\phi) \\ E_r(r,z) &= -\tfrac{1}{2} r E_z'(z)\cos(\omega t+\phi) \\ B_\phi(r,z) &= \tfrac{r\omega}{2c^2} E_z(z)\sin(\omega t+\phi) \end{aligned} \qquad [4.71]$$

These fields are only correct near the **z**-axis of the element. The map25D_TM element can be used to specify off-axis field information.

A so-called 'natural cubic spline' is calculated from the given (**z**,**Ez**) points. It is recommended to specify more points at those positions where the first derivative fluctuates significantly.

To keep map1D_TM as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have thus to be specified on the commandline.

The datapoints specified in the GDF file **filename** need to be sorted in ascending z-order, but do not need to be equidistant.

4.8.4 Map2D_B

map2D_B(ECS,mapfile.gdf,r,z,Br,Bz,Bfac) ;
Reads a 2D cylindrical symmetric field map for the magnetic field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r-coordinates [m].
z	Name of array in the file with the z-coordinates.[m]
Br	Name of array in the file with the radial magnetic field [T].
Bz	Name of array in the file with the longitudinal magnetic field [T].
Bfac	Multiplication factor for the magnetic field.

To keep map2D_B as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have thus to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **r-** and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

To calculate the fields, first the four datapoints around the particle position are determined. They are numbered 1 to 4 as shown in Figure 4–15.

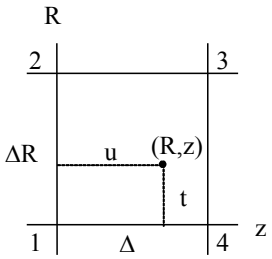


Figure 4–15: Datapoints around the particle position (R,z).

The relative distances *t* and *u* are calculated using:

$$t = \frac{(R - R_1)}{\Delta R}$$
$$u = \frac{(z - z_1)}{\Delta z} \qquad [4.72]$$

Bilinear interpolation of the magnetic field yields:

$$B_r = (1-t)(1-u)Br_1 + t(1-u)Br_2 + tuBr_3 + (1-t)uBr_4$$
$$B_\varphi = 0 \qquad [4.73]$$
$$B_z = (1-t)(1-u)Bz_1 + t(1-u)Bz_2 + tuBz_3 + (1-t)uBz_4$$

Within the element, the magnetic field is continuous in all directions.

4.8.5 Map2D_E

map2D_E(ECS,mapfile.gdf,r,z,Er,Ez,Efac) ;
Reads a 2D cylindrical symmetric field map for the electric field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r-coordinates [m]
z	Name of array in the file with the z-coordinates [m].
Er	Name of array in the file with the radial electric field [V/m].
Ez	Name of array in the file with the longitudinal electric field [V/m].
Efac	Multiplication factor for the electric field.

To keep map2D_E as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDFfile thus have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **r**-and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Bilinear interpolation of the electric field yields:

$$\begin{aligned} E_r &= (1-t)(1-u)Er_1 + t(1-u)Er_2 + tuEr_3 + (1-t)uEr_4 \\ E_\varphi &= 0 \\ E_z &= (1-t)(1-u)Ez_1 + t(1-u)Ez_2 + tuEz_3 + (1-t)uEz_4 \end{aligned} \qquad [4.74]$$

Within the element, the electric field is continuous in all directions.

4.8.6 Map2D_Et

`Map2D_Et(ECS,mapfile.gdf,r,z,Er,Ez,fac,tau) ;`
Reads a 2D electrostatic field-profile assuming linearly increasing amplitude as function of time.

`ECS`	Element Coordinate System.
`mapfile`	Filename of GDF file containing the fields.
`r`	Name of array in the file specifying the r-coordinates.
`z`	Name of array in the file specifying the z-coordinates.
`Er`	Name of array in the file specifying the electric field in the r-direction.
`Ez`	Name of array in the file specifying the electric field in the z-direction.
`fac`	Multiplication factor for the fields.
`tau`	Inverse slope [s].

The electromagnetic fields of this element are given by:

$$\begin{aligned} \mathbf{E}_{z,r}(z,r,t) &= \mathbf{E}_{z,r}(z,r)\frac{t}{\mathbf{tau}} \\ \mathbf{B}_\varphi(z,r,t) &= \frac{1}{r\,c^2\,\mathbf{tau}}\int_0^r r'\,E_z(z,r')\,dr' \end{aligned} \qquad [4.75]$$

where $\mathbf{E}_{z,r}(z,r)$ is read from **`mapfile`** and interpolated analogous to the **`map2D_E`** element. To prevent efficiency degradation, the magnetostatic field is pre-computed only once and stored in memory at the very beginning of the simulation.

Please note that a linear slope as function of time is the *only* time-dependent function that can be used as multiplication factor for an electrostatic field map.

The datapoints specified in the GDF file **`filename`** don't need to be in any particular order, because the complete set is first sorted in the **`r`**- and **`z`**-directions for convenient internal use. From this sorted data the grid characteristics, i.e zmin, zmax, and zdelta, are extracted. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

4.8.7 Map2Dr_E

`map2Dr_E(ECS,mapfile.gdf,x,z,Ex,Ez,L,Efac) ;`
Reads a 2D field map in the zx-plane for the electric field from the specified GDF file.

`ECS`	Element Coordinate System.
`mapfile`	Filename of GDF file containing the fields.
`x`	Name of array in the file with the x-coordinates [m].
`z`	Name of array in the file with the z-coordinates [m]
`Ex`	Name of array in the file with the electric field in x-direction [V/m].
`Ez`	Name of array in the file with the electric field in z-direction [V/m].
`L`	'Thickness' of field map [m] in y-direction.
`Efac`	Multiplication factor for the electric field.

To keep **`map2Dr_E`** as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **`filename`** don't need to be in any particular order, because the complete set is first sorted in the **`z`**- and **`x`**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. zmin, zmax, and zdelta. are calculated. It is verified that all the grid points are

present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

The field map is only active when $|y|<$**L**/2. In that case, bilinear interpolation of the electric field yields:

$$\begin{aligned} E_x &= (1-t)(1-u)Ex_1 + t(1-u)Ex_2 + tuEx_3 + (1-t)uEx_4 \\ E_y &= 0 \\ E_z &= (1-t)(1-u)Ez_1 + t(1-u)Ez_2 + tuEz_3 + (1-t)uEz_4 \end{aligned} \qquad [4.76]$$

Within the element, the electric field is continuous in all directions.

4.8.8 Map2D_V

map2D_V(ECS,mapfile.gdf,r,z,V,Vfac,[Vtype]) ;
Reads a 2D cylindrical symmetric field map for the electric potential from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r-coordinates [m]
z	Name of array in the file with the z-coordinates [m]
V	Name of array in the file with the electric potential [V].
Vfac	Multiplication factor for the electric potential.
Vtype	Interpolation method for **V**. Must be **lin**.

To keep map2D_V as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **r**- and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

When the interpolation type for the electric potential is 'Lin', a bilinear interpolation is performed. The source code allows other interpolation types to be implemented, but currently the only available interpolation type is 'Lin'.

When the electric potential is specified, the electric field is calculated using
$\mathbf{E} = -\nabla V = -\operatorname{grad} V$
which yields the following using the bilinear interpolation method for potential V:

$$\begin{aligned} E_r &= \frac{(1-u)V_1 - (1-u)V_2 - uV_3 + uV_4}{\Delta R} \\ E_\varphi &= 0 \\ E_z &= \frac{(1-t)V_1 + tV_2 - tV_3 - (1-t)V_4}{\Delta z} \end{aligned} \qquad [4.77]$$

Please note that because the potential is interpolated linearly, its derivative **E** is constant between the gridpoints. This results in discontinuities in the electric field. A higher-order interpolation is needed to produce a continuous electric field. Therefore it is currently more accurate to use the electric field in a simulation than the potential.

4.8.9 Map25D_E

map25D_E(ECS,mapfile.gdf,r,z,Er,Ephi,Ez,Efac) ;
Reads a 2.5D cylindrical symmetric field map for the electric field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r -oordinates [m].

z	Name of array in the file with the z-coordinates [m].
Er	Name of array in the file with the radial electric field [V/m].
Ephi	Name of array in the file with the azimutal electric field [V/m].
Ez	Name of array in the file with the longitudinal electric field [V/m].
Efac	Multiplication factor for the electric field.

To keep map25D_E as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **r**- and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Bilinear interpolation of the electric field yields:

$$\begin{aligned} E_r &= (1-t)(1-u)Er_1 + t(1-u)Er_2 + tuEr_3 + (1-t)uEr_4 \\ E_\varphi &= (1-t)(1-u)E\varphi_1 + t(1-u)E\varphi_2 + tuE\varphi_3 + (1-t)uE\varphi_4 \\ E_z &= (1-t)(1-u)Ez_1 + t(1-u)Ez_2 + tuEz_3 + (1-t)uEz_4 \end{aligned} \qquad [4.78]$$

The electric field is continuous in all directions. The angular electric field is not chosen to be zero, as is the case when using map2D_E.

4.8.10 Map25D_B

map25D_B(ECS,mapfile.gdf,r,z,Br,Bphi,Bz,Bfac) ;
Reads a 2.5D cylindrical symmetric field map for the magnetic field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r-coordinates [m]
z	Name of array in the file with the z-coordinates [m]
Br	Name of array in the file with the radial magnetic field [T].
Bphi	Name of array in the file with the angular magnetic field [T].
Bz	Name of array in the file with the longitudinal magnetic field [T].
Bfac	Multiplication factor for the magnetic field.

To keep map25D_B as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **r**- and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Very similar, bilinear interpolation of the magnetic field yields:

$$\begin{aligned} B_r &= (1-t)(1-u)Br_1 + t(1-u)Br_2 + tuBr_3 + (1-t)uBr_4 \\ B_\varphi &= (1-t)(1-u)B\varphi_1 + t(1-u)B\varphi_2 + tuB\varphi_3 + (1-t)uB\varphi_4 \\ B_z &= (1-t)(1-u)Bz_1 + t(1-u)Bz_2 + tuBz_3 + (1-t)uBz_4 \end{aligned} \qquad [4.79]$$

The magnetic field is continuous in all directions, but please note that if $\nabla\cdot\mathbf{B}\neq 0$ in the original datafile, the effect of magnetic monopoles is simulated.

4.8.11 map25D_TM

map25D_TM(ECS,mapfile.gdf,r,z,Er,Ez,Bphi,ffac,k,phi,w)
Reads a 2.5D cylindrical symmetric field map of a cavity in TM-mode from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
r	Name of array in the file with the r-coordinates [m].
z	Name of array in the file with the z-coordinates [m].
Er	Name of array in the file with the radial electric field [V/m].
Ez	Name of array in the file with the longitudinal electric field [V/m].
Bphi	Name of array in the file with the angular magnetic field [T].
ffac	Multiplication factor for the electromagnetic field.
k	Longitudinal wavenumber k in [m^{-1}]. Must be zero for the simulation of a standing wave cavity.
phi	RF phase offset: φ.
w	Angular frequency: ω.in [s-1].

To keep map25D_TM as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **mapfile** don't need to be in any particular order, because the complete set is first sorted in the **r-** and **z**-directions for convenient internal use. From this sorted data the grid characteristics, i.e. rmin, rmax, and rdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Very similar to map25D_B, bilinear interpolation of the electromagnetic field yields:

$$\begin{aligned} E_r &= \cos(\omega t - kz + \varphi)\big((1-t)(1-u)Er_1 + t(1-u)Er_2 + tuEr_3 + (1-t)uEr_4\big) \\ E_z &= \cos(\omega t - kz + \varphi)\big((1-t)(1-u)Ez_1 + t(1-u)Ez_2 + tuEz_3 + (1-t)uEz_4\big) \\ B_\varphi &= -\sin(\omega t - kz + \varphi)\big((1-t)(1-u)B\varphi_1 + t(1-u)B\varphi_2 + tuB\varphi_3 + (1-t)uB\varphi_4\big) \end{aligned} \qquad [4.80]$$

FISH2GDF can be used to import fieldmaps created by the SUPERFISH set of codes. The converted file can subsequently be read using:

```
map25D_TM("wcs","I","fieldmap.gdf","R","Z","Er","Ez","H", 1,0, phi,2*pi*freq) ;
```

4.8.12 Map3D_E

```
map3D_E(ECS,mapfile.gdf,x,y,z,Ex,Ey,Ez,Efac) ;
```

Reads a 3D rectangular field map for the electric field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
x	Name of array in the file with the x-coordinates [m].
y	Name of array in the file with the y-coordinates [m].
z	Name of array in the file with the z-coordinates [m].
Ex	Name of array in the file with the electric field in the x-direction [V/m].
Ey	Name of array in the file with the electric field in the y-direction [V/m].
Ez	Name of array in the file with the electric field in the z-direction [V/m].
Efac	Multiplication factor for the electric field.

To keep map3D_E as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x-**, **y-** and **z-**directions for convenient internal use. From this sorted data the grid characteristics, i.e. xmin, xmax, and xdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

To calculate the fields, first the eight datapoints around the particle position are determined. They are numbered 1 to 8 as shown in Figure 4–16.

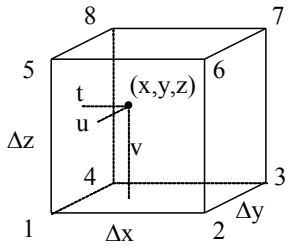


Figure 4–16: Datapoints around the particle position (x,y,z).

The relative distances *t*, *u* and *v* are calculated using:

$$t = \frac{(x - x_1)}{\Delta x}$$
$$u = \frac{(y - y_1)}{\Delta y} \qquad [4.81]$$
$$v = \frac{(z - z_1)}{\Delta z}$$

Bilinear interpolation of the electric field yields:

$$\begin{aligned} E_x &= (1-t)(1-u)(1-v)Ex_1 + t(1-u)(1-v)Ex_2 + tu(1-v)Ex_3 + (1-t)u(1-v)Ex_4 + \\ &\quad (1-t)(1-u)vEx_5 + t(1-u)vEx_6 + tuvEx_7 + (1-t)uvEx_8 \\ E_y &= (1-t)(1-u)(1-v)Ey_1 + t(1-u)(1-v)Ey_2 + tu(1-v)Ey_3 + (1-t)u(1-v)Ey_4 + \\ &\quad (1-t)(1-u)vEy_5 + t(1-u)vEy_6 + tuvEy_7 + (1-t)uvEy_8 \\ E_z &= (1-t)(1-u)(1-v)Ez_1 + t(1-u)(1-v)Ez_2 + tu(1-v)Ez_3 + (1-t)u(1-v)Ez_4 + \\ &\quad (1-t)(1-u)vEz_5 + t(1-u)vEz_6 + tuvEz_7 + (1-t)uvEz_8 \end{aligned} \qquad [4.82]$$

The electric field is continuous in all directions.

4.8.13 Map3D_TM

`map3D_TM(ECS,mapfile.gdf,x,y,z,Ex,Ey,Ez,Bx,By,Bz,ffac,phi,w) ;`

Reads a 3D rectangular field map for a standing electromagnetic wave pattern from the specified GDF-file.

`ECS`	Element Coordinate System.
`mapfile`	Filename of GDF-file containing the fields.
`x`	Name of array in the file specifying the x coordinates.
`y`	Name of array in the file specifying the y coordinates.
`z`	Name of array in the file specifying the z coordinates.
`Ex`	Name of array in the file specifying the electric field in the x direction.
`Ey`	Name of array in the file specifying the electric field in the y direction.
`Ez`	Name of array in the file specifying the electric field in the z direction.
`Bx`	Name of array in the file specifying the magnetic field in the x direction.
`By`	Name of array in the file specifying the magnetic field in the y direction.
`Bz`	Name of array in the file specifying the magnetic field in the z direction.
`ffac`	Multiplication factor for the field.
`phi`	Phase offset: φ.
`w`	Angular frequency: ω.

The electromagnetic fields of this element are given by:

$$\mathbf{E} = \cos(w\,t + \varphi)\,\mathbf{E}_{\text{int}}$$
$$\mathbf{B} = -\sin(w\,t + \varphi)\,\mathbf{B}_{\text{int}}$$

Where $\mathbf{E}_{\text{int}}$ and $\mathbf{B}_{\text{int}}$ are obtained by trilinear interpolation of the fields specified in **`mapfile`**. Pleaase see **`map3D_E`** and **`map3D_B`** for a description of the interpolation equations.

To keep map3D_EB as versatile as possible, no assumptions are made as to the names of the columns in the GDF-file specifying the data. Therefore the names of the columns in the GDF-file have to be specified on the commandline.

The datapoints specified in the GDF-file **`filename`** don't need to be in any particular order, because the complete set is first sorted in the **`x`**, **`y`** and **`z`** directions for convenient internal use. From this sorted data the grid characteristics, i.e. minimum, maximum and spacing are extracted. Then it is made sure that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Example:
```
Map3D_EB("wcs","z",1, "EBmap.gdf",
       "x","y","z","Ex","Ey","Ez","Bx","By","Bz",
       1.0, 0.0, 2998e6) ;
```

Where the file **EBmap.gdf** is created from the ASCII file **EBfield.txt**:
```
# Example EBfield. r in [mm], E in [T] and B in [Gauss]
x   y     z     Ex    Ey    Ez    Bx    By    Bz
…   …     …     …     …     …     …     …     …
…   …     …     …     …     …     …     …     …
```

Conversion from ASCII to GDF format and conversion of the units to the MKS system used by GPT are done simultaneously by ASCI2GDF with the following command:
```
ASCI2GDF -o EBmap.gdf EBfield.txt x 1e-3 y 1e-3 z 1e-3 Ex 1 Ey 1 Ez 1 Bx 1e-4
By 1e-4 Bz 1e-4
```

4.8.14 Map3D_Ecomplex

```
map3D_Ecomplex(ECS,mapfile.gdf,x,y,z,Exre,Eyre,Ezre,Exim,Eyim,Ezim,ffac,phi,w)
;
```
Reads a 3D rectangular field map for an oscillating electric field from the specified GDF-file.

ECS	Element Coordinate System.
mapfile	Filename of GDF-file containing the fields.
x	Name of array in the file specifying the x coordinates.
y	Name of array in the file specifying the y coordinates.
z	Name of array in the file specifying the z coordinates.
Exre	Name of array in the file specifying the real part of the electric field in the x direction.
Eyre	Name of array in the file specifying the real part of the electric field in the y direction.
Ezre	Name of array in the file specifying the real part of the electric field in the z direction.
Exim	Name of array in the file specifying the imaginary part of the electric field in the x direction.
Eyim	Name of array in the file specifying the imaginary part of the electric field in the y direction.
Ezim	Name of array in the file specifying the imaginary part of the electric field in the z direction.
ffac	Multiplication factor for the field.
phi	Phase offset: φ.
w	Angular frequency: ω.

The electric fields of this element are given by:
$\mathbf{E} = \cos(w\,t + \phi)\,\mathbf{E}_{\mathrm{re}} - \sin(w\,t + \phi)\,\mathbf{E}_{\mathrm{im}}$
Where $\mathbf{E}_{\mathrm{re}}$ and $\mathbf{E}_{\mathrm{im}}$ are obtained by trilinear interpolation of the fields specified in **mapfile**. Pleaase see **map3D_E** and **map3D_B** for a description of the interpolation equations.

To keep **map3D_Ecomplex** as versatile as possible, no assumptions are made as to the names of the columns in the GDF-file specifying the data. Therefore the names of the columns in the GDF-file have to be specified on the commandline.

The datapoints specified in the GDF-file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x**, **y** and **z** directions for convenient internal use. From this sorted data the grid characteristics, i.e. minimum, maximum and spacing are extracted. Then it is made sure that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

4.8.14.1 CST MW-Studio

Fields calculated by the Eigenmode solver of CST Microwave Studio can be imported in GPT with a trivial modification. The procedure for the **E** field is as follows:

In CST MW-Studio:

- Select the desired eigenmode in the Navigation Tree, under 2D/3D Results/Mode *n*.
- Typically, only a small part of the calculation domain is relevant for tracking. The desired subvolume to be exported can be defined under Properties…/Specials…/Subvolume…
- Export the fields in the subvolume with File/Export/Plot data (ASCII). The step width must be fixed, but it does not need to be equal in all directions.

With an ASCII editor capable of handling large files:

- The first two lines of the ASCII file produced by MW-Studio are typically

```
      x [mm]          y [mm]          z [mm]      ExRe [V/m]      EyRe [V/m]      EzRe [V/m]      ExIm [V/m]
…
---------------------------------------------------------------------------------------------------------------
-…
```

- These two lines must be changed into the following single line, where spacing is irrelevant

```
x y z ExRe EyRe EzRe ExIm Eyim Ezim
```

In GPT:

- Before this ASCII file can be used as a field-map in a GPT inputfile, it must be converted to the GDF format. Assuming the ASCII filename is **`mapE.txt`**, and assuming the dimensions are in mm, this can be done in a (GPT) batch file with
  `asci2gdf -o mapE.gdf mapE.txt x 1e-3 y 1e-3 z 1e-3`
- It is good practice to check the produced field map by opening it in GPTwin and just plotting a few 2D projections, such as z-Ezre, x-y, etc.
- The field map can subsequently be positioned in a GPT inputfile with:

```
Zpos=1.3 ; # z-position of z=0 in the field-map
fac=1 ;
w=2*pi*2998e6 ;
phi=0 ;
map3D_Ecomplex("wcs", "z",zpos, "mapE.gdf", "x","y","z",
"Exre","Eyre","Ezre", "Exim","Eyim","Ezim", fac,phi,w) ;
```

The procedure for importing the **H** field is fully analogous, where it is noted that the two field-map elements are imported on top of each other in the GPT inputfile. This is by design; GPT automatically adds all fields, and the combination of the **E** and **B** fields is the desired total field.

4.8.15 Map3D_Hcomplex

`map3D_Hcomplex(ECS,mapfile.gdf,x,y,z,Hxre,Hyre,Hzre,Hxim,Hyim,Hzim,ffac,phi,w)`
`;`

Reads a 3D rectangular field map for an oscillating magnetic field from the specified GDF-file.

`ECS`	Element Coordinate System.
`mapfile`	Filename of GDF-file containing the fields.
`x`	Name of array in the file specifying the x coordinates.
`y`	Name of array in the file specifying the y coordinates.
`z`	Name of array in the file specifying the z coordinates.
`Hxre`	Name of array in the file specifying the real part of the electric field in the x direction.
`Hyre`	Name of array in the file specifying the real part of the electric field in the y direction.
`Hzre`	Name of array in the file specifying the real part of the electric field in the z direction.
`Hxim`	Name of array in the file specifying the imaginary part of the electric field in the x direction.
`Hyim`	Name of array in the file specifying the imaginary part of the electric field in the y direction.
`Hzim`	Name of array in the file specifying the imaginary part of the electric field in the z direction.
`ffac`	Multiplication factor for the field.
`phi`	Phase offset: φ.
`w`	Angular frequency: ω.

The electromagnetic fields of this element are given by:

$\mathbf{B} = \mu_0 \cos(w\,t + \phi)\,\mathbf{H}_{re} - \mu_0 \sin(w\,t + \phi)\,\mathbf{H}_{im}$

Where $\mathbf{H}_{re}$ and $\mathbf{H}_{im}$ are obtained by trilinear interpolation of the fields specified in **mapfile**. Pleaase see **map3D_E** and **map3D_B** for a description of the interpolation equations.

To keep **map3D_Hcomplex** as versatile as possible, no assumptions are made as to the names of the columns in the GDF-file specifying the data. Therefore the names of the columns in the GDF-file have to be specified on the commandline.

The datapoints specified in the GDF-file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x**, **y** and **z** directions for convenient internal use. From this sorted data the grid characteristics, i.e. minimum, maximum and spacing are extracted. Then it is made sure that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

4.8.16 Map3D_V

map3D_V(ECS,mapfile.gdf,x,y,z,V,Vfac,[Vtype]) ;

Reads a 3D rectangular field map for the electric potential from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
x	Name of array in the file with the x-coordinates.
y	Name of array in the file with the y-coordinates.
z	Name of array in the file with the z-coordinates.
V	Name of array in the file with the electric potential.
Vfac	Multiplication factor for the electric potential.
Vtype	Interpolation method for **V**. Must be **lin**.

To keep map3D_V as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x-**, **y-** and **z-**directions for convenient internal use. From this sorted data the grid characteristics, i.e. xmin, xmax, and xdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

When the interpolation type for the electric potential is "Lin", a bilinear interpolation is performed. The source code allows other interpolation types to be implemented, but currently the only available interpolation type is "Lin".

When the electric potential is specified, the electric field is calculated using

$\mathbf{E} = -\nabla V = -\,\mathrm{grad}\,V$

which yields the following using the bilinear interpolation method for potential *V*:

$$\begin{aligned} E_x &= \left[(1-u)(1-v)V_1 - (1-u)(1-v)V_2 - u(1-v)V_3 + u(1-v)V_4 + \right. \\ &\quad \left. (1-u)vV_5 - (1-u)vV_6 - uvV_7 + uvV_8\right]/\Delta x \\ E_y &= \left[(1-t)(1-v)V_1 + t(1-v)V_2 - t(1-v)V_3 - (1-t)(1-v)V_4 + \right. \\ &\quad \left. (1-t)vV_5 + tvV_6 - tvV_7 - (1-t)vV_8\right]/\Delta y \\ E_z &= \left[(1-t)(1-u)V_1 + t(1-u)V_2 + tuV_3 + (1-t)uV_4 - \right. \\ &\quad \left. (1-t)(1-u)V_5 - t(1-u)V_6 - tuV_7 - (1-t)uV_8\right]/\Delta z \end{aligned} \qquad [4.83]$$

Please note that because the potential is interpolated linearly, its derivative **E** is constant between the gridpoints. This results in discontinuities in the electric field. A higher-order interpolation is needed to produce a continuous electric field. Therefore it is currently more accurate to use the electric field in a simulation than the potential.

4.8.17 Map3D_B

map3D_B(ECS,mapfile.gdf,x,y,z,Bx,By,Bz,Bfac) ;
Reads a 3D rectangular field map for the magnetic field from the specified GDF file.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
x	Name of array in the file with the x-coordinates.
y	Name of array in the file with the y-coordinates.
z	Name of array in the file with the z-coordinates.
Bx	Name of array in the file with the magnetic field in the x-direction.
By	Name of array in the file with the magnetic field in the y-direction.
Bz	Name of array in the file with the magnetic field in the z-direction.
Bfac	Multiplication factor for the magnetic field.

To keep map3D_B as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x-**, **y-** and **z-**directions for convenient internal use. From this sorted data the grid characteristics, i.e. xmin, xmax, and xdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Very similar, bilinear interpolation of the magnetic field yields:

$$\begin{aligned}
B_x &= (1-t)(1-u)(1-v)Bx_1 + t(1-u)(1-v)Bx_2 + tu(1-v)Bx_3 + (1-t)u(1-v)Bx_4 + \\
&\quad (1-t)(1-u)vBx_5 + t(1-u)vBx_6 + tuvBx_7 + (1-t)uvBx_8 \\
B_y &= (1-t)(1-u)(1-v)By_1 + t(1-u)(1-v)By_2 + tu(1-v)By_3 + (1-t)u(1-v)By_4 + \\
&\quad (1-t)(1-u)vBy_5 + t(1-u)vBy_6 + tuvBy_7 + (1-t)uvBy_8 \\
B_z &= (1-t)(1-u)(1-v)Bz_1 + t(1-u)(1-v)Bz_2 + tu(1-v)Bz_3 + (1-t)u(1-v)Bz_4 + \\
&\quad (1-t)(1-u)vBz_5 + t(1-u)vBz_6 + tuvBz_7 + (1-t)uvBz_8
\end{aligned} \qquad [4.84]$$

The magnetic field is continuous in all directions, but please note that if $\nabla\cdot\mathbf{B}\neq 0$ in the original datafile, the effect of magnetic monopoles is simulated.

4.8.18 Map3D_remove

map3D_remove(ECS,mapfile.gdf,x,y,z,R) ;
Reads a 3D rectangular map indicating if a particle must be removed or not.

ECS	Element Coordinate System.
mapfile	Filename of GDF file containing the fields.
x	Name of array in the file with the x-coordinates.
y	Name of array in the file with the y-coordinates.
z	Name of array in the file with the z-coordinates.
R	Name of array in the file containing either 1 or 0, where 1 indicates that all particles at that location must be removed.

To keep map3D_remove as versatile as possible, no assumptions are made as to the names of the columns in the GDF file specifying the data. The names of the columns in the GDF file have to be specified on the commandline.

The rows specified in the GDF file **filename** don't need to be in any particular order, because the complete set is first sorted in the **x-**, **y-** and **z-**directions for convenient internal use. From this sorted data the grid characteristics, i.e. xmin, xmax, and xdelta, are calculated. It is verified that all the grid points are present and lie within a range of $\pm\Delta\ 10^{-3}$ off their optimal position, where Δ is the grid spacing in the corresponding direction.

Particles are removed if the bilinar interpolated value of all corners of the enclosing box is larger than 0.5.

4.9 Free Electron Laser (FEL)

This section describes elements for the simulation of Free Electron Lasers (FELs) with GPT. These elements allow multi-mode, multi-pass FEL simulations, with emphasis on phase-space evolution of the electron bunch during passage through the undulator. This is an expert section for expert users. If you plan to use GPT for serious design work on FELs, please contact us for additional advice.

4.9.1 Gauss00mf

`gauss00mf(ECS, w0, L, fmin, fmax, Nm, Ain, dz, alpha, LVset ) ;`
Multiple first order Gaussian modes in free space.

`ECS`	Element Coordinate System.
`w0`	Radius of waist [m].
`L`	Total cavity length [m].
`fmin`	Minimum frequency [Hz].
`fmax`	Maximum frequency [Hz].
`Nm`	Number of modes to be included in the simulation.
`Ain`	Initial field amplitude [V/m] for all modes.
`dz`	Initial translation of center pulse [m].
`alpha`	Damping [s^{-1}].
`LVset`	Name of the set containing 'Litvinenko' particles. To disable this option, an empty string "" must be specified.

The **`gauss00mf`** element of GPT models a radiation pattern consisting of a (large) number of first-order Gaussian waves propagating in free space [17F23]. Both the effect of the waves on the particle trajectories and the effect of the particles on the wave amplitudes and phases are included.

4.9.1.1 Equations

The Gaussian waves included in the simulation are all assumed to have a polarization in the x-direction. The total electromagnetic fields of all modes *j*, each with amplitude A_j and phase φ_j, is then given by:

$$E_x = \sum_j A_j \frac{w_0}{w_j(z)} e^{-\frac{r^2}{w_j^2(z)}} \cos\left(\omega_j t - k_j z - \frac{k_j r^2 z}{2(z^2+z_{0,j}^2)} + \arctan\left(\frac{z}{z_{0,j}}\right) + \varphi_j\right) \quad [4.85]$$

$$B_y = E_x / c$$

where

$$w_j(z) = w_0\sqrt{1+z^2/z_{0,j}^2}$$

$$z_{0,j} = \tfrac{1}{2} k_j w_0^2$$

$$\omega_j = k_j c$$

$$k_j = \left(n\pi - \arctan(2L^2/n\pi w_0^2)\right)/L$$

and *n* is the mode number, *r* is the radial position of a particle and *z* the longitudinal position. The sign of both the electric and the magnetic field is reversed for all particles in the Litvinenko set.

The total stored energy of mode *j* in length *L* is given by:

$$W_j = \tfrac{1}{2} A_j^2 \pi\varepsilon_0 w_0^2 L \quad [4.86]$$

The differential equations for the amplitudes and phases of all modes are derived by first separating each wave *j* in two orthogonal waves with amplitudes *u* and *v*:

$$E_{x,j} = u_j T_j \cos(\theta_j) - v_j T_j \sin(\theta_j) \quad [4.87]$$

where

$$u_j = A_j \cos(\varphi_j)$$
$$v_j = A_j \sin(\varphi_j)$$
$$T_j = \frac{w_0}{w_j(z)} e^{-\frac{r^2}{w_j^2(z)}} \qquad [4.88]$$
$$\theta_j = \omega_j t - k_j z - \frac{k_j r^2 z}{2(z^2 + z_{0,j}^2)} + \arctan\left(z / z_{0,j}\right)$$

The total number of modes N_{all} in the specified frequency range **fmin** to **fmax** is given by:

$$N_{all} = \frac{2L}{c}(\mathbf{fmax} - \mathbf{fmin}) \qquad [4.89]$$

In the simulation only a specified limited number of modes **Nm** is included. The mode spacing ΔN can be calculated from:

$$\Delta N = \frac{Nall}{\mathbf{Nm}} \qquad [4.90]$$

The most accurate simulation, with ΔN=1, includes all modes in the simulation. When ΔN is 2, every other mode is included. In other words, every simulated mode represents not only itself but also its neighboring mode. With typical FEL parameters with high frequencies and large cavities, a ΔN parameter of more than one thousand is normal, indicating that every simulated mode represents over a thousand actual modes.

The mode numbers n_j used in the simulation are uniformly distributed between the specified frequencies by:

$$n_j = \frac{2L\,\mathbf{fmin}}{c} + \left(j - \tfrac{1}{2}\right)\Delta N, \quad j = 1 \ldots \mathbf{Nm} \qquad [4.91]$$

Because both ΔN and n must be integers, the specified **Nm** is adapted to model the simulated frequency range properly between **fmin** and **fmax**.

The energy lost by each macro-particle i, with charge Q_i, contributes to an increase in stored energy in all ΔN modes represented by mode j:

$$\frac{dW_j}{dt} = -\Delta N \sum_i Q_i \, \mathbf{v}_i \cdot \mathbf{E}_j \qquad [4.92]$$

The differential equations solved by this element can be derived from the orthogonality of the modes and conservation of energy. An additional frequency independent damping term is added to account for outcoupling and mirror losses. The equations are solved for each mode individually and given by:

$$\frac{du_j}{dt} = -\sum_i \frac{Q_i \, \Delta N}{\varepsilon_0 \pi w_0^2 L} v_{x,i} T_j \cos(\theta_j) - \mathbf{alpha}\, u_j$$
$$\frac{dv_j}{dt} = +\sum_i \frac{Q_i \, \Delta N}{\varepsilon_0 \pi w_0^2 L} v_{x,i} T_j \sin(\theta_j) - \mathbf{alpha}\, v_j \qquad [4.93]$$

The sign of the first term, not the damping, is reversed for all particles in the Litvinenko set.

The boundary conditions of the differential equations, the amplitudes and phases of all modes at t=0, are given by:

$$A_{j,t=0} = \mathbf{Ain}\ \Delta N$$
$$\varphi_{j,t=0} = k_j\, \mathbf{dz} \qquad [4.94]$$

The initial pulse is created in the center of the cavity. In order to overlap the radiation pulse with an electron bunch, the **dz** parameter can be used to translate the initial pulse. For example, for a cavity between z=0 and z=L the pulse at t=0 can be moved to z=0 by specifying **dz**=–L/2.

4.9.1.2 Frequency domain output

At every time output of GPT, the **G00mf** element writes the amplitude, phase, frequency and wavelength of each mode in the GPT outputfile:

A00 Array of amplitudes of all modes [V/m]. Can be plotted as function of phase, frequency or wavelength.
phi00 Array of phases of all modes [rad].
freq00 Array of frequencies of all modes [Hz].
lam00 Array of wavelengths of all modes [m].

The amplitude and phase of mode *j* written in the GPT outputfile are calculated from:

$$\begin{aligned} \mathbf{A00}_j &= A_j / \Delta N = \sqrt{u_j^2 + v_j^2} \Big/ \Delta N \\ \mathbf{phi00}_j &= \arctan\left(\frac{v_j}{u_j}\right) \end{aligned} \quad [4.95]$$

The division by ΔN for the amplitude compensates the multiplication by ΔN in the interaction strength. This makes the amplitude of the output spectrum independent of the number of modes.

The total energy in the pulse is also written in the outputfile:

W00 Total energy in the pulse [J].

$$\mathbf{W00} = \frac{\pi \varepsilon_0 w_0^2 L}{2\,\Delta N} \sum_j A_j^2 \quad [4.96]$$

The total energy in the pulse is equal to the energy lost by the particles:

$$W_{00} = \Delta\gamma\, mc^2 \frac{Q_{tot}}{q_e} \quad [4.97]$$

4.9.1.3 Time-domain output

When **Nm**≠1, at every time output of GPT, the **G00mf** element writes the time-structure of the Gaussian modes to the GPT outputfile in the following arrays:

At00 Array of amplitudes [V/m], to be plotted as function of **t00**.
Pt00 Array of power [W], to be plotted as function of **t00**.
t00 Array for the time-axis [s].

The time-domain output is defined by the Fourier transform of the frequency domain. When all modes are included in the simulation, ΔN=1, the time-domain output repeats itself after round-trip time $2L/c$. A mode spacing of ΔN in frequency domain is equivalent to N identical pulses in the round-trip time. As a result, all relevant time output is defined in the interval Δt given by:

$$\Delta t = \frac{2L}{c\,\Delta N} \quad [4.98]$$

The time-structure is calculated by the discrete Fourier transformation of the frequency spectrum at z=0 using:

$$\begin{aligned} \mathbf{t00}_k &= t_k = \left\{-\tfrac{1}{2}\Delta t, dt - \tfrac{1}{2}\Delta t, 2\,dt - \tfrac{1}{2}\Delta t, \ldots, \tfrac{1}{2}\Delta t\right\} \\ \mathbf{At00}_k &= At_{k+koffset} = \sum_j A_j \cos\left(\omega_j\, t_k + \varphi_j\right) \\ \mathbf{Pt00}_k &= \tfrac{1}{2}\pi\varepsilon_0 w_0^2 c\, \mathbf{At00}_k^2 \end{aligned} \quad [4.99]$$

where the spacing in time domain dt is given by:

$$dt = \frac{1}{7\,\mathbf{fmax}} \quad [4.100]$$

To ease the interpretation of the results, *koffset* is chosen such that the pulse is centered around t=0. This is implemented by a robust minimalization of σ from:

$$\sigma = \sum_k \left|At_{k+koffset}\right| \cdot \left|t_k\right| \quad [4.101]$$

All arrays for both frequency and time output can be visualized directly with GPTwin, or analyzed with the standard GPT analysis tools like GDFA.

4.9.2 Undueqfo

undueqfo(ECS,Nu,lamu,Bu,trim1,trim2) ;
Double focusing undulator with equal focusing in *x* and *y*.

ECS	Element Coordinate System.
Nu	Total number of undulator periods.
lamu	Undulator period [m]
Bu	Undulator field amplitude [T].
trim1	First matching coefficient (usually ¼).
trim2	Second matching coefficient (usually ¾).

The **B** field generated by this element is given by:

$$\mathbf{B} = \mathbf{Bu} F_s \begin{pmatrix} \sinh\left(\frac{k_u}{\sqrt{2}} x\right)\sinh\left(\frac{k_u}{\sqrt{2}} y\right)\sin(k_u z) \\ \cosh\left(\frac{k_u}{\sqrt{2}} x\right)\cosh\left(\frac{k_u}{\sqrt{2}} y\right)\sin(k_u z) \\ \sqrt{2}\cosh\left(\frac{k_u}{\sqrt{2}} x\right)\sinh\left(\frac{k_u}{\sqrt{2}} y\right)\cos(k_u z) \end{pmatrix} \quad [4.102]$$

where $k_u = \frac{2\pi}{\mathbf{lamu}}$ and $F_s = \mathbf{trim1}, \mathbf{trim2}, 1, 1, \ldots, 1, 1, \mathbf{trim2}, \mathbf{trim1}$ for each half undulator period.

To import realistic undulator fields, please see the **MAP3D_B** element.

4.9.3 Unduplan

unduplan(ECS,Nu,lamu,Bu,trim1,trim2) ;
Planar undulator.

ECS	Element Coordinate System.
Nu	Total number of undulator periods.
lamu	Undulator period [m].
Bu	Undulator field amplitude [T].
trim1	First matching coefficient (usually ¼).
trim2	Second matching coefficient (usually ¾).

The **B** field generated by this element is given by:

$$B = \mathbf{Bu} F_s \begin{pmatrix} 0 \\ \cosh(k_u y)\sin(k_u z) \\ \sinh(k_u y)\cos(k_u z) \end{pmatrix} \quad [4.103]$$

where $k_u = \frac{2\pi}{\mathbf{lamu}}$ and F_s is the shaping function for the first and the last half undulator period.

The dimensionless undulator strength *K* can be obtained using:

$$K = \left| \frac{q\mathbf{Bulamu}}{\sqrt{8}\pi mc} \right| \quad [4.104]$$

To import realistic undulator fields, please see the **MAP3D_B** element.

4.10 Scattering

Despite the name of this section, it contains a description of both boundary and scatter elements. In combination, they can be used to generate particles scattered off a surface, as described in section 1.11. Arbitrary diffraction patters can be simulated using the scatterbitmap element.

An aspect ratio of 1 is recommended when inspecting scattered trajectories. Without such an aspect-ratio all angles will be deformed, making interpretation much harder.

4.10.1 ForwardScatter

`forwardscatter(ECS,name,P,[nmin]) ;`
Scatter a particle in the forward direction with probability P.

`ECS`	Element Coordinate System.
`name`	Name of the surface.
`P`	Scatter probability.
`nmin`	Minimum nmacro.

A specularly reflected macro-particle is generated with the same energy as the incident particle in the forward direction. The new number of elementary particles this macro-particle represent is **`P*nmacro`**. When this **`P*nmacro<nmin`**, the new particle is not generated. Scatter output is written in the outputfile relative to the specified **`ECS`**.

One of the applications of the boundary elements is to remove particles using the **`forwardscatter`** element with a scattering probability of 0. This is demonstrated in the following example:

```
forwardscatter("wcs","I","remove", 0) ;
scatteriris("wcs","z", 1, 0.5,2 ) scatter="remove" ;
```

4.10.2 CopperScatter

`copperscatter(ECS,name,nmin,nmax) ;`
Scatter a particle from a copper surface.
The copperscatter element uses extrapolated data not yet experimentally confirmed.

`ECS`	Element Coordinate System.
`name`	Name of the surface.
`nmin`	Minimum nmacro, typically 1% of the initial nmacro.
`nmax`	Maximum nmacro, typically ignored by setting to 200% of the initial nmacro.

The probability for forward scattering is 1/3, the chance for backward is 2/3. The overall reflection chance is the product of the angle of incidence dependence and the dependence on the kinetic energy of the incident particle, as shown in Figure 4–17 and Figure 4–18. This overall reflection chance determines the **`nmacro`** of the reflected particle as calculated in **`forwardscatter`**. When the product of the scattering probabilities times the **`nmacro`** of the incident particle is below **`nmin`**, no new particle is generated. The total kinetic energy of the reflected particle varies uniformly between 45% and 95% of the kinetic energy of the incident particle.

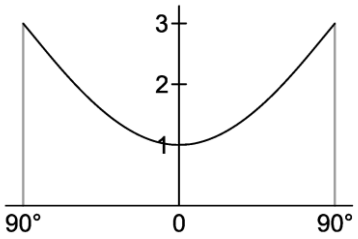


Figure 4–17: Reflection chance as function of angle of incidence. Zero angle corresponds to normal incidence.

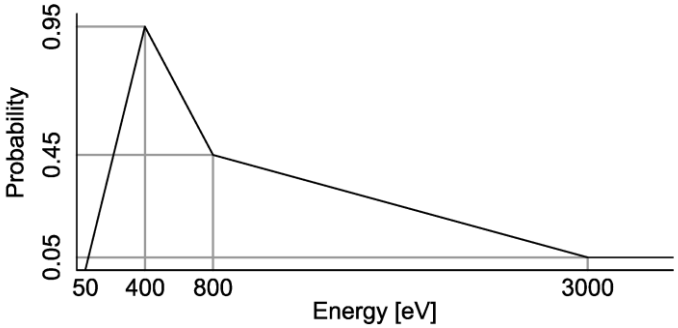


Figure 4–18: Reflection chance as function of incoming kinetic energy.

The simple function for the reflection chance as function of incoming kinetic energy comes form an extrapolaton of the data fom [24]. Custom scater routines can be written, as is explained in the Programmer's Reference.

4.10.3 ScatterBitmap

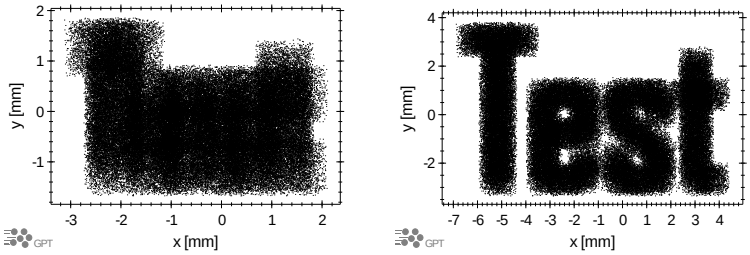
`scatterbitmap(ECS,name,filename,xres,yres) ;`
Scatter a particle with an arbitrary distribution as specified in a bitmap.

`ECS`	Element Coordinate System.
`name`	Name of the surface.
`filename`	Filename of a grayscale bitmap file (.bmp) containing the desired momentum distribution.
`xres`	Normalized x-momentum change per pixel.
`yres`	Normalized y-momentum change per pixel.

When a particle passed through a 'scatterbitmap' surface, the normalized horizontal and vertical momenta are changed according to the probability function given in **`filename`**. This can be used, for example, to model a diffraction process where the bitmap contains white-values representing the angular probability function for the diffraction process under investigation. In the example below, we use a simple bitmap contaning the text 'test' in white.

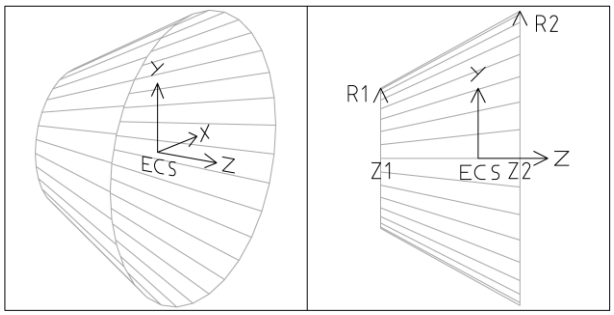
Example: GPT inputfile diffracting a 1x1 mm square input beam off a 'test' bitmap.

```
# setparticles("beam",10000,me,qe,0) ;

setxdist("beam","u",0,1e-3) ;
setydist("beam","u",0,1e-3) ;

setGdist("beam","u",1.2,0) ;

scatterbitmap("wcs","I","diffract","test.bmp",0.001,0.001) ;
scatterplate("wcs","z",0.01,1,1) scatter="diffract" ;

tout(0,1e-9,0.01e-9) ;
```



4.10.4 ScatterCone

scattercone(ECS,z1,R1,z2,R2) scatter="sm" ;
Cone-shaped scattering surface.



ECS	Element Coordinate System.
z1	*z*-position [m] of radius **R1**.
R1	Radius [m] of cone at *z*=**z1** position.
z2	*z*-position [m] of radius **R2**.
R2	Radius [m] of cone at *z*=**z2** position.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin cone surface. The particles are scattered according to the scattering model.

The equation for the cone boundary is given by:

$$x^2 + y^2 - \left(\mathbf{R1} + \frac{\mathbf{R2} - \mathbf{R1}}{\mathbf{z2} - \mathbf{z1}} (z - \mathbf{z1}) \right)^2 = 0 \qquad [4.105]$$

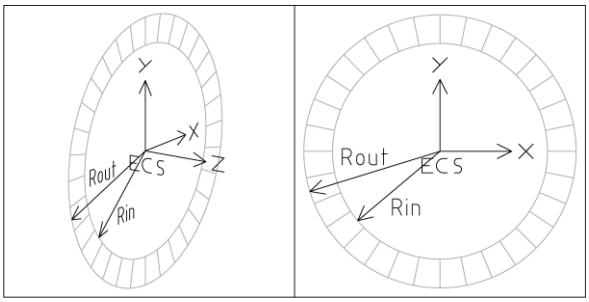
Example: GPT inputfile with cone surface, 50% forward scattering probability. Plot zx.

```
# Demonstration inputfile for cone scattering element
setstartline("beam",50, -2.5,0,0, 2.5,0,0, 0,0,1) ;

forwardscatter("wcs","I","scat", 0.5) ;
scattercone("wcs","I", 1,0.5, 2.0,1.5 ) scatter="scat" ;

tout(0, 6/c, 0.06/c) ;
```

4.10.5 ScatterIris

scatteriris(ECS,Rin,Rout) scatter="sm" ;

Iris- or disk-shaped scattering surface.



ECS	Element coordinate system.
Rin	Inner radius [m] of disk.
Rout	Outer radius [m] of disk.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin plate surface around the opening. The particles are scattered according to the scattering model. The normal of the disk is the z-axis.

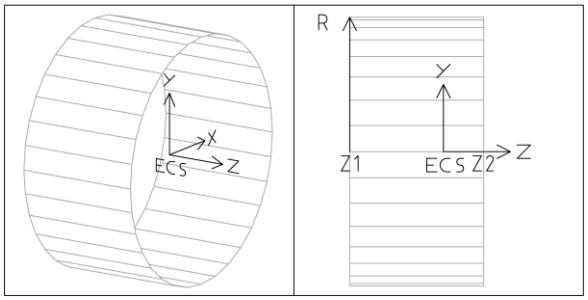
The description of the plate boundary is given by:
$z = 0$ if r>Rin and r<Rout.

Example: GPT inputfile with iris surface, particles are removed. Plot zx.

```
1. # Demonstration inputfile for iris scattering element
2. setstartline("beam",50, -2.5,0,0, 2.5,0,0, 0,0,1) ;
3.
4. forwardscatter("wcs","I","remove", 0) ;
5. scatteriris("wcs","z", 1, 0.5,2 ) scatter="remove" ;
6.
7. tout(0, 4/c, 0.04/c) ;
```

4.10.6 ScatterPipe

scatterpipe(ECS,z1,z2,R) scatter="sm" ;
Pipe-shaped scattering surface.



ECS	Element Coordinate System.
z1	*z*-position [m] where the pipe begins.
z2	*z*-position [m] where the pipe ends.
R	Radius [m] of pipe.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin pipe surface. The particles are scattered according to the scattering model.

The description of the cone boundary is given by:

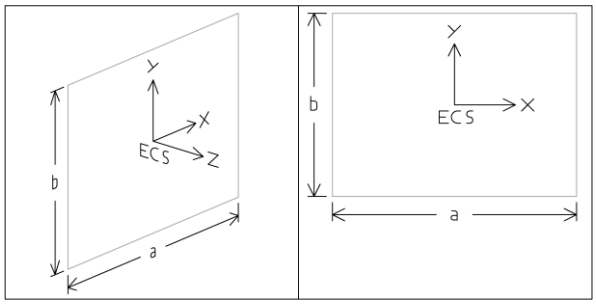
$$x^2 + y^2 - \mathbf{R}^2 = 0 \qquad [4.106]$$

Example: GPT inputfile with pipe surface, 50% scattering probability. Plot xy.

```
1.  # Demonstration inputfile for pipe scattering element
2.  setstartline("beam",50, -2,2,0, 2,2,0, 0,-1,0) ;
3.
4.  forwardscatter("wcs","I","scat", 0.5) ;
5.  scatterpipe("wcs","I", -1,1, 1 ) scatter="scat" ;
6.
7.  tout(0, 4/c, 0.04/c) ;
```

4.10.7 ScatterPlate

scatterplate(ECS,a,b) scatter=”sm” ;
Plate-shaped scattering surface.



ECS	Element Coordinate System.
a	Total plate length in x-direction.
b	Total plate length in y-direction.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin plate surface. The particles are scattered according to the scattering model. The normal of the plate is the z-axis.

The description of the plate boundary is given by:

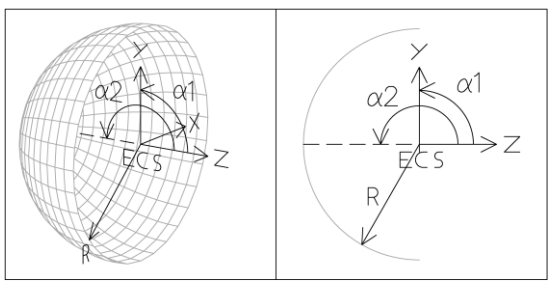
$$z = 0 \qquad [4.107]$$

Example: GPT inputfile with plate surface, particles are removed. Plot xy.

```
1.  # Demonstration inputfile for plate scattering element
2.
3.  setparticles("beam",1000,me,qe,0.0) ;
4.  setxdist("beam","u",0,2) ;
5.  setydist("beam","u",0,2) ;
6.  setGdist("beam","u",10,1) ;
7.
8.  forwardscatter("wcs","I","remove", 0.0) ;
9.  scatterplate("wcs","z",1, 1,1 ) scatter="remove" ;
10.
11. tout(2/c) ;
```

4.10.8 ScatterSphere

scattersphere(ECS,R,[a1,a2]) scatter=”sm” ;
Sphere-shaped scattering surface.



ECS	Element Coordinate System.
R	Radius of sphere.
a1	Start angle [rad] measured ccw from the **z**-direction in the zR-plane.
a2	End angle [rad] measured ccw from the **z**-direction in the zR-plane.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin sphere surface. The particles are scattered according to the scattering model. The symmetry axis of the sphere is the z-axis.

The description of the sphere boundary is given by:

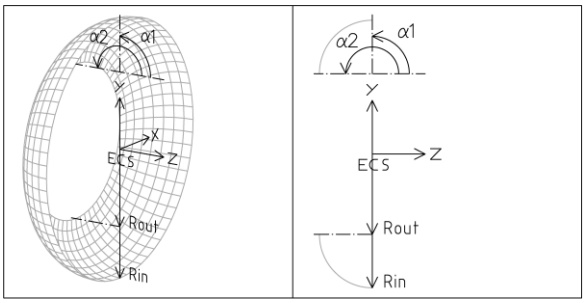
$$x^2 + y^2 + z^2 - \mathbf{R}^2 = 0 \qquad [4.108]$$

Example: GPT inputfile with sphere surface, 50% scattering probability. Plot zx.

```
1. # Demonstration inputfile for an open sphere scattering element
2. setstartline("beam",50, 2,0,-2, 2,0,2, -1,0,0) ;
3.
4. forwardscatter("wcs","I","scat", 0.5) ;
5. scattersphere("wcs","I", 1, pi/2,pi ) scatter="scat" ;
6.
7. tout(0, 4/c, 0.04/c) ;
```

4.10.9 ScatterTorus

scattertorus(ECS,Rout,Rin,[a1,a2]) scatter="sm" ;
Torus-shaped scattering surface.



ECS	Element Coordinate System.
Rout	Outer radius [m] of torus.
Rin	Inner radius [m] of torus.
a1	Start angle [rad] measured ccw from the **z**-direction in the zr-plane.
a2	End angle [rad] measured ccw from the **z**-direction in the zr-plane.
sm	Scatter model to be used.

New particles are generated when an incident particle passes through the infinitely thin torus surface. The particles are scattered according to the scattering model.

The description of the torus boundary is given by:

$$a^4 - 2a^2(x^2 + y^2 - z^2 + b^2) + (x^2 + y^2 + z^2 - b^2)^2 = 0 \qquad [4.109]$$

where a is the outer radius and b is the inner radius of the torus.

Example: GPT inputfile with torus surface, 50% scattering probability. Plot zx.

```
1. # Demonstration inputfile for torus scattering element
2. setstartline("beam",50, 3,0,-2, 3,0,2, -1,0,0) ;
3.
4. forwardscatter("wcs","I","scat", 0.5) ;
5. scattertorus("wcs","I", 1.1,1.0, 0,pi/2 ) scatter="scat" ;
6.
7. tout(0, 4/c, 0.04/c) ;
```

4.11 Miscellaneous

This section contains all built-in elements not listed in the above sections.

4.11.1 CCS

`ccs(ECS,CCSname) ;`
Defines a Custom Coordinate System.

`ECS`	The coordinate system transform.
`CCSname`	The name of the Custom Coordinate System.

See section 1.4.3 for the description of Custom Coordinate Systems.

4.11.2 CCSflip

`ccsflip(ECS,CCSname) ;`
Refer all particles to another axis.

`ECS`	The coordinate system transform.
`CCSname`	The name of the Custom Coordinate System to transfer the particle to.

All particles coordinates are refered to the axis named **`CCSname`** when $z>0$ in the specified ECS.

See section 1.4.3 for the description of Custom Coordinate Systems.

4.11.3 Collision

`collision() ;`
Detect particle-particle collisions and start a new 'combined' particle.

At the end of every successful timestep collisions between two particles are detected when:

$$\left|\mathbf{r}_i - \mathbf{r}_j\right|^2 < R_j^2 \qquad [4.110]$$

When a collision is detected, both particles are removed and a new particle is created. The new position is equal to the mass-weighted average position. The new momentum is the sum of the moments of the initial particles:

$$\begin{aligned} \mathbf{r}_{new} &= (n_i m_i \mathbf{r}_i + n_j m_j \mathbf{r}_j)/(n_i m_i + n_j m_j) \\ \mathbf{p}_{new} &= \mathbf{p}_1 + \mathbf{p}_2 \Rightarrow \gamma\beta_{new} = (n_i m_i \gamma\beta_i + n_j m_j \gamma\beta_j)/(n_i m_i + n_j m_j) \end{aligned} \qquad [4.111]$$

The new mass and charge are the sum of the mass and charge of the original particles as given by:

$$\begin{aligned} n_{new} &= n_i + n_j \\ q_{new} &= (n_i q_i + n_j q_j)/n_{new} \\ m_{new} &= (n_i m_i + n_j m_j)/n_{new} \end{aligned} \qquad [4.112]$$

4.11.4 Drift

drift(ECS,L,[R]) ;
Drift section.

ECS	Element Coordinate System.
L	Length of the drift section [m].
R	Radius [m]. When this parameter is specified, the particle is removed when $x^2 + y^2 > \mathbf{R}^2$.

This element is 'active' when $|z| < \frac{1}{2}\mathbf{L}$. It does nothing except optionally remove a particle when it is too far from the axis. If your set up has large sections without elements it is recommended to use **drift**. It can speed up calculation time because all the other local elements on the axis don't have to be checked each timestep.

Although this element can be replaced by a **rmax**, it is recommended to use **drift** when possible because it is local and therefore faster.

4.11.5 Gravity

gravity(gx,gy,gz) ;
Calculates uniform relativistic gravity force

gx	Gravitational acceleration [m/s^2] in x-direction.
gy	Gravitational acceleration [m/s^2] in y-direction.
gz	Gravitational acceleration [m/s^2] in z-direction.

For non-relativistic velocities, the gravitational field is simply given by:

$$\mathbf{F} = m\mathbf{g} \qquad [4.113]$$

where **g** is the gravitational acceleration. For relativistic particles however, the more complicated equation from the general theory of relativity must be used:

$$\mathbf{F} = \gamma m\left[(1+\boldsymbol{\beta}^2)\,\mathbf{g} - (\boldsymbol{\beta}\cdot\mathbf{g})\,\boldsymbol{\beta}\right] \qquad [4.114]$$

The equation is derived from the relativistic gravity force of a spherically symmetric object with mass *M* given by:

$$\mathbf{F} = -GM\frac{E}{c^2}\frac{(1+\boldsymbol{\beta}^2)\,r - (\boldsymbol{\beta}\cdot\mathbf{r})\,\boldsymbol{\beta}}{|r|^3} \qquad [4.115]$$

where *G* is the gravitational constant and *E* is the particle energy γmc^2. When the particle displacement is much smaller than *r*, the spherical field can be approximated by a uniform field using $\mathbf{g} = -GM\mathbf{r}/|r|^3$.

4.11.6 Randomize

randomize([seed]) ;
Initialize the GPT random-number generator.

seed	Number indicating the starting point in the sequence of random numbers generated by GPT. The system time is used when omitted.

Without randomization, all pseudorandom sequences will reproduce from run to run. Using randomize you can specify a different **seed** for a different, but reproducing, sequence. When **seed** is not specified, the system time is used to assure every GPT run uses a different sequence.

When **randomize** is used in combination with the random-ordering specification (**~**) of the initial particle distributions, please make sure it appears before the corresponding **set...dist** element(s).

4.11.7 Run

run(program, [param1, param2, ...]) ;
Runs an external command.

program Command to be executed.
param*N* Optional parameter(s).

Using **run**, any external program can be executed that doesn't require interactive (keyboard) input. When additional parameters are specified, they are separated by spaces and passed to the program. For example, the following lines will delete the file **filename** in the current directory.

```
run("del", "filename") ;  # Windows
run("rm" , "filename") ;  # UNIX
```

Variables, expressions and the redirection operators '**<**', '**|**' and '**>**' work as usual. For example the following code fragment runs **program -2 3.141592653589793 /options** and writes the output to the file **program.log**.

```
1. a = -2 ;
2. b = pi ;
3. run("program ",a,b, "/options", ">program.log")
```

Obviously, when specific UNIX or DOS commands are specified, the GPT inputfile is not portable anymore. Furthermore, only command-line programs are supported by the GPTwin graphical user interface.

4.11.8 Viscosity

viscosity(eta) ;
Calculates viscosity forces

eta viscosity [Pa s]: η.

This element adds a linear viscosity force as function of particle velocity. The viscosity force acts on all particles and is given by:

$$\mathbf{F} = -6\pi\eta R\,\boldsymbol{\beta} c \qquad [4.116]$$

The radius of the particles R can be set using the **setrmacrodist** keyword. Typical **eta** values are 17.1 10^{-6} for air and 1.00 10^{-3} for water at standard pressure and room temperature.

4.12 Remove particles

The elements listed in this section remove particles when inside a specific volume. No scattered particles are generated.

An alternative to these elements is to use the boundary elements of section 4.10 to define a boundary and use **forwardscatter** with a probability of 0 to remove the particle without generating a scattered particle.

4.12.1 Gminmax

Gminmax(t,Gmin,Gmax) ;
Removes all particles with a Lorentz factor outside the specified range.

t	Simulation time [s].
Gmin	Minimum Lorentz factor.
Gmax	Maximum Lorentz factor.

This element is only active when the simulation time has passed **t**. From that time on, it removes all particles with a Lorentz factor outside the range bracketed by **Gmin** and **Gmax**.

4.12.2 Multislit

multislit(ECS,a,b,L,d,N) ;
Removes a particle if it does not pass through an opening of a multiple slit.

ECS	Element Coordinate System.
a	Width of a slit opening [m] (in **x**-direction).
b	Height of a slit opening [m] (in **y**-direction).
L	Length of the element [m] (in **z**-direction).
d	Distance between the centers of two slit openings [m].
N	Number of slits.

This element simulates a multiple slit. The particles can only pass through the openings of the slit, all other particles are removed if $|z| < \frac{1}{2}\mathbf{L}$. This element is local.

4.12.3 Rmax

rmax(ECS,R,[L]) ;
Removes a particle when it is too far from the axis.

ECS	Element Coordinate System.
R	Maximum radius [m].
L	Length of the element [m]. If not specified, ∞ is assumed.

The particle is removed when: $x^2 + y^2 > \mathbf{R}^2$ and $|z| < \frac{1}{2}\mathbf{L}$.

Although **L** can be specified, this element is global.

4.12.4 StdXYZmax

stdxyzmax(Nstdx,Nstdy,Nstdz) ;
Removes particles too far off the center of the bunch.

Nstdx	Allowed number of standard deviations in x. Ignored when zero.
Nstdy	Allowed number of standard deviations in y. Ignored when zero.
Nstdz	Allowed number of standard deviations in z. Ignored when zero.

Every particle located at least **Nstdx**, **Nstdy** or **Nstdz** standard deviations away from the average *x*-, *y*- or *z*-coordinate of the bunch is removed.

This element is intended to 'clean up' the simulation by removing all outliers, all the particles located too far from the center of the bunch. This can be used to solve the following problems:

- Outliers can affect the GDFA data analysis routines and cause unrealistic values for bunch length, energy spread etc.
- Outliers can significantly increase the simulation time for a variety of reasons.

Typical values for **Nstdx**, **Nstdy** and **Nstdz** are between 4 and 6.

Particle *i* is removed if one of the following conditions is true:

$$\begin{aligned} &\mathbf{Nstdx} \neq 0 \wedge (x_i < \bar{x} - \mathbf{Nstdx}\ stdx \vee x_i > \bar{x} + \mathbf{Nstdx}\ stdx) \\ &\mathbf{Nstdy} \neq 0 \wedge (y_i < \bar{y} - \mathbf{Nstdy}\ stdy \vee y_i > \bar{y} + \mathbf{Nstdy}\ stdy) \\ &\mathbf{Nstdz} \neq 0 \wedge (z_i < \bar{z} - \mathbf{Nstdz}\ stdz \vee z_i > \bar{z} + \mathbf{Nstdz}\ stdz) \end{aligned} \qquad [4.117]$$

where

$$\begin{aligned} stdx &= \sqrt{\overline{(x-\bar{x})^2}} \\ stdy &= \sqrt{\overline{(y-\bar{y})^2}} \\ stdz &= \sqrt{\overline{(z-\bar{z})^2}} \end{aligned} \qquad [4.118]$$

4.12.5 XYmax

xymax(ECS,a,b,[L]) ;
Removes a particle when it is too far from the axis.

ECS	Element Coordinate System.
a	Maximum **x**-deviation [m].
b	Maximum **y**-deviation [m].
L	Length of the element [m]. If not specified, ∞ is assumed.

The particle is removed when: $|x| > \frac{1}{2}\mathbf{a}$ or $|y| > \frac{1}{2}\mathbf{b}$ if $|z| < \frac{1}{2}\mathbf{L}$.

Although **L** can be specified, this element is global.

4.12.6 Zminmax

zminmax(ECS,zmin,zmax) ;
Removes all particles in front of zmin or behind zmax.

ECS	Element Coordinate System.
zmin	Minimum *z*-coordinate [m].
zmax	Maximum *z*-coordinate [m].

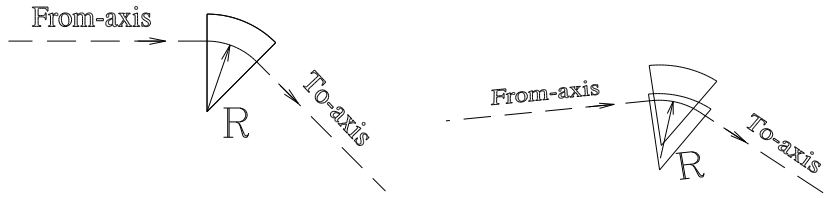
The particle is removed when $z<$**zmin** or $z>$**zmax**.

4.13 Obsolete elements

This section contains variables and elements maintained for backward compatibility. Please use the indicated alternatives when creating a new GPT inputfile.

4.13.1 Bend

`bend(fromCCS,toCCS,R,Bfield) ;`
Bending magnet.
Obsolete element, please use **`sectormagnet`** instead.



`fromCCS`	Name of the CCS the particles start on.
`toCCS`	Name of the CCS the particles are bended to.
`R`	Radius of the bend [m].
`Bfield`	**B** field in the bending magnet [T]. Normally this is $\pm$**`m`**c$\gamma\beta$/**`qR`**.

This element is an idealized bending magnet with a **B** field perpendicular to the **z**-directions of both the **`fromCCS`** and **`toCCS`**. The geometry is that of a wedge with the two straight sides perpendicular to the **z**-directions of the axes. When the axes do not intersect or when they are parallel, this element can not be used.

We define **d** and **e** as the **z**-directions and **C** as the intersection of the **`fromCCS`** and **`toCCS`**. The **B** field generated by this element5F[6] is given by:

$$\mathbf{B} = \begin{cases} \texttt{Bfield}\,\dfrac{\mathbf{d}\times\mathbf{e}}{|\mathbf{d}\times\mathbf{e}|} & \text{if } \mathbf{r}\cdot\mathbf{d}-\mathbf{C}\cdot\mathbf{d} > -\dfrac{|\mathbf{d}\times\mathbf{e}|}{1+\mathbf{d}\cdot\mathbf{e}}\texttt{R} \text{ and } \mathbf{r}\cdot\mathbf{e}-\mathbf{C}\cdot\mathbf{e} > -\dfrac{|\mathbf{d}\times\mathbf{e}|}{1+\mathbf{d}\cdot\mathbf{e}}\texttt{R} \\ 0 & \text{otherwise} \end{cases} \qquad [4.119]$$

In other words: It is the CCS specification that defines the geometry of the magnet, not the magnetic field.

4.13.2 Bz

`bz(ECS,filename,I) ;`
Axial global Bz-field.

[6] In reality, half a bending magnet is constructed on the **`fromaxis`** and the other half on the **`toaxis`**. When a particle leaves one half of the bending magnet it is placed on the other axis to go through the remaining part of the bend.

Obsolete element: Please use **map1D_B** instead.

ECS Element Coordinate System.

filename File with B_z samples of the **B** field along the **z**-axis in [T]. The file must have the following layout:

# Comment		Optional
N		Number of given **Bz** samples
$\mathbf{z}_1$	$\mathbf{Bz}_1$	z: **z** positions of samples
$\mathbf{z}_2$	$\mathbf{Bz}_2$	Bz: **B** field in **z** direction at (0,0,z)
$\mathbf{z}_3$	$\mathbf{Bz}_3$	
⋮	⋮	
$\mathbf{z}_N$	$\mathbf{Bz}_N$	

I Multiplication factor for **B**.

The B_z samples along the **z**-axis are specified in an external file. The **B** field generated by this element is given by:

$$\mathbf{B} = \mathbf{I}\left(-\frac{x}{2}\frac{dB_z}{dz}, -\frac{y}{2}\frac{dB_z}{dz}, B_z\right) \qquad [4.120]$$

These fields are only correct near the **z**-axis of the element. A so-called 'natural cubic spline' is calculated from the given (**z**,**Bz**) points. It is recommended to specify more points at positions where the first derivative fluctuates significantly.

4.13.3 Ezcell

ezcell(ECS,filename,Ezef,phi,w) ;

Standing wave resonant cavity.

Obsolete element: Please use **map1D_TM** instead.

ECS Element Coordinate System.

filename File with E_z samples of the **E** field along the **z**-axis. The file must have the following layout:

# Comment		Optional
N		Number of given **Ez** samples
$\mathbf{z}_1$	$\mathbf{Ez}_1$	z: **z**-positions of samples
$\mathbf{z}_2$	$\mathbf{Ez}_2$	Ez: **E** field in **z**-direction at (0,0,z)
$\mathbf{z}_3$	$\mathbf{Ez}_3$	
⋮	⋮	
$\mathbf{z}_N$	$\mathbf{Ez}_N$	

Ezef Multiplication factor.

phi Phase factor: ϕ in radians.

w Angular frequency: ω.in [s^{-1}].

This element can be used to simulate a standing wave resonant cavity where the actual E_z field is known. It has been designed to simulate a single-cell prebuncher.

The fields generated by this element in cylindrical coordinates are given by:

$$\begin{aligned} E_z &= \mathbf{Ez}\,\mathbf{Ezef}\cos(\omega t+\phi) \\ E_r &= -\frac{r}{2}\frac{d\,\mathbf{Ez}}{dz}\mathbf{Ezef}\cos(\omega t+\phi) \\ B_\phi &= \frac{r\omega\,\mathbf{Ez}}{2c^2}\mathbf{Ezef}\sin(\omega t+\phi) \end{aligned} \qquad [4.121]$$

The fields are only correct near the **z**-axis of the element. A so-called 'natural cubic spline' is calculated from the given (**z**,**Ez**) points. It is recommended to specify more points at the positions where the first derivative fluctuates significantly.

Combining the Maxwell equations in vacuum $\nabla \times \mathbf{E} = -\partial B / \partial t$ and $\nabla \times \mathbf{B} = \mu_0 \varepsilon_0 \, \partial \mathbf{E} / \partial t$ imposes the following restriction on the sampled electric field profile on axis: $\omega^2 \, \mathbf{Ez} + c^2 \, \mathbf{Ez''} = 0$. Previous implementations of **ezcell** used the second derivative to obtain the B_φ field, producing less accurate results.

4.13.4 m

m=expression ;
Specifies the mass of an elementary particle.
Obsolete element: Please use **setparticles** instead.

expression Particle mass in kg. If not specified, the mass of an electron, 9.10953e-31 kg, is taken..

4.13.5 q

q=expression ;
Specifies the charge of an elementary particle.
Use **setparticles** instead.

expression Particle charge in Coulomb. If not specified, the charge of an electron, -1.6021892e-19 C is taken.

4.13.6 SetCathode

setcathode(set,Ra,Rc,kT) ;
Modify a particle set according to a spherical cathode.
Obsolete element: Please use the new particle-set functions instead.

set Particle set.
Ra Aperture [m].
Rc Curvature radius [m]. Specify zero for a flat surface.
kT Temperature [eV], not yet implemented. Please use the general start elements.

This element modifies a particle distribution for the simulation of a spherical cathode as shown in Figure 4–19.

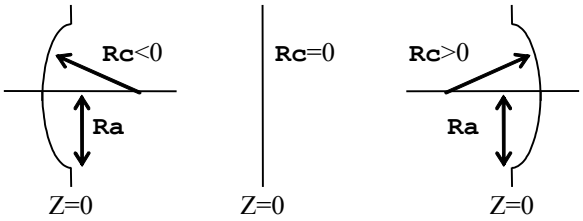


Figure 4–19: Spherical cathode for negatice, zero and positive **Rc**.

The particle coordinates in real space are modified by:

$$z_i = z_i \pm \left(\sqrt{Rc^2 - r_i^2} - \sqrt{Rc^2 - Ra^2} \right) \qquad [4.122]$$

where the + indicates a positive **Rc**, the – indicates a negative **Rc**. All particles with a radius larger than **Ra** are removed from the particle set.

The **startcathode** element must be used to generate the particles as function of time. When **startcathode** is not used, all particles will be started at t=0.

4.13.7 Setrmacro

setrmacro(set,radius) ;
Sets the radius of the macro-particles in a set.
Obsolete element: Please use **setrmacrodist** instead.

set	Particle set.
radius	Radius of particles [m].

This element can be used to specify the radius of the particles in a set.

4.13.8 StartCathode

startcathode(ECS,set, Ra,Rc,Lc, dzdt,dtmax) ;

Continuously add particles emerging from a spherical or flat surface during the simulation. For photo-cathode simulations, a curved laser front can be taken into account.
Obsolete element: Please use the new particle-set functions instead.

ECS	Element Coordinate System.
set	Particle set.
Ra	Aperture [m].
Rc	Cathode curvature radius [m]. Specify zero for a flat surface. When not zero, the **setcathode** element must be used before.
Lc	Laser curvature radius [m]. Specify zero for a flat laser front or when not applicable.
dzdt	Linear transformation between z-position and time in [m/s], may be negative.
dtmax	Maximum allowed extrapolation [s]. Specify 0 to start all particles one-by-one. A larger value speeds up the simulation, at the cost of accuracy.

This element can be used for the simulation of a spherical cathode or a continuous beam. It starts particles during the simulation according to a distribution, as defined using the regular start and set elements. However, the *z*-coordinate is used as time coordinate by dividing by **dzdt**. When simulating a spherical cathode, the previous element must be **setcathode**, with identical **Ra** and **Rc** parameters. When a photo-cathode is simulated, an optional curvature of the laser front, **Lc**, can be specified to account for the difference in arrival time between the center and the edge of the laser pulse on the cathode.

The 2.4 **spacecharge** routine should never be used in combination with **startcathode**. Instead, use **spacecharge3D** in the following way:

```
1. setrmacro(macro-particle-radius) ;
2. spacecharge3D() ;
```

For a flat surface and laser front, all particles emerge at the z=0 plane of the ECS of **startcathode**, but only one particle is started at t=0. The other particles follow at later times t_i, depending on their z-distribution and **dzdt**, the transformation between position and time:

$$t_i = \frac{z_{\max} - z_i}{\mathbf{dzdt}} \qquad [4.123]$$

The energy of an individual particle will not influence the moment that it is added to the simulation. It is as if the previously specified bunch moves with constant longitudinal velocity **dzdt** towards the cathode. When a particle crosses the boundary, it emerges with the specified velocities. When a particle is added to the simulation, a small extrapolation is performed to compensate for the fact that timesteps normally don't coincide with precise plane boundaries:

$$\begin{aligned} \Delta t_i &= t - t_i \\ x_i &= x_i + \Delta t_i c\beta x_i \\ y_i &= y_i + \Delta t_i c\beta y_i \\ z_i &= \Delta t_i c\beta x_i \end{aligned} \qquad [4.124]$$

Finally r and βr are transformed to the ECS of **startcathode**.

When the laser front is not flat, i.e. **Lc** is not zero, **startcathode** accounts for the different arrival time of the light, as shown in Figure 4–20. A positive **Lc** indicates that the center of the beam is illuminated first and a zero **Lc** indicates a flat laser front. The equations are modified accordingly:

$$t_i = \frac{z_{\max} - z_i}{\mathbf{dzdt}} \mp \frac{\left(\sqrt{Lc^2 - r_i^2} - \sqrt{Lc^2 - La^2}\right)}{c} \qquad [4.125]$$

The equations for a spherical cathode are analogous to those of a flat surface. The only difference is that particles are not positioned at the z=0 plane of the ECS of **startcathode**, but at the spherical cathode boundary:

$$z_i = \pm\left(\sqrt{Rc^2 - Ra^2} - \sqrt{Rc^2 - r_i^2}\right)$$
$$t_i = \frac{(z_{\max} - z_i) \mp \left(\sqrt{Rc^2 - r_i^2} - \sqrt{Rc^2 - Ra^2}\right)}{\texttt{dzdt}} \mp \frac{\left(\sqrt{Lc^2 - r_i^2} - \sqrt{Lc^2 - La^2}\right) - \left(\sqrt{Rc^2 - r_i^2} - \sqrt{Rc^2 - Ra^2}\right)}{c} \quad [4.126]$$

where the signs of **Rc** and the final z-coordinates are identical, as shown in Figure 4–20.

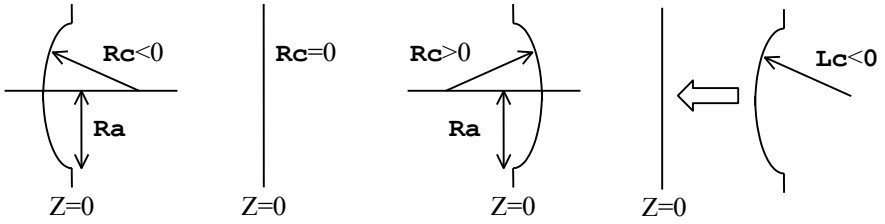


Figure 4–20: Spherical cathode for negative, zero and positive **Rc**.

4.13.9 Start

start(ECS,startfile) ;

Reads the initial coordinates of the macro-particles from a file. Particles are added to the particle set **"beam"**.

Obsolete element: Please use **setfile** instead.

ECS Element Coordinate System. The coordinates of the particles are interpreted relative to the ECS specified here. If no transformation is needed, the identity transformation must be specified: **"wcs","I".**

startfile File with the initial coordinates of the particles. If the file is a GPT outputfile with a **tout(t);** statement, this group must be specified by a **time=t;** statement above **start**. If the file is an ASCII file it must have the following layout:

# Comment						Optional
N						Number of particles
$\mathbf{x}_1$	$\mathbf{y}_1$	$\mathbf{z}_1$	$\boldsymbol{\beta}\mathbf{x}_1$	$\boldsymbol{\beta}\mathbf{y}_1$	$\boldsymbol{\beta}\mathbf{z}_1$	Particle 1
$\mathbf{x}_2$	$\mathbf{y}_2$	$\mathbf{z}_2$	$\boldsymbol{\beta}\mathbf{x}_2$	$\boldsymbol{\beta}\mathbf{y}_2$	$\boldsymbol{\beta}\mathbf{z}_2$	Particle 2
$\mathbf{x}_3$	$\mathbf{y}_3$	$\mathbf{z}_3$	$\boldsymbol{\beta}\mathbf{x}_3$	$\boldsymbol{\beta}\mathbf{y}_3$	$\boldsymbol{\beta}\mathbf{z}_3$	Particle 3
⋮	⋮	⋮	⋮	⋮	⋮	
$\mathbf{x}_N$	$\mathbf{y}_N$	$\mathbf{z}_N$	$\boldsymbol{\beta}\mathbf{x}_N$	$\boldsymbol{\beta}\mathbf{y}_N$	$\boldsymbol{\beta}\mathbf{z}_N$	Particle N

The initial positions of the particles are read from an external file. This can be the outputfile of a previous run or an ASCII file. The position and velocity are interpreted relative to the specified Element Coordinate System.

The correct way to instruct GPT to restart using the outputfile of a previous run is shown in Listing 4–3 and Listing 4–4.

Listing 4–3: **file1.in**: First inputfile writing output at t=3 to **file1.gdf**

```
1. startgrid( ... )
2. ...
3. tout(3) ;
```

Listing 4–4: **file2.in**: Second inputfile, reading the particle coordinates from **file1.gdf**

```
5. time=3 ;
6. start( "wcs","I", "file1.gdf" ) ;
7. ...
```

Outputfiles created by MR can also be used. Either use MR again with exactly the same parameter sweep or specify **var=value** above **start** to specify which group has to be used.

4.13.10 Startcyl

startcyl(ECS,nps,E0,Emod,sigE,R,div,divspr,Dmod,sigz/psi,[omega])]) ;

Specifies the initial distribution in real and velocity space of a cylindrical set of macro-particles. Particles are added to the particle set **"beam"**.
Obsolete element: Please use the new particle-set functions instead.

ECS	Element Coordinate System.
nps	Number of macro-particles. nps is truncated to enable a smooth distribution in xy-space. Possible values are 1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, ...
Eo	Mean total energy in eV.
Emod	String that determines the energy distribution function. **Emod** can be **"Uniform"** or **"Gaussian"**.
sigE	Energy spread in fraction of **Eo**. Total spread for a uniform distribution and 1/*e* for a Gaussian distribution.
R	Maximum beam radius.
div	Linear divergence with particle radius. A negative value is allowed.
divspr	Divergence randomly distributed among the particles.
Dmod	String that determines the density distribution function. **Dmod** can be **"Uniform"**, **"Gaussian"** or **"Class-C"**.
sigz	Bunch length: Total for a uniform distribution and 1/*e* for a Gaussian distribution.
psi	Half conduction angle for Class-C modulation.
omega	Modulation angular frequency for Class-C modulation.

The parameters represent a bunch as produced by an electron gun. The default gun orientation is in the z-direction but can be reoriented using the ECS.

The longitudinal velocity distribution can be chosen so that the energy distribution is either **"Uniform"** or **"Gaussian"**. The corresponding probability density functions are respectively given by:

Uniform6F[7]:
$$\begin{cases} \frac{N}{\sigma_E} & \text{if} -\frac{1}{2}\sigma_E \le E - E_0 \le \frac{1}{2}\sigma_E \\ 0 & \text{otherwise} \end{cases}$$

Gaussian:
$$\frac{N}{\sqrt{2\pi}\sigma}\exp\left(-\frac{z^2}{2\sigma^2}\right)$$

The transverse distribution in real space is a homogeneous cylinder with radius **R**.

The transverse distribution in velocity space consists of two independent divergences:
- A divergence **div** linear to the radius of the particle: *vr*=**div** (**R**/*r*). It only rotates the phase-space ellipse and therefore does not contribute to the emittance.
- A divergence uniformly distributed in the range [0,**divspr**] which is randomly assigned to the particles. The resulting bunch will have an emittance of **R divspr**.

The longitudinal spatial distribution can be **"Uniform"** (equidistant), **"Gaussian"** or **"Class-C"**. The corresponding probability density functions are respectively given by:

Uniform[7]:
$$\begin{cases} \frac{N}{\sigma_z} & \text{if} -\frac{1}{2}\sigma_z \le z \le \frac{1}{2}\sigma_z \\ 0 & \text{otherwise} \end{cases}$$

Gaussian:
$$\frac{N}{\sqrt{2\pi}\sigma_z}\exp\left(-\frac{z^2}{2\sigma_z^2}\right)$$

Class-C[8]:
$$\begin{cases} \frac{Nk_b\left(\cos(k_b z)-\cos(\psi)\right)}{2\left(\sin(\psi)-\psi\cos(\psi)\right)} & \text{if} -\frac{\psi}{k_b} \le z \le \frac{\psi}{k_b} \\ 0 & \text{otherwise} \end{cases}$$
$$k_b = \omega_{gun} v_z$$

[7] Please note that the total distribution length is specified as σ.
[8] The bunch length can be calculated using $L=2\psi/k_b$.

This element does not position the particles (pseudo) randomly. The used algorithm positions the particles as homogeneous as possible in both real and phase space. Therefore much fewer particles are needed to obtain the same statistics as when using randomly distributed particles. This generally saves GPT a considerable amount of CPU time.

4.13.11 Startgrid

`startgrid(ECS,Lx,dx,Ly,dy,Lz,dz,[Bx,By,Bz]) ;`
Starts the macro-particles in a rectangular grid with equal velocity. Particles are added to the particle set **`"beam"`**.
Obsolete element: Please use **`setstartxyzgrid`** instead.

`ECS`	Element Coordinate System. The coordinates of the particles are interpreted relative to the ECS specified here. If no transformation is needed, the identity transformation must be specified: **`"wcs","I"`**
`Lx`	Length of array in x-direction. This length may be zero.
`dx`	Spacing in x-direction.
`Ly`	Length of array in y-direction. This length may be zero.
`dy`	Spacing in **y**-direction.
`Lz`	Length of array in z-direction. This length may be zero.
`dz`	Spacing in z-direction.
`Bx`	Velocities/c of the particles in x-direction. Default 0.
`By`	Velocities/c of the particles in y-direction. Default 0.
`Bz`	Velocities/c of the particles in z-direction. Default 0.

The particles start in a rectangular grid centered around the origin. The grid extends from –**`Lx`**/2,–**`Ly`**/2, –**`Lz`**/2 to **`Lx`**/2,**`Ly`**/2,**`Lz`**/2 with spacing **`dx`**,**`dy`**,**`dz`**. The orientation and origin can be modified using the ECS.

It is simple to instruct GPT to initialize billions of particles using this keyword. Since GPT has no built-in limit for the number of particles, it must rely on the operating system to indicate an 'out of memory' condition. On some virtual memory machines, this error message can come too late, causing either GPT or the complete system to crash.

4.13.12 Startpar

`startpar(ECS,x,y,z,Bx,By,Bz) ;`
Specifies the initial position and velocity of a single macro-particle. Particles are added to the particle set **`"beam"`**.
Obsolete element: Please use **`setstartpar`** instead.

`ECS`	Element Coordinate System. The coordinates of the particle are interpreted relative to the ECS specified here. If no transformation is needed, the identity transformation must be specified: **`"wcs","I".`**
`x`	*x*-coordinate of the particle.
`y`	*y*-coordinate of the particle.
`z`	*z*-coordinate of the particle.
`Bx`	Velocity/c of the particle in the **x**-direction.
`By`	Velocity/c of the particle in the **y**-direction.
`Bz`	Velocity/c of the particle in the **z**-direction.

The ECS is obsolete here because the **`x`**,**`y`**,**`z`** and **`Bx`**,**`By`**,**`Bz`** parameters are sufficient to specify any position and any velocity. It is introduced for compatibility with other **`start`** functions.

4.13.13 StartpGB

`startpargb(ECS,x,y,z,GBx,GBy,GBz) ;`
Specifies the initial position and velocity of a single macro-particle. Particles are added to the particle set **`"beam"`**.
Obsolete element: Please use **`setstartpar`** instead.

ECS	Element Coordinate System. The coordinates of the particle are interpreted relative to the ECS specified here. If no transformation is needed, the identity transformation must be specified: **"wcs","I".**
x	*x*-coordinate of the particle.
y	*y*-coordinate of the particle.
z	*z*-coordinate of the particle.
GBx	Momentum $\gamma\beta$ of the particle in the **x**-direction.
GBy	Momentum $\gamma\beta$ of the particle in the **y**-direction.
GBz	Momentum $\gamma\beta$ of the particle in the **z**-direction.

References

1 H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM Philadelphia, 1992.

2 William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling, *Numerical Recipes, The Art of Scientific Computing*, Cambridge University Press, 2nd edition, 1992.

3 Francis F.Chen, *Introduction to plasma physics and controlled fusion*, Plenum Press, New York, 1984, Section 2.3.3.

4 J.H. Billen, L.M. Young, POISSON SUPERFISH, Los Alamos National Lab. Report LA-UR-96-1834.

5 M. Borland, *A Self-Describing File Protocol for Simulation Integration and Shared Postprocessors*, Proceedings of the PAC 1995 Conference.

6 S. Ramo, J.R. Whinnery, T. van Duzer, *Fields and waves in communication electronics*, John Wiley & Sons, 2nd edition, 1984, pp. 493.

7 G. Saxon, Private communication.

8 W.L. Briggs WL, H. van Emden and S. McCormick, *A Multigrid Tutorial*, SIAM Philadelphia, 2nd edition, 2000.

9 W. Hackbusch, *Multi-Grid Methods and Applications*, Springer, Berlin,1985.

10 J. Barnes, P. Hut, *A hierarchical O(N log N) force-calculation algorithm*, Nature **324**, 1986, p. 446.

11 J.E. Barnes, *A modified tree code: Don't laugh; It runs*, J. Comp. Phys **87**, 1990, p. 161.

12 P. Gibbon, G. Sutmann, *Long-Range Interactions in Many-Particle Simulation*, published in *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms*, J. Grotendorst, D. Marx, A. Muramatsu (Eds.), John von Neumann Institute for Computing, Jülich, NIC Series, Vol. **10**, ISBN 3-00-009057-6, 2002, p. 467.

13 C.A.J. van der Geer, 'FELIX design and instrumentaion', Thesis 1999, ISBN 90-386-0787-3, pp 23-25.

14 J.D. Jackson, *Classical Electrodynamics*, John Wiley & Sons, New York, 3rd edition, 1998, pp 130.

15 J. Barnes, P. Hut, *A hierarchical O(N log N) force-calculation algorithm*, Nature **324**, 1986, p. 446.

16 J.E. Barnes, *A modified tree code: Don't laugh; It runs*, J. Comp. Phys **87**, 1990, p. 161.

17 W.J. Duffin, *Electricity and magnetism*, McGraw-Hill Book Company, 4th edition, 1990, Chapter 7.10.

18 D.C. Carey, *The optics of charged particle beams*, Accelerators and storage rings, Harwood academic publishers, Chur, 1987.

19 Ximen Jiye, *Aberration Theory in Electron and Ion Optics*, Academic Press, 1986, pp. 18.

20 M. Berz, B. Erdélyi, K. Makino, *Fringe field effects in small rings of large acceptance*, PRST-AB **3**, 2000, 124001 and references therein.

21 Thanks to Bruno Muratori, Daresbury Laboratory, UK.

22 J.D. Jackson, *Classical Electrodynamics*, John Wiley & Sons, New York, 2nd edition, 1975.

23 M.J. de Loos, C.A.J. van der Geer, S.B. van der Geer, A.F.G. van der Meer, D. Oepts, R. Wünsch, Proc. of the 24th Int. Free Electron Laser Conference 2002, Argonne, USA.

24 J.L.H. Jonker, Philips Res. Rep., 12, 1957, p. 249.

Index