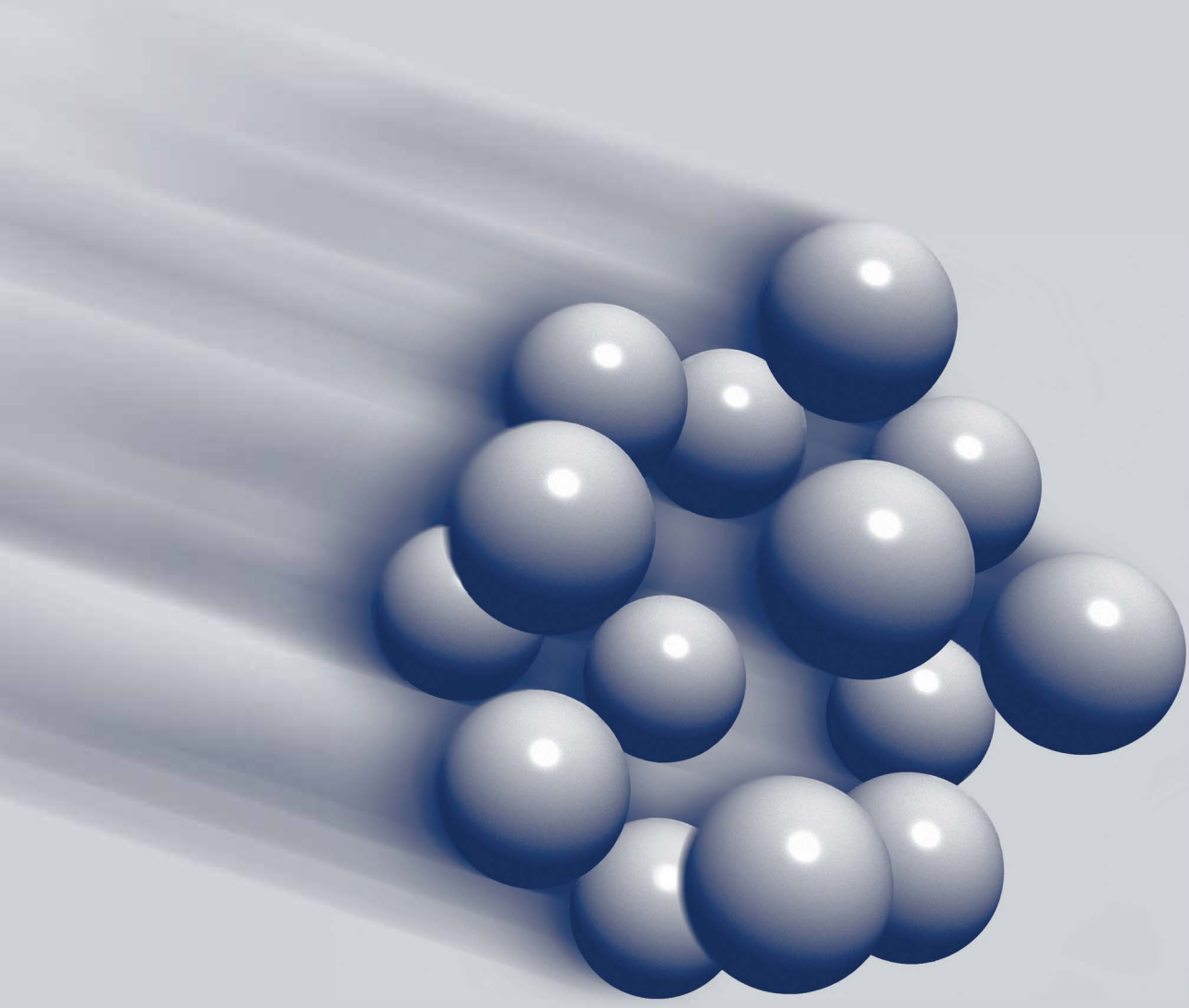
General Particle Tracer



Custom Elements
Version 3.43

About GPT

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

If there are any 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

Preface

The functionality of the General Particle Tracer GPT can be extended with custom elements. Using the GPT programming interface it is possible to write custom elements performing a variety of tasks. Examples are custom 3D electromagnetic fields, particle generators/removers and new space-charge models. It is even possible to combine custom elements with any number of additional differential equations to be solved by the GPT kernel. Furthermore, custom data analysis routines, GDFA programs, can be developed to perform non-standard calculation on the simulation results. Custom elements and programs are similar to built-in elements and must be written in the C programming language to ensure run-time efficiency.

To be able to read this manual, experience with writing computer code is useful, but typically not necessary. Although custom code needs to be written in the C language, experience with C is generally not needed. The distribution of GPT comes with the source code of almost all built-in elements which can be used as examples. For complicated elements, we refer to [1] for an introduction to the C language.

For information about the use of GPT, we refer to the GPT User Manual.

Table of contents

Introduction

One of the essential features of GPT is its adaptability and flexibility. Even without any programming experience it is easy to write a new GPT element specifying custom time dependent 3D electromagnetic fields. The required interfacing code is generated fully automatic and the efficiency of every custom element is identical to the built-in elements. All custom elements are platform independent and can freely be exchanged with other users.

Chapter one of this manual explains the philosophy of the GPT custom elements and the procedure to add them to the GPT kernel. The first two sections of this chapter provide a jumpstart to the development of simple custom elements specifying 3D electromagnetic fields for PC and Linux users.

Chapter two is a reference to the GPT kernel functions and data structures. It also provides some very advanced kernel functions that can only be used by experienced C programmers. However, none of these advanced functions are required for the development of standard custom elements.

Chapter three describes the procedure to add custom GDFA data analysis routines. Due to the limited number of functions, it also serves as a reference.

1 Custom GPT elements

Accurately modeling the set-up is essential for correct simulation results. If it is not possible to describe the set-up using the built-in elements, please invest some time in the development of custom elements. The only reason we can think of not to add a custom element is when the actual fields are not known. Unfortunately, this appears to be very common. If the fields themselves are the problem, try to find numerical or analytical expressions for the fields before continuing with GPT.

Once the fields are known, this chapter explains how to model them into GPT. PC users can start writing GPT elements very fast using the GPTwin wizard explained in section 1.1. Linux users can modify the example code generated by the GPTwin wizard as listed in section 1.2. The remaining sections in this chapter explain the details and philosophy of custom GPT elements in detail.

1.1 GPTwin wizard

The most common custom GPT elements specify 3D electromagnetic fields as function of position and time. For such elements it is not required to read all the documentation presented in this manual. It is sufficient to use the GPTwin wizard. As an example, we will describe the procedure to add an electrostatic quadrupole lens as custom element using GPTwin. The syntax will be:

`equad(ECS,L,G)`

`ECS`	Element Coordinate System
`L`	Length of the quadrupole [m]
`G`	Electric gradient [V/m^2]

where the fields are given by:

$$\mathbf{E} = \begin{cases} (\mathtt{G}y, \mathtt{G}x, 0) & \text{if } |z| < \frac{1}{2}\mathtt{L} \\ 0 & \text{otherwise} \end{cases}$$

To add a custom element using GPTwin, you must first select the "Elems" button on the Elements Toolbar (View/Elements), as shown in Figure 1-1. Then you can insert a new element from the menu (Elements/New) or right-click in the Elements Toolbar to display a pop-up menu.

Figure 1-1: GPT-Elements and GDFA-Programs toolbar.

The first page of the GPT-Element wizard asks for the name of the element, the filename and the parameters, see Figure 1-2. In our case, the element is named **`equad`**, in the filename **`equad.c`**. The parameters are **`L`**, the length of the element [m], and **`G`**, the gradient of the fields [V/m^2]. The type of element can either be local or global. Local elements can not overlap and are more efficient. It is always safer to make the element global to start with.

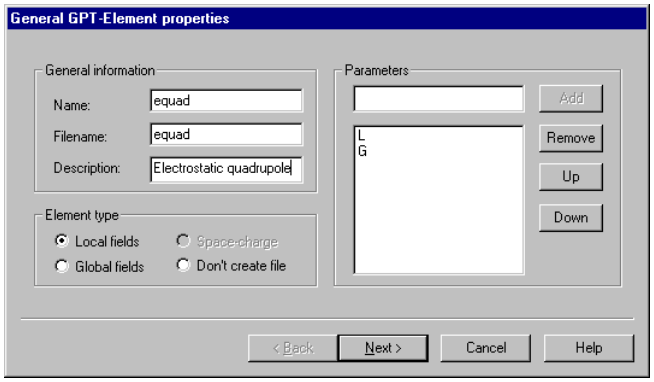


Figure 1-2: First page of the GPT-Element wizard for an electrostatic quadrupole.

The second page of the element wizard can be used to define the actual electromagnetic fields, see Figure 1-3. Expressions can be specified in C notation and the variable defining the z-range of the element can be selected. The variables that can be used in the expressions are the coordinates in the elements own coordinate system, (upper case) **`X`**, **`Y`** and **`Z`**, and the simulation time **`t`**. Furthermore, the element's parameters as defined on the first page in the GPT-Element Wizard are available.

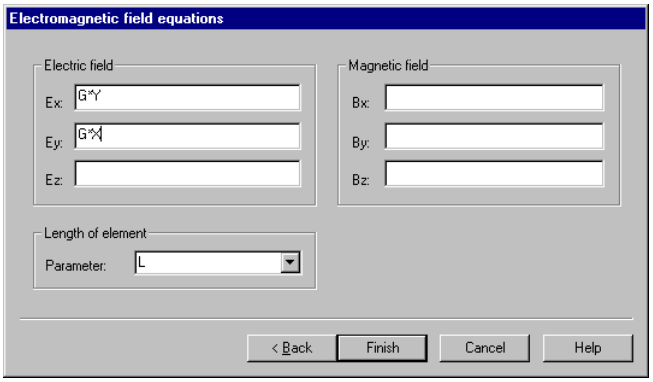


Figure 1-3: Second page of the GPT-Element wizard for an electrostatic quadrupole.

As an introduction to inexperienced C programmers, table Table 1-A lists the most commonly used operators and mathematical functions. When additional parameters need to be used in the calculations, it is most convenient to just specify **`EX`** for Ex etc. and change the generated source code after it has been created.

Table 1-A: Most commonly used C operators and mathematical functions.

Description	Examples
Addition	`X+L`
Subtraction	`X-L`
Multiplication	`G*X`
Division	`L/2`
Power	`pow(X,3)`
Square root	`sqrt(X)`
Absolute value	`fabs(Z)`
Trigonometric functions	`sin(t)`, `cos(t)`, `tan(t)`, `asin(X)`, `acos(X)`, `atan(X)`

After the custom GPT-Element is generated, it is listed in the Elements Toolbar. Double-clinking the element shows the source code of the element, see Listing 1-i on page 6. Although this is a good time to make changes to the generated source-code, custom element can always be modified.

Compiling the custom element can be done using the Menu (Elements/Compile GPT-Elems) or by right-clicking in the Elements-Toolbar. Both actions open and run a file named "**`build.bat`**". If one or more errors appear in the messages window, they should be corrected and "**`build.bat`**" must be run again using the Menu (File/Run) or the Run button on the Standard Toolbar. When no error messages appear, a new GPT executable is compiled including the **`equad`** element: From this point forward, **`equad`** can be used in a GPT inputfile just like all built-in elements.

1.2 Linux jumpstart

The code generated by the GPTwin wizard as explained in the previous section can very well be used as a starting point for the development of custom GPT elements on Linux machines. The details of the code are explained in the following sections, but when you are in a hurry, it is often sufficient to just modify the example code listed in Listing 1-i. The procedure to add a custom Linux element is outlined in section 2.1.

Listing 1-i: GPTwin generated code for an electrostatic quadrupole lens:

```
/* equad.c: Electrostatic quadrupole */

#include <stdio.h>
#include <math.h>
#include "elem.h"

/* Info structure containing all relevant parameters for this element */
struct equad_info
{
  double L ;
  double G ;
} ;

/* Forward declaration of the routine calculating the electromagnetic fields */
static int equad_sim(gptpar *par,double t,struct equad_info *info) ;

/* Initialization routine */
void equad_init(gptinit *init)
{
  struct equad_info *info ;

  /* Read Element Coordinate System (ECS) from parameter list */
  gptbuildECS( init ) ;

  /* Print usage line when the number of parameters is incorrect */
  if( gptgetargnum(init)!=2 )
    gpterror( "Syntax: %s(ECS,L,G)\n", gptgetname(init) ) ;

  /* Allocate memory for info structure */
  info = (struct equad_info *)gptmalloc( sizeof(struct equad_info) ) ;

  /* Read all parameters as doubles and store them in info structure */
  info->L        = gptgetargdouble(init,1) ;
  info->G        = gptgetargdouble(init,2) ;

  /* Register the routine calculating the electromagnetic fields to the kernel */
  gptaddEBelement( init, equad_sim, gptfree, GPTELEM_LOCAL, info ) ;
}


/* The following routine calculates the electromagnetic fields at each time step */
static int equad_sim(gptpar *par,double t,struct equad_info *info)
{
  /* Copy of parameters in info structure for convenience */
  double L, G ;

  /* Check element boundaries, return 0 when outside (local) element */
  if( fabs(Z)>info->L/2 ) return( 0 ) ;

  /* Retrieve parameters from info structure */
  L        = info->L ;
  G        = info->G ;

  /* Calculate electromagnetic fields from the above parameters
   * Particle coordinates: X,Y and Z   must be written UPPERCASE
   * Simulation time     : t           must be written lowercase
   * Electric field      : EX = ... ; EY = ...  ; EZ = ... ;
   * Magnetic field      : BX = ... ; BY = .... ; BZ = ... ;
   */
  EX = G*Y ;
  EY = G*X ;

  /* Return 1 to notify that particle is INSIDE element */
  return( 1 ) ;
}
```

1.3 The initialization routine

Every GPT element has a so-called initialization routine, or entry point. The initialization routine is called once every time the element is listed in the GPT inputfile. Simple GPT elements contain only an initialization routine. And the most simple initialization routine does nothing at all: Listing 1-ii shows the most basic custom GPT element, named **nothing**.coded in the file **nothing.c**

Listing 1-ii: Simple custom GPT element named **nothing**.

```
1.  /* nothing.c: Simplest GPT custom element possible */
2.
3.  #include <stdio.h>
4.  #include <math.h>
5.  #include "elem.h"
6.
7.  void nothing_init(gptinit *init)
8.  {
9.    /* Actual initialization code will come here */
10. }
```

The name of the initialization routine must always be the name of the element appended by **_init**. with the following declaration:

```
void name_init(gptinit *init)
```

The name is a requirement for the automatically generated interface code, as explained in the following section. The function itself has only one parameter: A pointer to a **gptinit** structure named **init**. Although **init** contains all information about the element and its parameters, it should never be accessed directly. This is to ensure compatibility with future GPT versions. GPT kernel functions are provided to access the members of the **gptinit** structure.

For people novice to C, we will very briefly explain some details of the code:

/* */ All lines within these brackets are treated as comment lines by the compiler. The first line typically descibes the element briefly.

#include Provides the declarations of a number of standard functions from the C library. Similarly, the "**elem.h**" file provides the declarations for the custom routines of the GPT kernel. In this example all **#include** lines can essentially be omitted because we don't use any routines at all. However, in realistic examples these lines are always needed.

The next example reads a double-precision floating point value and an integer parameter from the inputfile and prints an error message when the number of arguments is not correct, see Listing 1-iii.

Listing 1-iii: Read a floating-point and integer parameter from the GPT inputfile.

```
1.  /* getdblin.c: Read a double and an integer as parameters */
2.
3.  #include <stdio.h>
4.  #include <math.h>
5.  #include "elem.h"
6.
7.  void getdblint_init(gptinit *init)
8.  {
9.    double arg1 ;
10.   int arg2 ;
11.
12.   if( gptgetargnum(init)!=2 )
13.     gpterror( "Syntax: %s(double,int) ;\n", gptgetname(init) ) ;
14.
15.   arg1 = gptgetargdouble(init,1) ;
16.   arg2 = gptgetargint(init,2) ;
17.
18.   printf( "arg1=%f, arg2=%d\n", arg1, arg2 ) ;
19. }
```

As is clear from line 7, the name of the element is **getdblint**. However, to maintain MS-DOS compatibility, we still use filenames of only 8 characters resulting in the filename **getdblin.c**.

The function **gptgetargnum** returns the number of arguments. When the number of arguments is not equal, specified as **!=** in the C language, to 2 an error is printed using the **gpterror** function. The syntax of **gpterror** is just like the C function **printf**: A string, followed by a number of parameters whose values are inserted in the string at the position of the **%**. Every **%s** in the string must have a corresponding string parameter, every **%f** a floating point-number and every **%d** must be followed by an integer. The string returned

by **gptgetname** is the name of the element. When the number of arguments is not 2, the error printed will be:

```
Filename(linenumber): syntax: getdblint(double,int) ;
```

The following lines read the first and second parameters of this element as floating point and integer values respectively using the **gptgetarg** functions. When the arguments are not of the correct type, an error is printed and GPT is terminated. It is also possible to test the argument type beforehand using **gptgetargtype**, but this is generally not needed. To be able to test the element the obtained parameters are printed with the **printf** function, but in a normal element this line would not be there.

Clearly the **init** parameter plays a vital role because it is passed along to the **gptgetargnum**, **gptgetname** and **gptgetarg** functions. So what is this **init**? It contains all parameters of the element in an internal representation. The reason for this secrecy is that this allows us to provide future compatibility when the declaration needs to be changed.

1.4 Custom Element Interface

When the code for a new element has been written, it must be compiled and added to the kernel. Normally, this can be done completely from within GPTwin. This section describes what happens behind the scenes. Information that is useful for understanding GPT and sometimes relevant for debugging purposes. The procedure to follow when a custom element needs to be added or removed on a Linux machine is presented on page 19.

Schematically the GPT executable has a structure as shown in Figure 1-4. Any number of custom elements can be added and the interface between built-in elements is precisely the same as the interface with the custom elements.

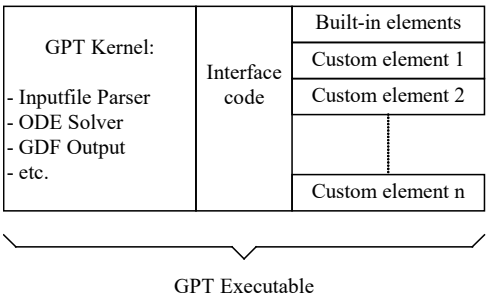


Figure 1-4: Schematic picture of the GPT executable.

The list of custom elements is located in the **`GPT/kerel`** directory in a file named **`elemlist`**. The files shown in the custom section in the GTPwin Elements toolbar are read from this file. Linux users can manually edit this file using any ascii editor like vi or emacs. When **`elemlist`** is modified, new interface code needs to be generated. GPTwin writes this code automatically when an element is added in the file gps.c. Linux users must specify make in the **`GPT/kernel`** directory.

Once the interface code is generated, the new element must be compiled and linked to the GPT kernel. Custom elements are always located in the **`GPT/elems`** directory. Linux users just specify **`make`** in this directory. GPTwin users can select Elements/Compile from the menu. This opens a file named **`build.bat`** with the same **`make`** command. The make command compiles all changed custom elements and rebuilds the GPT executable containing all elements specified in **`elemlist`**. When an error is present, Linux users can specify **`make`** again, GPTwin users can run **`build.bat`** again.

When there are no errors, the new GPT executable is copied to the **`GPT/bin`** directory. Linux users should include this directory in their PATH environment variable. It is generally not a good idea to change an executable that is running. Therefore do not change a custom element while GPT is running, since this requires a rebuild of the executable itself. If this is required anyway, make a copy of the GPT executable first, run this copy, and then change the custom elements.

1.5 Callback functions

During simulations the GPT kernel relies completely on callback functions. To explain the callback mechanism used by the kernel, we will start by explaining a simple and intuitive example of a callback function: The standard C implementation of the sorting algorithm "quicksort".

Quicksort is a sorting algorithm capable of sorting any kind of data in any order. For example arrays of integers, floating point numbers or even strings can all be sorted using this algorithm. Every data-type can be sorted in various ways: Ascending, descending, case insensitive etc.

Because of the diversity, one could think that many implementations for quicksort are needed: One for every kind of data and sort order. An other option would be to have only one very complex quicksort implementation capable of handling all data types and sort-orders. The Standard C library however provides one simple and efficient function, **`qsort`**, that is capable of sorting all kinds of data in every order.

Suppose for example we need to sort a buffer of *N* integers:

```
int buffer[N] ;
```

First we need to write a function capable of comparing two integers in the desired sort order. For example:

```
1. int compare_int(int *a, int *b)
2. {
3.   if(*a > *b) return( +1 ) ;
4.   if(*a < *b) return( -1 ) ;
5.   return( 0 ) ;
6. }
```

Now that we have written this function, the **`qsort`** function can be used in the following way:

```
qsort(buffer,sizeof(int),N,compare_int)
```

This will sort the buffer of *N* integers using our own function comparing two individual items. When we want to sort in reverse order for example, we don't have to modify the sorting code, we only need to write a different comparison routine. The same applies if we want to sort a different data-type. Here is for example a routine sorting double precision floating point numbers in reverse order, ignoring the sign:

```
1. int compare_double_reverse(double *a, double *b)
2. {
3.   if(fabs(*a) > fabs(*b)) return( -1 ) ;
4.   if(fabs(*a) < fabs(*b)) return( +1 ) ;
5.   return( 0 ) ;
6. }
```

It is clear that this mechanism provides optimal "teamwork". The quicksort implementation does not need information about the data type and sort order and the user does not need to worry about the implementation of the quicksort algorithm. In this example the comparison function is called a callback function. The quicksort routine is instructed to call this function.

An advantage of the callback mechanism is the separation between the sorting algorithm and the sort order on source-file level. The sort-order can be specified in a different source file because it doesn't require any specific information about the sort-algorithm and vice-versa. In GPT this means that every custom element can be developed as one separate source-file that can be debugged and exchanged with other users as a unit. There is no need to keep track of changes made at different locations in the GPT kernel, because all information about an element is contained in a single file. It is our experience that this greatly reduces errors and allows people in teams to work much more efficient.

Complex custom elements make use of a number of callback functions. For example, specifying electromagnetic fields while solving an additional differential equation requires at least three callback routines. Without the callback mechanism, the code for such an element would be present in at least three different locations in the overall structure. During the development of such an element this is generally not a problem. However, understanding such an element written by a colleague or trying to figure out what part of the code corresponds to which element can sometimes be very hard. Not to speak of keeping track of revisions of individual elements! The callback mechanism allows developers to keep the code for one element in one source file, even if it acts on different locations in the overall structure. These source files acts as logical units, can be documented separately, allow revision control and can be exchanged with colleagues.

1.6 The info structure

Every GPT element reads its parameters in the initialization routine. Typical GPT elements have parameters that can be divided in two categories:

- Position and orientation
- Element parameters such as length, strength, radius, phase etc.

The position and orientation parameters are handled internally by the GPT kernel. This guarantees that all elements can be positioned anywhere in 3D space with any orientation in a consistent way. The callback function specifying electromagnetic fields is presented with particle coordinates in the local coordinate system. The required back-and forth transformations are transparently performed by the GPT kernel. Properly aligned elements are internally handled more efficiently, but developers of custom elements do not need to worry about the details. An exception are custom space-charge models that, due to the required efficiency, act directly on the internal GPT data structures in the Word Coordinate System (WCS).

Because every element has its own parameters, there is no standard structure or array where these parameters are stored. Therefore, every element must define its own **`info`** structure. For a solenoid with a radius and a current, this info structure looks like:

```
1. struct solenoid_info
2. {
3.   double radius ;
4.   double current ;
5. }
```

Naturally, the variables could equally well be named **`R`** and **`I`**, but this is a matter of taste. The main advantage of such an info-structure for every element is readability: All parameters can have a well-chosen name.

The info structure is the communication medium between the initialization routine and all the callback routines. For more complicated elements, it also provides communication between different callback functions. When the same element is present at a different location, it can have different element parameters. For example when two solenoids are present, they can have different positions, and different radii. Therefore, the initialization routine allocates an **`info`** structure in memory every time an element is placed in the set-up. The GPT kernel makes sure the callback function for the magnetic fields is called with the correct info structure as parameter. This method still works very efficiently when hundreds of elements are present in a complex 3D set up.

1.7 Comparison with an object oriented model

Many experienced programmers have pointed out to us that GPT elements are in fact object oriented, while not written in a typical object oriented language. This is because most of the characteristics are identical:

Object oriented language like C++	GPT Element
Class member functions share all class variables.	GPT callback functions share the variables defined in the info structure.
Reduction of name space pollution because every object/class has its own name-space. Identical function names can exists in different classes.	Reduction of name space pollution because every GPT element is written in a separate source file The names of all functions, except the `_init` entry point, are only known within the file module and can therefore be identical.
Well-designed classes can be used in different programs.	Well-designed GPT elements can be used by different colleagues, even with differen GPT versions.
Well-designed classes are often useful for different applications.	Well-designed GPT elements are often used by colleagues working on different projects.

The main difference between the object oriented and the GPT model is object hierarchy. Using GPT it is not possible to define a GPT element based on an existing one. It is required to copy the original before making the desired modifications.

1.8 Reading parameters into the info structure

Because the same element can be positioned more than once at different locations with different parameter in the same set-up, every element must have its own info-structure. To accomplish this, a number of steps have to be taken. First, following the solenoid example of the previous section, a variable named **info** must be declared:

```
struct solenoid_info *info ;
```

Technically, the variable **info** is a pointer to a structure of type **solenoid_info**. In other words, it is not the info structure itself, it only points to an info structure. The actual info structure is maintained by the GPT kernel and can be accessed after the following line:

```
info = (struct solenoid_info *)gptmalloc( sizeof(struct solenoid_info) ) ;
```

The **gptmalloc** function allocates memory for the info structure and returns the correct value (the address) for the **info** variable. The GPT kernel makes sure that every element has its own parameters. After the info variable is properly initialized, it member variables can be accessed as demonstrated below:

```
info->radius  = gptgetargdouble(init,1) ;
info->current = gptgetargdouble(init,2) ;
```

The **info->** notation can both appear at the left- and right-hand side of expressions. In other words, expressions like **power = info->current*info->current ;** are possible.

1.9 Calculating electromagnetic fields

An element calculating electromagnetic fields makes use of a callback function for the actual field calculation. For the electrostatic quadrupole of section 1.1, this callback function can be written as:

```
1.  static int equad_sim(gptpar *par,double t,struct equad_info *info)
2.  {
3.    /* Check element boundaries, return 0 when OUTSIDE (local) element */
4.    if( fabs(Z)>info->L/2 ) return( 0 ) ;
5.
6.    /* Calculate electromagnetic fields from the above parameters */
7.    EX = info->G*Y ;
8.    EY = info->G*X ;
9.
10.   /* Return 1 to notify particle is INSIDE element */
11.   return( 1 ) ;
12. }
```

The name of the callback function is the name of the element, appended by **_sim** by convention. Any other valid function name is also possible. The parameters of the function are a particle with its coordinates, the simulation time **t** and the (address of the) info-structure of the element. For convenience, the particle coordinates can be specified as uppercase **X**, **Y** and **Z** and are always presented in the element's local coordinate system. The GPT kernel takes care of all the required coordinate transforms.

The parameters of this element, **G** and **L**, can be accessed by prepending them with **info->**. The readability of the file is improved, especially when an element has many arguments, when we start the function with:

```
G = info->G ;
L = info->L ;
```

Now you can use **G** and **L** instead of **info->G** and **info->zlen**. This however is a matter of taste.

The first task of every **_sim** routine is to test if the particle is inside the element. And, if not, to **return(0)**. In this case, the test is **if(fabs(Z)<info->L/2)**, even if **equad** is not properly aligned with the z-axis, because the *z* coordinate in the element's own coordinate system is aligned by definition. The **fabs** function returns the absolute value of its argument.

Once we know for sure that the particle is inside the element, the **_sim** routine must calculate the fields, **EX**, **EY**, **EZ**, **BX**, **BY** and **BZ** at the particle position. In this case, only **EX** and **EY** are specified. These fields are calculated in the element's coordinate system; The GPT kernel transforms them back to the Word Coordinate System (WCS).and adds them to the fields of all other elements and the space-charge.

To indicate that the particle is inside the element, the **_sim** routine ends with a **return(1)** statement.

The main task of the **_init** routine is checking and storing the parameters in the info structure. As will be explained below, the Element Coordinate System (ECS) matrix must be stored and the **_sim** routine must be registered as callback function. The complete **equad** initialization routine is given below:

```
1.  void equad_init(gptinit *init)
2.  {
3.    struct equad_info *info ;
4.
5.    /* Read Element Coordinate System (ECS) from parameter list */
6.    gptbuildECS( init ) ;
7.
8.    /* Print usage line when the number of parameters is incorrect */
9.    if( gptgetargnum(init)!=2 )
10.     gpterror( "Syntax: %s(ECS,L,G)\n", gptgetname(init) ) ;
11.
12.   /* Allocate memory for info structure */
13.   info = (struct equad_info *)gptmalloc( sizeof(struct equad_info) ) ;
14.
15.   /* Read all parameters as doubles and store them in info structure */
16.   info->L        = gptgetargdouble(init,1) ;
17.   info->G        = gptgetargdouble(init,2) ;
18.
19.   /* Register the routine calculating the electromagnetic fields to the kernel */
20.   gptaddEBelement( init, equad_sim, gptfree, GPTELEM_LOCAL, info ) ;
21. }
```

After the declaration of the **_init** routine, the matrix for the element coordinate system is constructed from the ECS specification on the command-line using the **gptbuildECS** function. This function also removes the ECS specification from the parameter-list. Thus when an **equad** is positioned using:

```
equad("wcs","z",1, L,G)
```

The number of parameters is 5 before, but only 2 after the **gptbuildECS** function. This allows future ECS specifications to be possible without having to rewrite all custom elements. The ECS matrix is stored by the GPT kernel and used for all transformations back and forth between the WCS and the **_sim** routine's ECS. Special versions of transformation code are used for **"I"** and **"z"** transformation to save CPU time, but this is completely transparent to the user.

To make sure the **_sim** routine is actually called, it must be registered by the GPT kernel using the **gptaddEBelement** function. For a global element, the fourth parameter must be **GPTELEM_GLOBAL**.

1.10 Removing a particle

When a particle hits a wall or iris, it needs to be removed from the simulation. The procedure to remove a particle is almost identical to specifying electromagnetic fields. As a result both actions, specifying fields and/or removing a particle, can be combined in the same **_sim** routine. As a simple example. the **_sim** routine that only removes a particle when z<**zmin** is presented below:

```
1. static int zmin_sim(gptpar *par,double t,struct equad_info *info)
2. {
3.   /* Remove particle when Z<zmin */
4.   if( Z<info->zmin ) gptremoveparticle(par) ;
5.
6.   /* Return 1 to notify particle is INSIDE element */
7.   return( 1 ) ;
8. }
```

The function **gptremoveparticle** doesn't actually remove the particle from the simulation. This is because if a particle is removed in a timesteps that fails the accuracy criterion, the timesteps should be retried with the removed particle present again. Particles are only "marked for removal" and are only removed permanently after a successful timestep is completed.

It is important to realize that when such a **zmin** element is included in a simulation, the GPT kernel may still request other elements to calculate electromagnetic fields for particles with a z-coordinate less than **zmin**. This is because the order of the called elements is generally not know. Therefore, if a particle with z<**zmin** is in a solenoid, this solenoids **_sim** function may be called before the **zmin**.

1.11 Additional tips for writing custom GPT elements

- Don’t write elements that are too complex. It is usually better to separate such an element in two or three simpler elements that, when combined in the inputfile, do the same. For example it is not a good idea to write an element starting particles and calculating electromagnetic fields. Separated elements have a much higher chance to be reused, by yourself or others, and are less error prone. The loss in CPU time generated by the overhead of having two elements instead of one is almost always negligible.

- The **`info`** structure should contain parameters such that the function calculating the electromagnetic fields runs as fast as possible. This is because the initialization function is executed only once, where the electromagnetic fields are typically calculated millions of times. For example, consider storing the square of some parameter if this eliminates the use of a costly **`sqrt`** operation in the **`_sim`** routine. Or maybe pre-multiply a constant with $1/4\pi\varepsilon_0$ before storing it in the info structure. On the other hand, please don’t over-optimize. Especially premature optimization can be very inefficient. Only when the element’s equations are thoroughly tested, optimization can start.

- Don’t add a new element while GPT is running. If you intend to add a custom element while you are executing a long GPT run, it is best to make a copy of the executable and run the copy. This way the standard GPT executable in the binaries directory can be modified without interfering with the running copy.

2 Element code reference

This chapter lists all functions and data structures that can be used in custom GPT elements.

2.1 Unix procedure for adding a GPT custom element

To add a custom element to the Linux version of GPT, the following steps have to be taken:

- Type **`cd ~/gpt/elems`** (where **`~/gpt`** is the GPT installation directory).
- Write the actual code. A good starting point is a copy of one of the built-in elements provided in the **`sources`** directory.
- Add your element to the file **`elemlist`**. This is an ascii file with two words on each line. The first word specifies the name of your element; this is the name of the initialization (**`_init`**) function without the actual **`_init`**. This is also the name how the element appears in the GPT inputfile. The second word in **`elemlist`** provides the filename without extension containing the C code for the element. For an element named **`myelemente`** with the source code in file **`myelem.c`** you should add the line "**`myelement myelem`**" to **`elemlist`**.
- Type **`make`** twice. This compiles your new element and it creates the code needed by the GPT kernel to efficiently interface with the new element.
- When an existing element is modified, **`make`** needs to be run only once to create a new GPT executable.

GPT is now ready to run with your new element. Whenever you modify your custom element, only the last **`make`** needs to be repeated.

To remove a custom element, delete it from **`elemlist`** and then run **`make`** twice.

2.2 Obtaining element parameters

All parameter handling must be performed in the initialization routine. The **`init`** parameter of the initialization routine contains all element parameter information and must be passed unmodified to all functions described in this section. When parameter must be passed to other functions, please read section 1.7 describing how to store parameters in the **`info`** structure.

To code to check if the number of arguments is correct is typically:

```
if( gptgetargnum(init)!=??? )
    gpterror( "Syntax: %s(ECS,???)\n", gptgetname(init) ) ;
```

The **`gpgetargdouble`**.function is most often used to read the parameters.

2.2.1 gptgetargdouble

`double gptgetargdouble(gptinit *init, int argnum) ;`
Returns the specified argument as a floating point number. If the argument is not a floating point number, an error is printed and GPT is terminated.

`init`	First argument of the init function.
`argnum`	Argument number, starting from 1.

Example: Read floating point arguments

```
1. void foo_init(gptinit *init)
2. {
3.   double arg1, arg2, ... ;
4.
5.   arg1 = gptgetargdouble(init,1) ;
6.   arg2 = gptgetargdouble(init,2) ;
7.   ...
8. }
```

2.2.2 gptgetargint

`int gptgetargint(gptinit *init, int argnum) ;`
Returns the specified argument as an integer.

If the argument is not an integer, an error is printed and GPT is terminated. Whether or not an argument is an integer, does not depend on intermediate results in an expression. For example **`(1/3)*3`** is an integer according to GPT.

`init`	First argument of the init function.
`argnum`	Argument number, starting from 1.

2.2.3 gptgetargnum

`int gptgetargnum(gptinit *init) ;`

Returns the number of parameters listed in the inputfile. Typical use is to check the number of arguments and if this is not correct print a usage line.

`init` First argument of the init function.

Example: Print the total number of parameters

```
1.  void foo_init(gptinit *init)
2.  {
3.    printf( "%d", gptgetargnum(init) ) ;
4.  }
```

2.2.4 gptgetargstring

`const char *gptgetargstring(gptinit *init, int argnum) ;`

Returns the specified argument as a string. If the argument is not a string, an error is printed and GPT is terminated.

`init` First argument of the init function
`argnum` Argument number starting from 1

Example: Open the file specified by the first parameter

```
1.  void foo_init(gptinit *init)
2.  {
3.    FILE *fp ;
4.    const char *filename ;
5.
6.    filename = gptgetargstring(init,1) ;
7.    if( (fp=fopen(filename,"r"))==NULL )
8.      gptinperror( "%s: %s\n", filename, strerror(errno) ) ;
9.    ...
10.   fclose(fp) ;
11. }
```

Note: The C library function **`strerror()`** in this case returns an ascii string containing information about why the file could not be opened.

2.2.5 gptgetargtype

`int gptgetargtype(gptinit *init, int argnum) ;`
Returns the type of the specified argument.

`init`	First argument of the init function
`argnum`	Argument number starting from 1
Return value	One of the constants listed in Table 2-A.

Table 2-A: Return values of **`gptgetargtype`**

Return value constant	Read using
`GPTTYPE_INT`	**`gptgetargint`**, **`gptgetargdouble`**
`GPTTYPE_DOUBLE`	**`gptgetargdouble`**
`GPTTYPE_STRING`	**`gptgetargstring`**

The code presented in Listing 2-i prints all parameters and their types. Please note that an integer parameter can also be read as a floating-point number, but not vice-versa.

Listing 2-i: Print element parameters and types.

```
1.  /* Tutorial: printpar.c - Print parameters */
2.
3.  #include <stdio.h>
4.  #include "elem.h"
5.
6.  void printparameter_init( gptinit *init )
7.  {
8.    int i ;
9.
10.   fprintf( stderr, "Elementname: %s\n", gptgetname(init) ) ;
11.   fprintf( stderr, "Number of parameters: %d\n",
12.     gptgetargnum(init) ) ;
13.
14.   for(i=1 ; i<=gptgetargnum(init) ; i++)
15.   {
16.     switch( gptgetargtype(init,i) )
17.     {
18.       case GPTTYPE_INT:
19.         fprintf( stderr, "Arg %d (integer): %d\n",
20.           i, gptgetargint(init,i) ) ;
21.         break ;
22.
23.       case GPTTYPE_DOUBLE:
24.         fprintf( stderr, "Arg %d (double): %f\n",
25.           i, gptgetargdouble(init,i) ) ;
26.         break ;
27.
28.       case GPTTYPE_STRING:
29.         fprintf( stderr, "Arg %d (string): %s\n",
30.           i, gptgetargstring(init,i) ) ;
31.         break ;
32.     }
33.   }
34. }
```

2.2.6 gptgetname

`const char *gptgetname(gptinit *init) ;`
Returns the name of the element.

`init`	First argument of the init function.

Example: Print the name of an element: **`foo`** in this case.

```
1.  void foo_init(gptinit *init)
2.  {
3.    printf( "Elementname: %s\n", gptgetname(init) ) ;
4.  }
```

2.3 Specifying Electromagnetic fields

A function specifying electromagnetic fields must start with a **gptbuildECS** function in the initialization routine. Then the number of parameters can be checked and an info structure created. The function calculating the actual electromagnetic fields, typically named elementname_sim, can be registered using the **gptaddEBelement** function.

In the **_sim** function itself, the macro's **X**, **Y** and **Z** can be used to obtain the particle coordinates. The simulation time is the variable **t**. The electromagnetic fields can be stored using the macro's **BX**, **BY**, **BZ**, **EX**, **EY** and **EZ**. Both the particle coordinates and the electromagnetic fields are in the element's coordinate-system. The GPT kernel provides all required transformations.

2.3.1 BX, BY, BZ

BX
BY
BZ

Macros to specify the magnetic field at the position of the particle. They can be used everywhere in the simulation-routine.

Example: Specify magnetic field

```
1. int foo_sim(gptpar *par, double t, struct foo_info *info)
2. {
3.   BX = ... ;
4.   BY = ... ;
5.   BZ = ... ;
6. }
```

2.3.2 EX, EY, EZ

EX
EY
EZ

Macros to specify the electric field at the position of the particle. They can be used everywhere in the simulation-routine.

Example: Specify electric field

```
1. int foo_sim(gptpar *par, double t, struct foo_info *info)
2. {
3.   EX = ... ;
4.   EY = ... ;
5.   EZ = ... ;
6. }
```

2.3.3 gptaddEBelement

```
void gptaddEBelement(gptinit *init, simfn simfn,
  exitfn exitfn, int type, void *info ) ;
typedef int (*simfn)(gptpar *par,double t,void *info) ;
typedef void (*exitfn)(void *info) ;
```

Registers an element providing electromagnetic fields as function of position and time.

init	First argument of the **_init** function.
simfn	Address of the function returning the fields, the simulation function.
exitfn	Address of the function called once at the end of the simulation. Typically **gptfree**.
type	Type specifying the kind of element. It must be one of the following: **GPTELEM_LOCAL** For a local element **GPTELEM_GLOBAL** For a global element
info	Pointer to any additional information. It is passed to both **simfn** and **exitfn**.

simfn must be declared as:

```
int simfn(gptpar *par,double t,void *info) ;
```

par	Structure containing all particle information relative the the ECS. Use **X**,**Y**,**Z**, **EX**, **EY**, **EZ**, **BX**, **BY** and **BZ** macro's to access the coordinates. Never access **par** directly.
t	Simulation time [s].
info	The same info pointer as specified in gptaddEBelement.
Return value:	1 if particle is in local element, 0 if not.

exitfn must be declared as"

```
void exitfn(void *info) ;
```

info	The same info pointer as specified in gptaddEBelement.

Example: Code fragment.

```
gptaddEBelement( init, test_sim, gptfree, GPTELEM_LOCAL, info ) ;

static int test_sim(gptpar *par,double t,struct test_info *info)
{
  /* Check element boundaries, return 0 when OUTSIDE (local) element */
  if( fabs(Z)>info->L/2 ) return( 0 ) ;

  /* Calculate electromagnetic fields from the above parameters */
  EX = info->... ;
  EY = info->... ;

  /* Return 1 to notify particle is INSIDE element */
  return( 1 ) ;
}
```

2.3.4 gptbuildECS(gptinit *init);

`void gptbuildECS(gptinit *init) ;`
Builds an ECS as specified by the first parameters of the element as supplied in the inputfile.

`init` First argument of the init function.

`gptbuildECS` must be the first GPT kernel function called from within the init function. The ECS specification is removed from the list of element parameters. Therefore parameter checking must take place after this function has been called.

Example: A sample EB element

```
1. void foo_init(gptinit *init)
2. {
3.   gptbuildECS(init) ;
4.
5.   if( gptgetargnum(init)!=2 )
6.     gptinperror("Syntax: %s(arg1,2)",gptgetname(init) ) ;
7.
8.   ...
9. }
```

2.3.5 gpterror

`void gpterror(const char *fmt,...) ;`
Prints an error message and terminates GPT. The syntax is equivalent to the **`printf()`** syntax.

Always use this function and never use **`exit()`** to terminate GPT.

2.3.6 gptremoveparticle

`void gptremoveparticle(gptpar *par) ;`
Instructs the GPT kernel to kill a particle.

The particle is effectively removed only when the timestep is completed successfully. Otherwise the step will be redone using a smaller timestep with the particle still present.

`par` Particle to be killed

Example: Kill all particles with a negative *z*-coordinate.

```
1. int foo_sim(gptpar *par, double t, struct foo_info *info)
2. {
3.   if( Z<0 ) gptremoveparticle(par) ;
4.   return( 0 ) ;
5. }
```

2.3.7 gptwarning

`void gptwarning(const char *fmt,...) ;`
Prints a warning message and continues normal execution. The syntax is equivalent to the **`printf()`** syntax.

We have a tendency never to give warnings during initialization. An inputfile is either correct or not. Therefore we give errors (**`gpterror`**) or nothing. If you have a different opinion or encounter a typical warning situation, please don't hesitate to use this function.

2.3.8 X, Y, Z

X
Y
Z

Macros to obtain the *x*, *y* and *z* coordinates of a particle. They can be used in the calculation of the fields in a simulation-routine.

These coordinates are relative to the ECS. See section 1.9 for details.

Example: Field structure of a quadrupole lens, provided **info->G** contains the field gradient.

```
1.  int foo_sim(gptpar *par, double t, struct foo_info *info)
2.  {
3.    BX = info->G * Y ;
4.    BY = info->G * X ;
5.
6.    return( 1 ) ;
7.  }
```

2.4 Predefined constants

To simplify writing custom elememts, the constants listed in Table 2-B are predefined in the GPT kernel. Therefore you can use all these constants in elements without having to set their value first.

Table 2-B: Predefined constants.

Name	Value	Description
gpt_c	299792458	Speed of light
gpt_deg	$180/\pi$	Conversion factor from radians to degrees
gpt_e	2.71828182845904523536	e
gpt_eps0	$1/\mu_0 c^2$	Permittivity of vacuum: ε_0
gpt_ma	$1.66057 \cdot 10^{-27}$	Unified mass constant
gpt_me	$9.10953 \cdot 10^{-31}$	Mass of an electron
gpt_mp	$1.67265 \cdot 10^{-27}$	Mass of a proton
gpt_mu0	$4\pi \cdot 10^{-7}$	Permeability of vacuum: μ_0
gpt_pi	3.14159265358979323846	π
gpt_qe	$-1.6021892 \cdot 10^{-19}$	Charge of an electron

Note: Some C complilers also have predefined constants, such as **PI**, but we strongly advise using **gpt_pi** instead. This way the portability of your custom element is ensured.

2.5 Low level memory management

The standard C memory management functions: **malloc**, **free** and **realloc** have their respective GPT counterparts **gptmalloc**, **gptfree** and **gptrealloc**. When an out of memory situation occurs, the GPT functions automatically generate an error message and terminate GPT. Furthermore, they are slightly more robust.

In most cases the C++ container classes such as a **vector** are a better choice. The main reason the following functions are kept is for reasons of backward compatibility. Please note that info structures containing C++ classes must be allocated using **new** and not using **gptmalloc**.

2.5.1 gptfree

void gptfree(void *mem)
Frees the memory allocated by **gptmalloc** or **gptrealloc**.

mem Pointer to memory obtained using **gptmalloc** or **gptrealloc**.

2.5.2 gptmalloc

void *gptmalloc(size_t size)
Dynamically allocates the amount of bytes specified by **size**.

size Amount of memory to be allocated in bytes.
Return value: A pointer to the allocated memory. When not enough memory is available, an error message is generated and GPT is terminated.

To return the allocated memory to the system, the **gptfree** function must be used. Using **gptrealloc**, the allocation size can be modified.

Example: Allocate and free memory

```
1.  #define COUNT 100
2.  double *data ;
3.  int i ;
4.
5.  data = (double *)gptmalloc(COUNT*sizeof(double)) ;
6.  for(i=0 ; i<COUNT ; i++) data[i] = 0.0 ;
7.
8.  /* Do something with the data here */
9.
10. gptfree(data) ;
```

2.5.3 gptrealloc

void *gptrealloc(void *mem, size_t size)
Modifies the amount of memory allocated by **gptmalloc** or a previous **gptrealloc**.

mem Pointer to memory obtained using **gptmalloc** or a previous **gptrealloc**.
size Amount of memory to be allocated in bytes. This parameter can be both larger and smaller than the previous number of allocated size.
Return value: A pointer to the newly allocated memory. When not enough memory is available, an error message is generated and GPT is terminated. Please note that the memory location can also change when the amount of memory is decreased.

To return the allocated memory to the system, the **gptfree** function must be used

2.6 Item arrays

An item array is an array that can easily grow or shrink. The data within the array can be of any type, but must always be unordered: Elements within an item array cannot be dependent of their position within the array. Furthermore, the memory location can change when the array grows and even when it shrinks.

Unlike using **gptrealloc**, item arrays are an efficient mechanism when items are often added or removed. This is because memory is always allocated in blocks specified by the **chunkcount** parameter, providing the user with a parameter defining the trade-off between memory usage and speed.

Item arrays are allocated and deallocated using the **gptmallocitemarray** and **gptfreeitemarray** functions. Individual items/objects in the arrays are allocated and deallocated using the **gptmallocitem** and **gptfreeitem** functions.

Item arrays or generally not used by GPT users. They merely provide the basic operations of the synchronization arrays covered in the next section.

2.6.1 gptfreeitem

void *gptfreeitem(void *mem, size_t blocksize, int chunkcount, int len, int n)
Frees an item in an item array. The array is shrunk automatically and can even change position in memory.

mem	Pointer to memory obtained using **gptmallocitemarray** or a previous call to **gptmallocitem** or **gptfreeitem**.
blocksize	Size of an item in the array.
chunkcount	Minimum number of items that memory is reserved for.
len	Current length of the array.
n	Index of the item to be removed.
Return value:	A pointer to the newly allocated memory: An array of **len-1** items, occupying **blocksize** byte each.

This operation moves the last element in the array to the position of the removed item to keep the array sequential. The memory of the last item is only freed when **3*chunksize/2** items are to be freed.

2.6.2 gptfreeitemarray

void gptfreeitemarray(void *mem, size_t blocksize, int chunkcount, int len)
Frees the memory allocated by **gptmallocitemarray**, **gptmallocitem** or **gptfreeitem**.

mem	Pointer to memory obtained using **gptmallocitemarray** or a previous call to **gptmallocitem** or **gptfreeitem**.
blocksize	Size of an item in the array.
chunkcount	Minimum number of items that memory is reserved for.
len	Current length of the array.

2.6.3 gptmallocitemarray

`void *gptmallocitemarray(size_t blocksize, int chunkcount, int len)`
Allocates memory for an item array.

`blocksize`	Size of an item in the array.
`chunkcount`	Minimum number of items that memory is reserved for.
`len`	Number of items requested.
Return value:	A pointer to the newly allocated memory: An array of **`len`** items, occupying **`blocksize`** byte each.

Example: Various functions

```
1.  #define CHUNKCOUNT 256
2.
3.  double *data ;
4.  int len, i ;
5.
6.  len = 100 ;
7.
8.  data = gptmallocitemarray(sizeof(double),CHUNKCOUNT,len) ;
9.  for(i=0 ; i<COUNT ; i++) data[i] = 0.0 ;
10.
11. /* Do something with the data here */
12.
13. /* Add an item */
14. data = gptmallocitem(data,sizeof(double),CHUNKCOUNT,len) ;
15. data[len] = 1000.0 ;
16. len++ ;
17.
18. /* Remove the 10'th element */
19. data = gptfreeitem(data,sizeof(double),CHUNKCOUNT,len,10) ;
20. len-- ;
21.
22. gptfreeitemarray(data,sizeof(double),CHUNKCOUNT,len) ;
```

2.6.4 gptmallocitem

`void *gptmallocitem(void *mem, size_t blocksize, int chunkcount, int len)`
Allocates memory for an additional item in an item array. New items will always be appended at the end.

`mem`	Pointer to memory obtained using **`gptmallocitemarray`** or a previous call to **`gptmallocitem`** or **`gptfreeitem`**.
`blocksize`	Size of an item in the array.
`chunkcount`	Minimum number of items that memory is reserved for.
`len`	Current length of the array.
Return value:	A pointer to the newly allocated memory: An array of **`len+1`** items, occupying **`blocksize`** byte each.

2.7 Particle sets

Before the simulation starts, particle coordinates are stored in so-called sets. For example, it is possible to have a set for heavy ions and a set for electrons. Both types of particles will be traced simultaneously during simulations. Different sets of the same type of particle can also be generated, for example an electron beam and its halo. Particle sets are typically created using the **`gptgetparset`** function, because this function also works properly if the specified set already exists.

During initialization, the coordinates of particles already added to a set can be modified after **`gptgetparsetpars`** is called. The function **`gptgetdistribution`** can be used to generate specific distributions like the built-in start elements. New particles can be added using the **`gptaddparmqn`** function.

New particles can also be added to sets during the simulation using the same **`gptaddparmqn`** function. When the timestep succeeds, they are added to the "master-list" and will be traced from the beginning of the next timestep. When the timestep fails, they are removed from memory.

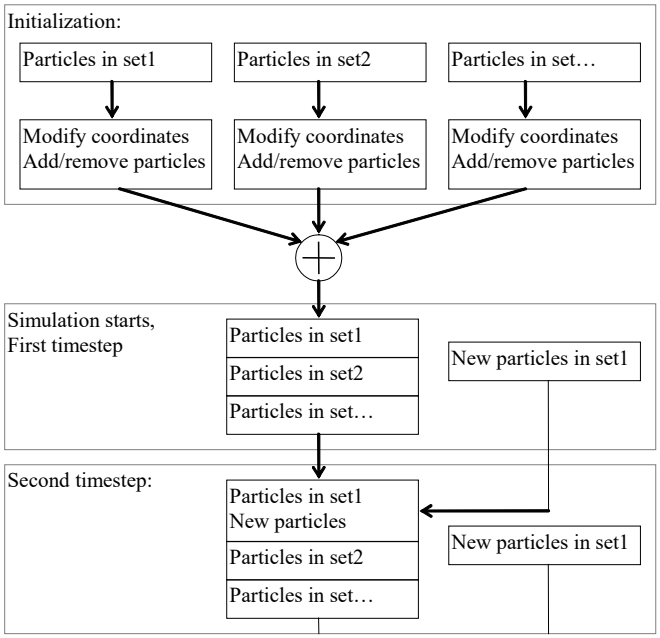


Figure 2-1: Using particle sets.

Listed below are the functions that can be used in GPT elements to manipulate particle sets.

2.7.1 gptaddparmqn

void gptaddparmqn(gptparset *ppp, double *Wr, double *WGBr, double m, double q, double n)

Adds a particle to a particle set.

ppp	Pointer to a previously created particle set.
Wr	Array of position coordinates (x, y, z).
WGBr	Array of momentum coordinates ($\gamma\beta x$, $\gamma\beta$y, $\gamma\beta z$).
m	Mass of the elementary particle.
q	Charge of the elementary particle.
n	Number of elementary particles this macro-particle represents.

This is the most general function to add a particle to a particle set. To add more than one particle, call this function as often as needed.

Example: Add one particle to the specified particle set.

```
1.  /* setstrpa.c - Start a single particle in a set */
2.  #include <stdio.h>
3.  #include <math.h>
4.  #include "elem.h"
5.
6.  void setstartpar_init(gptinit *init)
7.  {
8.    int numarg ;
9.    double r[3], GBr[3] ;
10.   double m,q,n ;
11.   gptparset *set ;
12.
13.   numarg = gptgetargnum(init) ;
14.   if( numarg!=7 && numarg!=10 )
15.     gpterror( "Syntax: %s(set,x,y,z,GBx,GBy,GBz,[m,q,n])\n", gptgetname(init) ) ;
16.
17.     set = gptgetparset( gptgetargstring(init,1) ) ;
18.     r[0] = gptgetargdouble(init,2) ;
19.     r[1] = gptgetargdouble(init,3) ;
20.     r[2] = gptgetargdouble(init,4) ;
21.   GBr[0] = gptgetargdouble(init,5) ;
22.   GBr[1] = gptgetargdouble(init,6) ;
23.   GBr[2] = gptgetargdouble(init,7) ;
24.
25.   if( numarg==7 )
26.   {
27.     gptaddpar(set,r,GBr) ;
28.   } else
29.   {
30.     m = gptgetargdouble(init,8) ;
31.     q = gptgetargdouble(init,9) ;
32.     n = gptgetargdouble(init,10) ;
33.     gptaddparmqn(set,r,GBr,m,q,n) ;
34.   }
35. }
```

2.7.2 gptaddpar

`void gptaddpar(gptparset *ppp, double *Wr, double *WGBr)`
Adds a particle to a particle set. Uses the default values for m and q and sets n to zero.

`ppp`	Pointer to a previously created particle set.
`Wr`	Array of position coordinates (x, y, z).
`WGBr`	Array of momentum coordinates ($\gamma\beta x$, $\gamma\beta$y, $\gamma\beta z$).

This function is equivalent to

```
1. double m, q ;
2. gptgetvaldouble("m",&m) ;
3. gptgetvaldouble("q",&q) )
4. gptaddparmqn( ppp, Wr ,WGBr, m, q, 0.0 ) ;
```

2.7.3 gptaddparmq

`void gptaddparmq(gptparset *ppp, double *Wr, double *WGBr, double m, double q)`
Adds a particle to a particle set. Sets n to zero.

`ppp`	Pointer to a previously created particle set.
`Wr`	Array of position coordinates (x, y, z).
`WGBr`	Array of momentum coordinates ($\gamma\beta x$, $\gamma\beta$y, $\gamma\beta z$).
`m`	Mass of the elementary particle.
`q`	Charge of the elementary particle.

This function is equivalent to

```
1. gptaddparmqn( ppp, Wr ,WGBr, m, q, 0.0 ) ;
```

2.7.4 gptcreateparset

`gptparset *gptcreateparset(const char *name)`
Creates a named particle set.

`name`	Name of the particle set. It is not recommended to use tabs, spaces or any non-printable characters in the name.
Return value:	A pointer to the newly created particle set.

Example: Creates a particle set named “beamhalo”.

```
1. gptparset *beamhalo ;
2. beamhalo = gptcreateparset( "beamhalo" ) ;
```

2.7.5 gptgetdistribution

```
std::vector<double> gptgetdistribution(gptinit *init, int len, int startparam,
  enum gptdistdim distdim, gptparset *set) ;
enum gptdistdim { gptdist_1D=100, gptdist_2DR, gptdist_2Dsin } ;
```

Use this function to generate a specific distribution like the built-in start elements.
Breaking change in parameters and return value in GPT version 3.3.

init	First paramter of the init function
startparam	Start location of distribution specification.
distdim	Distribution type. Must be one of: gptdist_1D, gptdist_2DR, gptdist_2Dsin.
set	Particle set.

Return value: Vector containing the requested distribution for the given set.

Before **gptgetdistribution** can be used, first a pointer to the particles in a set must be obtained using the functions **gptgetparset** and **gptgetparsetpars**. Then the specific distribution can be generated, where the length is automatically chosen based on the given **set**.

Example: **setzdist** source code.

```
1.  /* setzdist.c - Set longitudinal particle distribution */
2.
3.  #include <vector>
4.  using namespace std ;
5.
6.  #include <stdio.h>
7.  #include <math.h>
8.  #include <ctype.h>
9.  #include "elem.h"
10.
11. void setzdist_init(gptinit *init)
12. {
13.   gptparset *set ;
14.   gptinitpar *par ;
15.   const char *name ;
16.   int i, len ;
17.
18.   if( gptgetargnum(init)<2 )
19.     gpterror( "Syntax: %s(set,DIST)\n", gptgetname(init) ) ;
20.
21.   name = gptgetargstring(init,1) ;
22.
23.   /* Get particle set */
24.   if( gpttestparset( name )==NULL )
25.     gptwarning( "The particle set \"%s\" does not exist\n", name ) ;
26.   set = gptgetparset( name ) ;
27.   par = gptgetparsetpars( set,&len ) ;
28.
29.   /* Set distribution */
30.   vector<double> dist = gptgetdistribution(init,2,gptdist_1D,set) ;
31.   for( i=0 ; i<len ; i++ ) par[i].Wr[2] = dist[i] ;
32. }
```

2.7.6 gptgetparset

gptparset *gptgetparset(const char *name)
Returns a pointer to the specified particle set or creates a new set.

name	Name of the particle set.
Return value:	Pointer to the specified particle set. A new set is created if the specified particle set doesn't exist.

Example: Creates and removes the "beamhalo" particle set.

```
1. gptparset *ppp ;
2. ppp = gptgetparset( "beamhalo" ) ;
3. …
4. gptremoveparset( ppp ) ;
```

2.7.7 gptgetparsetpars

gptinitpar *gptgetparsetpars(gptparset *ppp, int *len)
Returns a pointer to the particles and the number of particles in the specified set.

ppp	Pointer to a previously created particle set.
len	Number of particles in the set.
Return value:	Pointer to the first particle within the set, see section 2.7.1 on page 33.

This function needs to be called to be able to modify the particle coordinates within a particle set. Please note that the location in memory of the particles in the set is subject to change when particles are added or removed. Therefore, an other call to **gptgetparsetpars** can be necessary when the particle coordinates are accessed or modified after particles have been added or removed.

Example: Sets all the masses within a set to the specified value.

```
1.  void setmass_init(gptinit *init)
2.  {
3.    gptparset *ppp ;
4.    double newmass ;
5.    int i, len ;
6.    gptinitpar *pars ;
7.
8.    if( gptgetargnum(init)!=2 )
9.      gpterror( "Syntax: %s(setname,newmass)\n", gptgetname(init) ) ;
10.
11.   ppp = gptgetparset( gptgetargstring(init,1) ) ;
12.   newmass = gptgetargdouble(init,2) ;
13.
14.   pars = gptgetparsetpars(ppp,&len) ;
15.   for( i=0 ; i<len ; i++ ) pars[i].m = newmass ;
16. }
```

2.7.8 gptinitpar

typedef gptinitpar
Definition of a particle in a particle set that is not yet added to the master array.

```
1.  class gptinitpar
2.  {
3.    /* Basic particle coordinates */
4.    double Wr[3] ;
5.    double GBr[3] ;
6.    double m, q, n, r ;
7.    double tstart ;
8.
9.    /* Additional settings */
10.   axis_handle axis ;
11.   int ID ;
12. } ;
```

Where the following class members can be used:

Wr	Particle's x, y and z-coordinate [m], measures in WCS
GBr	Particle's normalized momentum in *x*, *y* and *z*-direction, measured in WCS.
m	Mass [kg] of elementary particle this macro-particle represents.
q	Charge [C] of elementary particle this macro-particle represents.
n	Number of elementary particles this macro-particle represents.
r	Radius [m] of macro-particle to be used in space-charge calculations.
tstart	Release time [s] of particle.
axis_handle	Initial CCS for the particle, default "wcs".
ID	Particle ID. If specified it must be unique across all MPI nodes.

2.7.9 gptremovepar

gptinitpar *gptremovepar(gptparset *ppp, int n, int *len) ;
Removes the specified particle from a particle set.

ppp	Pointer to a previously created particle set.
n	Index of the particle to be removed.
len	The number of particles within the set is stored in this variable.
Return value:	Pointer to the first particle within the set.

Please note that when particle number **n** is removed, an other particle is copied to the n^{th} position. Furthermore, the particle array can be copied to a different memory location when a particle is removed. Therefore it is advised to use the following method to remove particles:

```
pars = gptgetparsetpars(ppp,&len) ;
for(i=0 ; i<len ; i++)
  if( pars[i] … )
    pars = gptremovepar(ppp,i--,&len) ;
```

Example: Remove all particles when *z*<**zmin** or *z*>**zmax.**

```
1.  void foo_init(gptinit *init)
2.  {
3.    gptparset *ppp ;
4.    double zmin, zmax ;
5.    int i, len ;
6.    gptinitpar *pars ;
7.
8.    if( gptgetargnum(init)!=3 )
9.      gpterror( "Syntax: %s(setname,zmin,zmax)\n", gptgetname(init) ) ;
10.
11.   ppp = gptgetparset( gptgetargstring(init,1) ) ;
12.   zmin = gptgetargdouble(init,2) ;
13.   zmax = gptgetargdouble(init,3) ;
14.
15.   pars = gptgetparsetpars(ppp,&len) ;
16.   for(i=0 ; i<len ; i++)
17.     if( pars[i].Wr[2]<zmin || pars[i].Wr[2]>zmax )
18.       pars=gptremovepar(ppp,i--,&len)
19. }
```

2.7.10 gptremoveparset

`void gptremoveparset(gptparset *ppp)`
Removes an existing particle set.

`ppp` Pointer to a previously created particle set.

The particle set and all its particles are removed. The occupied memory is returned to the system.

Example: Creates and removes the “beamhalo” particle set.

```
1. gptparset *beamhalo ;
2. beamhalo = gptcreateparset( "beamhalo" ) ;
3. …
4. gptremoveparset( beamhalo ) ;
```

2.7.11 gpttestparset

`gptparset *gpttestparset(const char *name)`
Returns a pointer to the specified particle set or NULL if it is not present.

`name` Name of the particle set.
Return value: Pointer to the specified particle set. NULL if the specified particle set doesn’t exist.

Example: Displays a message if the “beamhalo” set doesn’t exist.

```
1. if( gpttestparset( "beamhalo" )==NULL )
2.   gptwarning( "The beamhalo set doesn't exist\n" ) ;
```

2.8 Vector operations

These kernel functions simplify the operations with vectors containing 3 coordinates. They are all implemented as macro's to improve run-time performance. Therefore the operand(s) should never contain **++** or **--** operations.

2.8.1 gptSQR

gptSQR(x)
Returns the square of its argument.

x Any numeric argument.
Return value: The square of the **x**

Implemented as:
```
#define gptSQR(x) ((x)*(x))
```

2.8.2 gptVECINP

gptVECINP(x,y)
Calculates the inproduct **x•y**

x Any vector containing three coordinates.
y Any vector containing three coordinates.
Return value: The inproduct of **x** and **y**.

Implemented as:
```
#define gptVECINP(a,b) ((a)[0]*(b)[0] + (a)[1]*(b)[1] + (a)[2]*(b)[2])
```

2.8.3 gptVECLEN

gptVECLEN(x)
Calculates the length of a vector: **|x|**

x Any vector containing three coordinates.
Return value: The length of **x**

Implemented as:
```
#define gptVECLEN(x) sqrt(gptVECSQR(x))
```

2.8.4 gptVECSQR

gptVECSQR(x)
Calculates the square of the length of a vector: **|x|²**

x Any vector containing three coordinates.
Return value: The square of the length of **x**

The kinetic energy **T** of a particle can for example be calculated as: $\mathbf{T} = (\gamma - 1)\, mc^2$
```
T=(sqrt(1.0+gptVECSQR(WGBr))-1.0)*m*gpt_c*gpt_c
```

Implemented as:
```
#define gptVECSQR(x) (gptSQR((x)[0])+gptSQR((x)[1])+gptSQR((x)[2]))
```

2.9 Coordinate transformations

The GPT kernel normally takes care of all required coordinate transformations. However, sometimes it is desirable to perform these operations “manually”.

The basic data type for a orthonormal Carthesian coordinate transformation is a **gpttransform**. Most functions in this section convert back and forth directions and positions based on a **gpttransform**.

The functions **gptr2carth** and **gptrphi2carth** convert from cylindrical-symmetric to Cartesian. Please always use these functions when possible because they are much more accurate and robust than a simple implementation.

2.9.1 gptadddirectiontoWCS

```
void gptadddirectiontoWCS( gpttransform *t, double *Bin, double *Bout ) ;
```

Adds a direction vector to WCS

t	Transformation to be applied
Bin	Input vector
Bout	Output vector

The performed calculation in matrix notation is:

$$\vec{B}_{out} = (M + I) \cdot \vec{B}_{in}$$

where I is the identity matrix.

2.9.2 gptconcattransform

```
void gptconcattransform( gpttransform *t, gpttransform *t1, gpttransform
*t2 ) ;
```

Multiplies two sequential gpttransforms into one direct gpttransform

t	Final transformation
t1	First transformation
t2	Second transformation

The performed calculation in matrix notation is:

$$\mathbf{M}_t = \mathbf{M}_{t1}\mathbf{M}_{t2}$$

$$\mathbf{o}_t = \mathbf{M}_{t1}\mathbf{o}_{t2} + \mathbf{o}_{t1}$$

2.9.3 gptdirectiontoUCS

```
void gptdirectiontoUCS( gpttransform *t, double *Bin, double *Bout ) ;
```

Transforms a direction vector to UCS

t	Transformation to be applied
Bin	Input vector
Bout	Output vector

The performed calculation in matrix notation is:

$$\vec{B}_{out} = M^{-1} \cdot \vec{B}_{in}$$

2.9.4 gptdirectiontoWCS

`void gptdirectiontoWCS( gpttransform *t, double *Bin, double *Bout ) ;`
Transforms a direction vector to WCS

`t`	Transformation to be applied
`Bin`	Input vector
`Bout`	Output vector

The performed calculation in matrix notation is:
$\vec{B}_{out} = M \cdot \vec{B}_{in}$

2.9.5 gptr2carth

`void gptr2carth(double Fr,double x,double y,double *Fx,double *Fy)`
Converts a radial vector to cartesian vectors.

`Fr`	Radial vector.
`x`	x- coordinate.
`y`	y-coordinate.
`Fx`	Calculated x-vector.
`Fy`	Calculated y-vector.

The performed equation is: $\begin{cases} Fx = Fr\cos(\varphi) = \dfrac{Fr \cdot x}{\sqrt{x^2 + y^2}} \\ Fy = Fr\sin(\varphi) = \dfrac{Fr \cdot y}{\sqrt{x^2 + y^2}} \end{cases}$

However, for a very small x or y, the above equations will cause numerical problems. Therefore, the equations are executed as:

Condition	**`Fx`**	**`Fy`**
x=0, y=0	0	0
$\lvert x\rvert > \lvert y\rvert$	$\dfrac{\text{sign}(x)}{\sqrt{1+(y/x)^2}}$	$\dfrac{\text{sign}(x)\cdot(y/x)}{\sqrt{1+(y/x)^2}}$
$\lvert x\rvert < \lvert y\rvert$	$\dfrac{\text{sign}(y)\cdot(x/y)}{\sqrt{1+(x/y)^2}}$	$\dfrac{\text{sign}(y)}{\sqrt{1+(x/y)^2}}$

2.9.6 gptrphi2carth

`void gptrphi2carth(double Fr,double Fphi,double x,double y,double *Fx,double *Fy)`
Converts radial and angular vectors to cartesian vectors.

`Fr`	Radial vector.
`Fphi`	Angular vector.
`x`	x-coordinate.
`y`	y-coordinate.
`Fx`	Calculated x-vector.
`Fy`	Calculated y-vector.

The performed equation is: $\begin{cases} Fx = Fr\cos(\varphi) - Fphi\sin(\varphi) \\ Fy = Fr\sin(\varphi) + Fphi\cos(\varphi) \end{cases}$

The equations are performed using **`gptr2carth`** to avoid numerical problems at small x or y values.

2.9.7 gpttoUCS

`void gpttoUCS( gpttransform *t, double *rin, double *rout ) ;`

Transforms a position vector to UCS

t	Transformation to be applied
rin	Input vector
rout	Output vector

The performed calculation in matrix notation is:
$\vec{r}_{out} = M^{-1} \cdot (\vec{r}_{in} - \vec{o})$

2.9.8 gpttoWCS

```
void gpttoWCS( gpttransform *t, double *rin, double *rout ) ;
```

Transforms a position vector to WCS

t	Transformation to be applied
rin	Input vector
rout	Output vector

The performed calculation in matrix notation is:
$\vec{r}_{out} = M \cdot \vec{r}_{in} + \vec{o}$

2.9.9 gpttransform

Basic data type for GPT carthesian coordinate transformations.

A gpttransform has two elements:

m[3][3]	The matrix describing a 3D rotation. It must be an orthonormal matrix.
o[3]	The vector describing a 3D translation.

The matrix **m** of a **gpttransform** is automatically checked to be the identity transform *I*. All functions using a **gpttransform** are optimized to use this information and will automatically replace the matrix-vector multiplication with a copy operation if possible.

2.10 Compatibility

This section lists functions that have become obsolete, and functions that have been significantly changed or removed in this version. Please use the alternative functions instead when available.

Maintaining compatibility with previous releases enables users to use the newest features directly when they become available. However, for large projects such as GPT, maintaining compatibility is difficult to achieve without compromising efficiency. Therefore we urge users not to use the functions listed below. Their purpose is solely to keep old custom code useable.

To our regret GPT version 3.3 has a relatively large number of breaking changes compared to its predecessor. The main reason for that is that the 3.3 release is an MPI version with different memory management: All pointers to memory locations had to be rewritten in terms of 'handles' that are consistent across all nodes.

2.10.1 odemtaddfprfunction

```
void odemtaddfprfunction( int position, odemtfprfnc func, void *info ) ;
typedef void (*odemtfprfnc)(double t, double *x, double *p, void *info, int
nOff, int nCPU) ;
```

Multi-threaded version of **odeaddfprfunction** to be used on multi-processor systems.
Removed in GPT version 3.2. Please use the openMP **#pragma omp parallel for** instead.

2.10.2 gptcreatesyncarray

```
gptsyncarray *gptcreatesyncarray(gptsyncset *psp, size_t blocksize,
    void (*notifyfunc)(gptsyncarray *psa, gptsyncstatus stat, int n, void
*info),
    void *info) ;
```

Creates a synchronized array in a synchronization set.
Removed in GPT version 3.3.

2.10.3 gptcreatesyncitem

```
void gptcreatesyncitem(gptsyncset *psp) ;
```

Adds an item to a synchronization set.
Removed in GPT version 3.3.

2.10.4 gptcreatesyncset

```
gptsyncset *gptcreatesyncset(int itemcount, int chunkcount) ;
```

Creates a synchronization set.
Removed in GPT version 3.3.

2.10.5 gptremovesyncarray

```
int gptremovesyncarray(gptsyncset *psp, gptsyncarray *psa) ;
```

Removes a synchronized array in a synchronization set.
Removed in GPT version 3.3.

2.10.6 gptremovesyncitem

```
void gptremovesyncitem(gptsyncset *psp, int n) ;
```

Adds an item to a synchronization set.
Removed in GPT version 3.3.

2.10.7 gptremovesyncset

```
int gptremovesyncset(gptsyncset *psp) ;
```

Removes a synchronization set and all contained arrays.
Removed in GPT version 3.3.

2.10.8 gptsyncset

typedef gptsyncset
Definition of a set of arrays to be synchronized.
Removed in GPT version 3.3.

2.10.9 gptsyncstatus

typedef enum gptsyncstatus
Passed as parameter to notification functions in case of a synchronization event.
Removed in GPT version 3.3.

2.10.10 gptaddparticle

```
void gptaddparticle(gptinit *init, double *r,
  double *Br) ;
```

Adds a particle to the list of macro particles to be traced in the “beam” particle set.
Please use **gptaddparmqn** and related functions instead.

init	First argument of the init function.
r	Array containing the particle’s position.
Br	Array containing the particle’s normalized velocity.

This function can only be called from within the initialization function and **gptbuildECS** must be called first.

2.10.11 gptaddparticleGB

```
void gptaddparticleGB(gptinit *init, double *r,
  double *GBr) ;
```

Adds a particle to the list of macro particles to be traced in the “beam” particle set.
Please use **gptaddpar** and related functions instead.

init	First argument of the init function.
r	Array containing the particle’s position.
GBr	Array containing the particle’s normalized momentum.

This function can only be called from within the initialization function and **gptbuildECS** must be called first.

2.11 GDF Output functions

This section describes the functions for writing additional information in the GPT outputfile. They provide a simple interface between GPT and the GDF outputfile. Especially, the **gptoutputdouble** function used in combination with **gdfmgetval** function of a custom GDFA Prog can be very powerful.

2.11.1 gptoutputdouble

void gptoutputdouble(const char *name, double x)
Writes a floating point value in the GPT outputfile.

name	Name of the value to be written.
x	Corresponding floating point value.

The written variable can be read from the output GDF2A of using the GDFA function **gdfmgetval**.

Example: Write “pi” in the GPT outputfile.

```
1. gptoutputdouble( "pi", gpt_pi ) ;
```

2.11.2 gptoutputdoublearray

void gptoutputdoublearray(const char *name, double *x, int len)
Writes a fixed-length array of floating point values in the GPT outputfile.

name	Name of the array to be written.
x	Corresponding array of floating point values.
len	Length of array.

Example: Write a linear ramp in the GPT outputfile.

```
1. #define COUNT 100
2. double x[COUNT] ;
3. int i ;
4.
5. for(i=0 ; i<COUNT ; i++) x[i] = 0.01 * i ;
6. gptoutputdoublearray("ramp",x,COUNT)
```

2.11.3 gptoutputdoublegroup

void gptoutputdoublegroup(const char *name, double x)
Starts a new group in the GPT outputfile, defined by a floating point value.

name	Name of the group to be started.
x	Corresponding floating point value.

This function is used internally by the GPT kernel to write time and position output. When may arrays and variables need to be written, it can be useful to group them in a separate group. A new group must always be terminated by using the **gptoutputendgroup** function.

Example:

```
1.  double stat1, stat2 ;
2.  double *stats[COUNT] ;
3.
4.  gptoutputdoublegroup( "statistics", 0 ) ;
5.
6.  gptoutputdouble( "stat1", stat1 ) ;
7.  gptoutputdouble( "stat2", stat2 ) ;
8.  gptoutputdoublearray( "stats", stats, COUNT ) ;
9.
10. gptoutputendgroup() ; /* This ends the "statistics" group in the GPT outputfile */
```

2.11.4 gptoutputendgroup

void gptoutputendgroup(void)
Ends a previously created group in the GPT outputfile. This group is typically created using **gptoutputdoublegroup**.

2.12 Differential equations

The GPT differential equation solver consists of two parts:

- The ordinary differential equation (ODE) manager.
- The Runge-Kutta driver.

The ODE manager is responsible for maintaining the lists of functions to be called at the various stages of the integration process. The Runge-Kutta driver is responsible for performing the actual timesteps and calculating the stepsizes. It makes use of the same interface provided by the ODE manager. The separations between the ODE manager and the Runge-Kutta driver allows future compatibility with different integration scenarios.

The following interfaces are used to communicate with the ODE manager:

Interface	Kernel function	Description
INI	**`odeaddinifunction`**	Initialization before a timestep.
FPR	**`odeaddfprfunction`**	Calculation of the derivatives of all variables.
FPR	**`odemtaddfprfunction`**	Multi-threaded **`odeaddfprfunction`**
OUT	**`odeaddoutfunction`**	Output functions. Derivative information is already present.
ERR	**`odeadderrfunction`**	Calculation of the scaled error for all variables.
END	**`odeaddendfunction`**	Initialization after a successful timestep.

The **`odeaddvar`** function defines the interface for adding and removing differential equations before or during a simulation. After this function has been called, the gray areas in Figure 2-2 can be used for the calculation of derivatives and the optional output of the integrated values. If required, the specific accuracy of the added variables can also be defined.

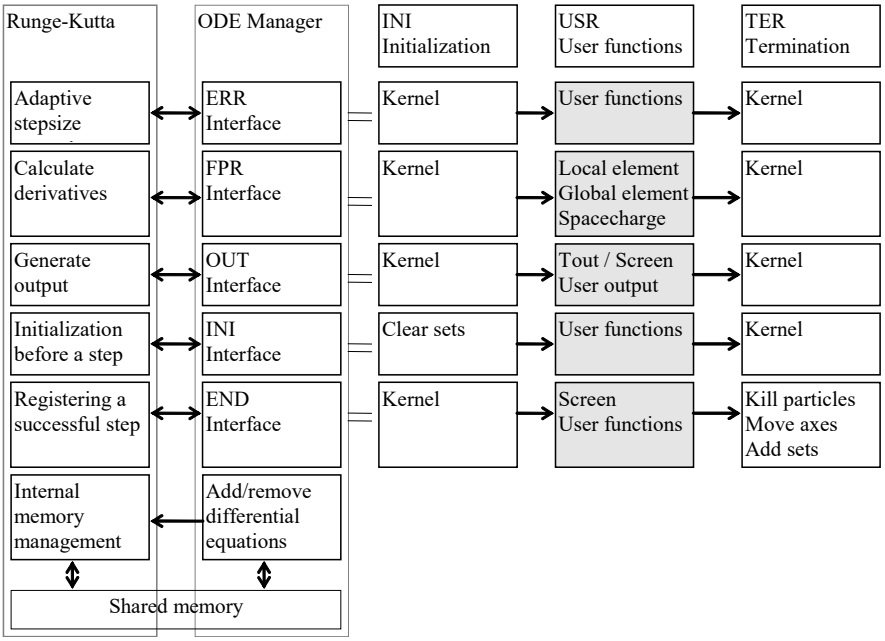


Figure 2-2: Implementation of differential equations in GPT. The gray areas denote parts of the GPT kernel that can be modified by the custom elements.

2.12.1 odeaddendfunction

```
void odeaddendfunction( int position, odeendfnc func, void *info ) ;
typedef void (*odeendfnc)(double tstart, double tend, double dt, const double
*xstart, const double *xend, void *info, void *stepinfo) ;
```

Adds a function to the END interface of the ODE manager.
Breaking changes in parameter list in GPT version 3.3.

positions	**ODEFNC_USR** must be specified.
func	User defined function to be called.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:

```
void func(double t, double *dt, double *x, void *info) ;
```

tstart	Simulation time at the start of the timestep.
tend	Simulation time.at the end of the timestep.
dt	Maximum timestep of next step, NOT adjustable anymore since GPT version 3.3.
xstart	Pointer to array of ODE variables at the start of the timestep.
xend	Pointer to array of ODE variables at the end of the timestep.
info	Any additional information. The info parameter of **odeaddendfunction**.

2.12.2 odeadderrfunction

```
void odeadderrfunction( int position, odeerrfnc func, void *info ) ;
typedef double (*odeerrfnc)(double t, double dt, const double *xstart, const
double *xend, const double *xerr, void *info) ;
```

Adds a function to the ERR interface of the ODE manager.
Breaking changes in parameter list (**const** added) in GPT version 3.3.

positions	**ODEFNC_USR** must be specified.
func	User defined function to be called.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:

```
double func(double t, double dt, double *xstart, double *xend, double *xerr,
void *info) ;
```

t	Simulation time.
dt	Maximum timestep of present step.
xstart	Pointer to array of ODE variables at the beginning of the timestep.
xend	Pointer to array of ODE variables at the end of the timestep.
xerr	Pointer to the corresponding estimated error in ODE variables.
info	Any additional information. The info parameter of **odeadderrfunction**.

Return value: Scaled error information. The timestep will fail if the returned value is larger than 1. This return value is also used in the calculation of the next timestep.

2.12.3 odeaddfprfunction

```
void odeaddfprfunction( int position, odefprfnc func, void *info ) ;
typedef void (*odefprfnc)(double t, const double *x, double *p, void *info) ;
```

Adds a function to the FPR interface of the ODE manager.
Breaking change in parameter list (**const** added) in GPT version 3.3.

positions	**ODEFNC_USR** must be specified.
func	User defined function to be called.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:

```
void func(double t, double *x, double *p, void *info) ;
```

t	Simulation time.
x	Pointer to array of ODE variables.
p	Pointer to array of derivatives to be calculated.
info	Any additional information. The info parameter of **odeaddfprfunction**.

2.12.4 odeaddinifunction

```
void odeaddinifunction( int position, odeinifnc func, void *info ) ;
typedef void (*odeinifnc)(double t, double *x, struct odeinfo *odeinfo, void
*info) ;
```

Adds a function to the INI interface of the ODE manager.
Breaking change in parameter list in GPT version 3.3.

positions	**ODEFNC_USR** must be specified.
func	User defined function to be called.
odeinfo	Internal use..
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:

```
void func(double t, double *x, void *info) ;
```

t	Simulation time.
x	Pointer to array of ODE variables.
info	Any additional information. The info parameter of **odeaddininfunction**.

Example: Print a dot at the beginning of each timestep.

```
1. void myfunction(double t, double *x, void *odeinfo, void *info)
2. {
3.   fprintf( stderr, "." ) ;
4. }
5.
6. odeaddinifunction(ODEFNC_USR,myfunction,NULL) ;
```

2.12.5 odeaddoutfunction

```
void odeaddoutfunction( int position, odeoutfnc func, void *info ) ;
typedef int (*odeoutfnc)(double t, double *dt, const double *x, void *info) ;
```

Adds a function to the OUT interface of the ODE manager.
Breaking change in parameter (**const** added) in GPT version 3.3.

positions	**ODEFNC_USR** must be specified.
func	User defined function to be called.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:

```
int func(double t, double *dt, double *x, void *info) ;
```

t	Simulation time.
dt	Maximum timestep of present step.
x	Pointer to array of ODE variables.
info	Any additional information. The info parameter of **odeaddoutfunction**.

2.12.6 odeaddvar

```
int odeaddvar( struct odeinfo *odeinfo, int num, int *offset )
```
Add one or more variables to the ODE solver.

odeinfo	Must be specified as **&odeinfo**
num	Number of variables to add
offset	Stored offset in the variables array
Return value	Nonzero on error

The **odeaddvar** function maintains a list of variables. Because this list is not stored at a fixed position in memory, the offset in the list defines a specific variable. For example, when two variables are added and offset is set at 23 at return, the requested variables are numbered 23 and 24. When these variables need to have an initial value of 3 and 4 respectively, the following lines can be used:

```
1. int offset
2. odeaddvar( &odeinfo, 2, &offset ) ;
3. odeinfo.x[offset+0] = 3 ;
4. odeinfo.x[offset+1] = 4 ;
```

Typically the offset must be stored in the info structure of the element, because the are required by other functions as well. The **odeaddfpr** function must be used to define the differential equations, with respect to time, to be solved.

2.12.7 simparaddaddfnc

```
template<typename T>
void simparaddaddfnc(simparaddfnc<T> fnc, T *info);
using simparaddfnc = void(*)(const gptpar &par, T *info);
```
Creates a callback when a particle is about to be tracked.

func	User defined function to be called when a particle is about to be tracked.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:
```
void func(const gptpar &par, __datatype__ *info) ;
```

par	Fully initialized particle, including particle ID.
info	Any additional information. The datatype must match the datatype of the info parameter of **simparaddaddfnc**.

After calling **simparaddaddfnc**, the function **func** is called whenever a new particle is about to be tracked. Internally this function is used in the **acceptance** element to record the initial coordinates of the particles.

2.12.8 toutaddfnc

```
void toutaddfnc( toutfnc func, void *info ) ;
typedef void (*toutfnc)(double t, double dt, const double *x, const
   gpttransform *tf, bool fields, void *info) ;
```
Adds a function writing additional output in the time group of the GPT outputfile
Breaking change: Parameter (**const** added) in GPT version 3.30.
Breaking change: Coordinate system added in GPT version 3.36

func	User defined function to be called.
info	Pointer to any additional information. It is passed to **func** as an argument.

func must be declared as:
```
void func(double t, double dt, const double *x, const gpttransform *tf, bool
   fields, void *info) ;
```

t	Simulation time.
dt	Maximum timestep of present step.
x	Pointer to array of ODE variables.
tf	Coordinate transform.

fields	Flag to indicate of electromagnetic field information is present: **true** for tout output, **false** for snapshots.
info	Any additional information. The info parameter of **odeaddoutfunction**.

After calling **toutaddfnc**, the function **func** is called whenever time output is written to the GPT outputfile. Normally, the **gptoutputdouble** function is used within **func** to write the values of additional differential variables.

2.13 Space-charge elements

Writing custom GPT space-charge elements is typically more complicated than writing a custom element specifying electromagnetic fields. Because an Element Coordinate System (ECS) can not be specified, space-charge elements act directly in the World Coordinate System. For efficiency reasons and flexibility, looping over all particles must be performed within the element allowing "mesh based" space-charge models to be written using the same interface.

The **odeaddfpr** function as described in section 2.12.1 on page 49 must be used to calculate the space-charge fields. The position parameter must be specified as **ODEFNC_USR**. It is also possible to sole additional differential equations from within a space-charge model.

2.13.1 Example

As an example, we will describe the creation of a custom space-charge element with very simple equations. We will call it:

```
spacecharge3Dexample([zmin,zmax]) ;
```

The model is a point-to-point calculation of classical coulomb interaction. The equation for the electric field at the position of particle *i* due to all point charges *j* is given by:

$$\mathbf{E}_i = \sum_{j \neq i} \frac{Q_j}{4\pi\varepsilon_0} \frac{\mathbf{r}_i - \mathbf{r}_j}{\left|\mathbf{r}_i - \mathbf{r}_j\right|^3}$$

As long as the velocities of all particles remain far from relativistic, this model is correct. The complete listing for this element is given in Listing 2-ii.

The initialization routine is very similar to any other custom GPT element. The range is limited to **zmin**<*z*<**zmax.** When the number of arguments is 2, **zmin** and **zmax** are stored in the info structure. When **zmin** and **zmax** are not specified, the **DBL_MIN** constant is used representing infinity. The only new line in the initialization routine is the last line:

```
odeaddfprfunction( ODEFNC_USR, sc_sim,info ) ;
```

This instructs GPT to call the space-charge routine **sc_sim** during the tracking process. The name **sc_sim** is just used as an example here, the name can be freely chosen as long as it is used consistently throughout the element.

sc_sim itself calculates the space-charge forces, using the parameters stored in the info structure. It makes use of the particle array **pars** containing all particle coordinates. The structure members of **pars** are listed in section 2.13.2. The number of particles currently present in the simulation is stored in the variable **numpar**. Please remember that **numpar** is not a constant during the simulations: Particles can be added or removed.

The **sc_sim** code loops over all particle coordinates twice. For all particles **i**, it loops over all particles **j** to add the resulting electric fields. Only alive and particles within **zmin** and **zmax** are taken into account. The total charge of macro-particle *j* is calculated by **pars[j].q*pars[j].n**. This is the charge of each elementary particles represented by macro-particle *j*, multiplied by the amount of elementary particles represented by macro-particle *j*.

Please always add '**+=**' electric or magnetic fields contributions. Assigning '**=**' a field surely invalidates the field calculation of standard elements. The relatively unknown **continue** statement stops all further processing in the while loop and starts the next iteration. In this context it can be interpreted as: "Skip this particle and go to the next", precisely what is needed for **zmin**,**zmax** boundaries and "alive" checks.

Listing 2-ii: Example custom space-charge routine: **sc3dexpl.c**.

```
/* sc3dexpl.c - 3D Point-to-point space-Charge routine. Example only. */

#include <stdio.h>
#include <math.h>
#include <float.h>
#include "elem.h"

#define ffac (1/(4*gpt_pi*gpt_eps0))

struct sc_info
{
  double zmin ;
  double zmax ;
} ;

static void sc_sim( double t, double *x, double *p, void *info ) ;


void spacecharge3dexample_init(gptinit *init)
{
  struct sc_info *info ;
  int numarg ;

  numarg = gptgetargnum(init) ;
  if( numarg!=0 && numarg!=2 )
    gpterror( "Syntax: %s([zmin,zmax])", gptgetname(init) ) ;

  info = gptmalloc( sizeof(struct sc_info) ) ;

  if( numarg==2 )
  {
    info->zmin=gptgetargdouble(init,1) ;
    info->zmax=gptgetargdouble(init,2) ;
  } else
  {
    info->zmin=-DBL_MAX ;
    info->zmax= DBL_MAX ;
  }

  odeaddfprfunction( ODEFNC_USR, sc_sim,info ) ;
}

static void sc_sim( double t, double *x, double *p, void *vinfo )
{
  unsigned int i, j ;
  struct sc_info *info = (struct sc_info *)vinfo ;

  double rji[3] ;
  double zmin, zmax, r2, r3, mfac ;

  zmin    = info->zmin ;
  zmax    = info->zmax ;

  for(i=0 ; i<numpar ; i++)
    if( pars[i].alive && pars[i].Wr[2] > zmin && pars[i].Wr[2] < zmax )
      for(j=0 ; j<numpar ; j++)
  {
    if( i==j || !pars[j].alive ) continue ;
    if( pars[j].Wr[2] < zmin || pars[j].Wr[2] > zmax ) continue ;

    rji[0] = pars[i].Wr[0] - pars[j].Wr[0] ;
    rji[1] = pars[i].Wr[1] - pars[j].Wr[1] ;
    rji[2] = pars[i].Wr[2] - pars[j].Wr[2] ;

    r2 = rji[0]*rji[0] + rji[1]*rji[1] + rji[2]*rji[2]  ;
    r3 = r2 * sqrt(r2) ;

    mfac = ffac*pars[j].q*pars[j].n/r3 ;

    pars[i].WE[0] += mfac*rji[0] ;
    pars[i].WE[1] += mfac*rji[1] ;
    pars[i].WE[2] += mfac*rji[2] ;
  }
}
```

2.13.2 pars array

Space-charge elements have direct access to the **pars** array. This is an array of particle structures with length **numpar**. Because the array is zero based, **pars[0]** till and including **pars[numpar-1]** can be accessed. The variable **numpar** should not be modified under any circumstance.

The actual particle coordinates are directly accessible through the member variable **Wr**. It contains the *x*, *y* and *z* coordinates of the particle, measured in the World Coordinate System (WCS). Custom Coordinate Systems (CCS) and Element Coordinate Systems (ECS) have no significance in space-charge models. The calculated electromagnetic fields due to the custom space-charge model should be added to the member variables **WE** and **WB**. Please never assign **WE** or **WB** a value. We strongly suggest to always use addition **+=** operator as used in **pars[i].WE[0] += Ej**.

When a particle is removed from the simulation, the **alive** member variable is set to 0.

The following structure members can be used, but should not be modified:

pars[i].Wr[0]	*x*-coordinate of particle *i* [m] in WCS.
pars[i].Wr[1]	*y*-coordinate of particle *i* [m] in WCS.
pars[i].Wr[2]	*z*-coordinate of particle *i* [m] in WCS.
pars[i].GBr[0]	Normalized momentum in *x*-direction of particle *i* in WCS.
pars[i].GBr[1]	Normalized momentum in *y*-direction of particle *i* in WCS.
pars[i].GBr[2]	Normalized momentum in *z*-direction of particle *i* in WCS.
pars[i].G	Lorentz factor γ of particle *i*.
pars[i].n	Number of elementary particles represented by particle *i*.
pars[i].q	Charge [C] of the elementary particles represented by particle *i*.
pars[i].m	Mass [kg] of the elementary particles represented by particle *i*.
pars[i].r2	Square of radius of macro particle *i* [m^2]. Storing the square of the radius typically saves a **sqrt** operation within the inner loop of space-charge routines, hereby saving valuable CPU time.
pars[i].alive	Nonzero (true) when the particle is not removed from the simulation. Zero (false) otherwise.
Pars[i].ID	Identification number of the particle. This number remains always the same, even though the array index **i** might change due to the removal of other particles.

The electromagnetic fields can be modified, but fields should be added to the already present value:

pars[i].WE[0]	x-component of the electric field at the position of particle *i*.
pars[i].WE[1]	y-component of the electric field at the position of particle *i.*
pars[i].WE[2]	z-component of the electric field at the position of particle *i*.
pars[i].WB[0]	x-component of the magnetic field at the position of particle *i*.
pars[i].WB[1]	y-component of the magnetic field at the position of particle *i*.
pars[i].WB[2]	z-component of the magnetic field at the position of particle *i*.

2.13.3 numpar

As indicated in the previous section, **numpar** counts the number of particles present in the simulation. This count includes both alive particles and particles removed from the simulation. However, when the GPT kernel reclaims memory occupied by removed particles, the particle array is shrunk and **numpar** decreased. Therefore, we suggest always using one of the following C constructions:

```
1. For(i=0 ; i<numpar ; i++) if( pars[i].alive) …
```

```
1. For(i=0 ; i<numpar ; i++)
2. {
3.   if( !pars[i].alive) continue ;
4.   …
5. }
```

These constructions always work properly, but for very specialized space-charge routines there is an other potential problem. When the GPT kernel reclaims memory, the last elements in the **pars** array are copied over the removed particles. This saves CPU time, but changes the order of the particles in the **pars** array. The particle **ID** must be used to uniquely identify a particle.

2.14 Boundary elements

The code for a boundary element is very similar to that of a custom element providing custom electromagnetic fields. As an example, the code for a rectangular plate at z=0 with dimensions |x|<a/2 and |y|<b/2 is explained below.

The start is as usual:

```
1.  /* plate.c - Define a rectangular plate boundary */
2.
3.  #include <stdio.h>
4.  #include <math.h>
5.  #include "elem.h"
```

The info structure contains just the dimensions of the plate, **a** and **b**.

```
7.  struct plate_info
8.  {
9.    double a ;
10.   double b ;
11. } ;
```

Next comes the forward declaration of the actual boundary-checking function. By convention, the name of this function is the name of the element appended by **_bound**.

```
13. static int plate_bound(gptpar *par, double t, double dt, gpttrajectory *traj,
      struct plate_info *info ) ;
```

The initialization routine is similar to the init routine of a custom electromagnetic element, except that **gptaddboundaryelement** is called at the end. This instructs the GPT kernel to call the specified boundary function for every line-segment in all particle trajectories. The info structure containing the parameters of the element is passed along.

```
15. void scatterplate_init(gptinit *init)
16. {
17.   struct plate_info *info ;
18.
19.   gptbuildECS( init ) ;
20.
21.   if(gptgetargnum(init)!=2 )
22.     gpterror( "Syntax: %s(ECS,a,b)\n", gptgetname(init) ) ;
23.
24.   info = gptmalloc( sizeof(struct plate_info) ) ;
25.
26.   info->a = gptgetargdouble(init,1) ;
27.   info->b = gptgetargdouble(init,2) ;
28.
29.   gptaddboundaryelement( init, plate_bound, gptfree, 0, info ) ;
30. }
```

Now comes the interesting part of the element: The actual boundary function. The most important parameter of this function is the particle-trajectory **traj**, serving as the main communication method between this function and the GPT kernel. On input for the boundary function, it contains the following structure members:

rstart[3]	Start of line-segment, converted to the ECS.
rend[3]	End of line-segment, converted to the ECS.
dr[3]	Shortcut for **rend-rstart**.
lambda	Lambda of previous boundary, see below.

Every boundary function must first calculate its **lambda**, defined as the fraction of the line segment that has crossed (if at all) the boundary. When there is no intersection, the boundary function should just return 0. For a plate located at z=**zp**, **lambda** is given by: **lambda=(zp-rstart[2])/dr[2]**.

If **lambda** is between 0 and 1, the intersection point **P** is defined by **P=rstart+lambda*dr**. After this point is calculated, it can be checked if it falls within the plate dimensions. If not, the boundary function returns 0.

When it is clear the trajectory crosses the plate, the following **traj** structure members must be set:

lamda	Calculated lambda.
P[3]	Intersection point.
n[3]	Surface normal at intersection point.

A problem arises when two boundaries are crossed within a single line-segment. In that case the boundary with the smallest lambda it the surface actually is hit. Therefore, the boundary function must return 0 directly when **traj->lamba<lambda**.

The code below shows this all in action. It is important to note that the **traj** structure members are only overwritten after it is absolutely clear that this boundary crosses the line segment and no other boundary comes first.

```
32. static int plate_bound(gptpar *par, double t, double dt, gpttrajectory *traj,
      struct plate_info *info )
33. {
34.   int i ;
35.   double lambda ;
36.   double P[3] ;
37.
38.   /* Calculate intersection point */
39.   lambda = -traj->rstart[2]/traj->dr[2] ;
40.   if( lambda > traj->lambda ) return( 0 ) ;
41.   for(i=0 ; i<3 ; i++) P[i] = traj->rstart[i] + lambda*traj->dr[i] ;
42.
43.   /* Test if P is within plate dimensions */
44.   if( fabs(P[0]) >= info->a/2 || fabs(P[1]) >= info->b/2 ) return( 0 ) ;
45.
46.   /* Store result */
47.   for(i=0 ; i<3 ; i++) traj->P[i] = P[i] ;
48.   traj->lambda = lambda ;
49.   traj->n[0] =  0.0 ;
50.   traj->n[1] =  0.0 ;
51.   traj->n[2] = -1.0 ;
52.
53.   return( 1 ) ;
54. }
```

Ambitious users can refer to the **t** parameter of the boundary function, defined as the simulation time in seconds, to create boundaries moving or changing shape as function of time.

2.14.1 gptaddboundaryelement

```
void gptaddboundaryelement( gptinit *init, gptboundaryfnc boundfnc,
  exitfn exitfnc, int type, void *info ) ;
```

Registers a boundary element.

init	First argument of the **_init** function.
boundfnc	Address of the function returning the fields, the simulation function.
exitfnc	Address of the function called once at the end of the simulation. Typically **gptfree**.
type	Maintained for future compatibility. Must be zero.
info	Pointer to any additional information. It is passed to both **boundfnc** and **exitfnc**.

This function adds a new boundary element to the simulation. The registered function **boundfnc** is called once for every line-segment in all particle trajectories. When the registered boundary element calculates an intersection between a line-segment and the boundary, the scatter model of the boundary material is called, see **gptinstallscatterfnc**. This scatter model is responsible for removing the incident particle and optionally creating (back)scattered particles. The **extfnc** is called once at the end of the simulation.

boundfnc must be declared as:

```
int gptboundaryfnc(gptpar *par, double t, double dt, gpttrajectory *traj,
  void *info ) ;
```

par	Particle travelling along the trajectory.
t	Simulation time [s]
dt	Timestep [d]. This parameter is generally not important.
traj	Particle trajectory information, see section 2.14.2 for details.
info	Info structure passed to **gptaddboundaryelement**.
Return value:	1 of the trajecotry crosses the bounndary, 0 otherwise.

exitfnc must be declared as:

```
void exitfnc(void *info) ;
```

info Info structure passed to **gptaddboundaryelement**.

2.14.2 gpttrajectory

Communication method between the GPT kernel and boundary/scattering elements.

```
typedef struct gpttrajectory
{
  double rstart[3], rend[3] ;
  double dr[3], ndr[3] ;
  double lambda ;
  double P[3], n[3] ;
  double inp, GBint[3], Gint ;

  struct gptboundaryelem *pelem ;
} gpttrajectory ;
```

Read parameters for a boundary element:

rstart[3] X, Y, and Z coordinates at start of trajectory in ECS.
rend[3] X, Y, and Z coordinates at end of trajectory in ECS.
dr[3] Shortcut for **rstart-rend**. Particle direction in ECS.

Write parameters for a boundary element:

lambda Fraction of the line segment intersecting the boundary. **lambda** is normalized between 0 and 1, where 0 corresponds to an intersection precisely at **rstart**. **lambda** and all other structure members may only be overwritten if the **lambda** of the boundary element is smaller than this variable.
P[3] Intersection point on the straight line between **rstart** and **rend** in ECS. **P=rstart+lambda*dr**
n[3] Surface normal at intersection point **P** in ECS. Does not need to have length 1.

Read parameters for a scatter element:

P[3] Intersection point in WCS.
n[3] **Normalized** surface normal in WCS.
dr[3] Particle direction in WCS. Please note that the **rstart** and **rend** members can not be used in a scatter element.
ndr[3] Normalized particle direction in WCS: **dr/|dr|**.
inp Inner product of the surface normal and particle direction. The angle of intersection can be calculated by α=acos(**inp**).
Gbint[3] Interpolated normalized momentum at the intersection point **P**.
Gint Interpolated Lorentz factor γ at the intersection point **P**.

2.15 Scatter elements

A number of helper routines can be used in a scatter element to provide most of the work. For these helper functions to work properly, a variable named **scatinfo** of type **struct scatter_info** must be declared in the info structure of the element.

2.15.1 gptinstallscatterfnc

```
struct symbol *gptinstallscatterfnc(const char *name, gptscatterfnc scatter,
  void *info) ;
```

Defines a new boundary material generating scatted particles.

name	Name of the surface material.
scatter	Scatter routine itself
info	Info structure to be passed to the scatter function.

The scatter function must be declared as:

```
void name_scat(gptpar *par,double t,double dt,
  struct gpttrajectory*trajectory, void *info) ;
```

par	Particle crossing the boundary
t	Simulation time [s]
dt	Timestep [s]
trajectory	Information about intersection point and surface normal, see section 2.14.2 on page 60.

Once a scatter function is installed using **gptinstallscatterfnc**, the scatter function is called for every line segment that crosses a boundary. It is recommended to use **gptscatterinit** in combination with **gptinstallscatterfnc**.

2.15.2 gptscatterinit

```
void gptscatterinit(gptinit *init, struct scatter_info *scatinfo, const char
*name) ;
```

Initialized the scatter statistics to be written to the GPT outputfile.

init	First parameter of the _init function
scatinfo	The address of the scatinfo structure within the elements' info structure.
name	Name of the surface material..

Must be used in combination with **gptinstallscatterfnc**.

Quite a large number of additional callback functions are registered by this function. For example an output routine and routines for updating scatter statistics at the beginning and end of a timestep. The **scatinfo** variable, typically declared as member of the elements' **info** structure, provides the means of communication between these functions.

Example: A typical example of the **_init** function of a scatter element is given below:

```
1.  Struct example_info
2.  {
3.  double P ;
4.
5.    struct scatter_info scatinfo ;
6.  }
7.
8.  void example_init(gptinit *init)
9.  {
10.   struct example_info *info ;
11.   const char *name ;
12.
13.   gptbuildECS( init ) ;
14.
15.   if( gptgetargnum(init)!=2 )
16.     gpterror( "Syntax: %s(ECS,name,P)\n", gptgetname(init) ) ;
17.
18.   info = gptmalloc( sizeof(struct forwscat_info) ) ;
19.
20.   name    = gptgetargstring(init,1) ;
21.   info->P = gptgetargdouble(init,2) ;
22.
23.   if( info->P<0 || info->P>1 )
24.     gpterror( "Probability P must be between 0 and 1." ) ;
25.
26.   /* Install all functions */
27.   gptscatterinit(init,&info->scatinfo,name) ;
28.   gptinstallscatterfnc(name,example_scat,info) ;
29. }
30.
31. void example_scat(gptpar *par,double t,double dt, gpttrajectory *ptraj,void *vinfo)
32. {
33.   ...
```

2.15.3 gptscatterinitpar

void gptscatterinitpar(struct scatter_info *scatinfo) ;
Initialized scatter statistics at the beginning of a **_scat** function.

scatinfo The address of the scatinfo structure within the elements' info structure.

This function must be the first function called in the **_scat** function as registerd by **gptscatterinit** It increases output buffers and initializes the total charge and energy in the **scatinfo** structure. Not calling this function can result in a GPT kernel crash.

The C notation for the address of a member of a pointer to a structure could be a problem for novice C programmers. When there is a:
struct example_info *info
with member variable **scatinfo** of type **struct scatter_info**, the correct notation is:
&info->scatinfo

2.15.4 gptscatteraddparmqn

void gptscatteraddparmqn(struct scatter_info *scatinfo,gptparset *ppp,
double *Wr, double *WGBr, double m, double q, double n) ;
Add a particle to the simulation from within a **_scat** function.

scatinfo The address of the **scatinfo** structure within the elements' info structure.
Other params See the documentation of **gptaddparmqn**.

Use this function to add a (back)scattered particle in a **_scat** function as registered by **gptscatterinit**. The function is identical to the **gptaddparmqn** function, but simultaneously updates scatter statistics. The **gptscatterinitpar** function must be called once before this function can be used.

2.15.5 gptscatterremoveparticle

void gptscatterremoveparticle(struct scatter_info *scatinfo,gpttrajectory
***traj,**
gptpar *par) ;

Removes a particle from the simulation from within a **_scat** function.

scatinfo	The address of the **scatinfo** structure within the elements' info structure.
traj	Particle trajectory information.
par	Particle to be removed.

Use this function to remove the incident particle in a **_scat** function as registered by **gptscatterinit** The function is identical to the **gptremoveparticle** function, but simultaneously updates the scatter statistics. The **gptscatterinitpar** function must be called once before this function can be

2.16 Miscellaneous

This section describes some miscellaneous functions, not related to any of the above categories. Generally these functions are for internal use or advanced users only.

2.16.1 dblequ

`int dblequ( double a, double b ) ;`
Tests if two floating point numbers are precisely or almost equal.

`a`	First argument
`b`	Second argument
Return value:	True if **`a`** and **`b`** are precisely or almost identical: $\begin{cases}\mathbf{a}(1+10\varepsilon)\geq\mathbf{b}\\ \mathbf{a}(1-10\varepsilon)\leq\mathbf{b}\end{cases}\text{or}\begin{cases}\mathbf{a}(1+10\varepsilon)\leq\mathbf{b}\\ \mathbf{a}(1-10\varepsilon)\geq\mathbf{b}\end{cases}$ where ε is the working precision of the machine. Effectively the last (10-based) digit is ignored in the comparison.

Testing if two floating point numbers are precisely equal is quite simple: "**`if(a==b)`**…" However, because of the finite working precision of floating point units, this expression usually is not correct in computer code. For example "**`if(1.1+1.2==2.3)`**…" will typically not work as desired. To test if two floating point numbers are "almost" equal, the **`dblequ`** function can be used: "**`if( dblequ(1.1+1.2,2.3) )`**…"

2.16.2 dblisint

`int dblisint(double x) ;`
Tests if a floating point value is (almost) an integer.

`x`	Floating point value to be rounded
Return value:	Nonzero if **`x`** is (almost) an integer: **`dblequ(dblroundtointdouble(x),x))`**.

Example:
`dblisint(+1.0)` returns **`1`**
`dblisint(+1.1)` returns **`0`**
`dblisint(-1.0)` returns **`1`**
`dblisint(-1.1)` returns **`0`**
`dblisint(1+ε)` returns **`1`**

2.16.3 dblrountoint

`int dblroundtoint(double x) ;`
Rounds a double value to the nearest integer.

`x`	Floating point value to be rounded.
Return value:	Nearest integer.

Example:
`dblroundtoint(+1.2)` returns **`+1`**
`dblroundtoint(+1.7)` returns **`+2`**
`dblroundtoint(-1.2)` returns **`-1`**
`dblroundtoint(-1.7)` returns **`-2`**

2.16.4 dblroundtointdouble

`double dblroundtointdouble(double x) ;`
Rounds a double value to the nearest integer, but returns this as a floating point number.

`x`	Floating point value to be rounded.
Return value:	Nearest integer returned as floating point number.

Example:
`dblroundtointdouble(+1.2)` returns **`+1.0`**
`dblroundtointdouble(+1.7)` returns **`+2.0`**
`dblroundtointdouble(-1.2)` returns **`-1.0`**
`dblroundtointdouble(-1.7)` returns **`-2.0`**

2.16.5 dblsolvequadratic

`int dblsolvequadratic(double a, double b, double c, double *result )`
Find the root of a second-order polynomial

`a`	See below. Can be zero.
`b`	See below.
`c`	See below.
`result`	Array for one or two solutions.
return value:	Number of solutions found

This function solves all real x from the expression:
$\mathbf{a}\,x^2 + \mathbf{b}\,x + \mathbf{c} = 0$

The equations used by **`dblsolvequadratic`** are [2:, pp. 183]
$$q = -\frac{1}{2}\left(\mathbf{b} + \operatorname{sgn}(\mathbf{b})\sqrt{\mathbf{b}^2 - 4\,\mathbf{ac}}\right)$$
$$x_1 = q/\mathbf{a}$$
$$x_2 = \mathbf{c}/q$$
When **`a`** is zero, the one and only solution is given by:
$$x = -\mathbf{c}/\mathbf{b}$$

Example:
```
1. double result[2] ;
2. int nsol ;
3.
4. nsol = dblsolvequadratic(1,-7,12,result) ;
5. for(i=0 ; i<nsol ; i++) printf( "%f\n", result[I] ) ;
```

Output:
```
1. 3
2. 4
```

2.16.6 dblsolvecubic

`int dblsolvecubic(double a, double b, double c, double d, double *result )`
Find the root of a third-order polynomial

`a`	See below. Can be zero.
`b`	See below. Can be zero.
`c`	See below.
`d`	See below.
`result`	Array for one, two or three solutions.
return value:	Number of solutions found.

This function solves all real x from the expression:
$\mathbf{a}\,x^3 + \mathbf{b}\,x^2 + \mathbf{c}\,x + \mathbf{d} = 0$

The equations used by **`dblsolvecubic`** are given in [2]. When **`a`** is zero, the calculations are delegated to the **`dblsolvequadratic`** function.

2.16.7 gptaddmainfunction

`void gptaddmainfunction( int position, void (*func)(void *info), void *info ) ;`
Use this function to specify another function to be called at a specified position in the main cycle.

`position`	Determines when in the main loop the specified function will be called. It must be one of the predefined listed in Table 2-C.
`func`	The function to be called at the specified **`position`**.
`info`	User defined information. The function **`func`** will be called with **`info`** as argument.

Table 2-C: Predefined position constants for the **`gptaddmainfunction`**.

Constant	Mode	Description
`GPTMAINFNC_STP`	Kernel	Commandline, inputfile parsing, open outputfile.
`GPTMAINFNC_INI`	User	Particles can still be added to sets. Sets can be modified.
`GPTMAINFNC_LST`	Kernel	Particle sets are transformed to master list.
`GPTMAINFNC_DYN`	Kernel	Dynamic memory management using syncsets starts for all particle related components.
`GPTMAINFNC_PAR`	User	Complete particle list is present. Last chance to modify particle distribution.
`GPTMAINFNC_SIM`	Kernel	Main simulation starts.
`GPTMAINFNC_TER`	User	Termination routines.
`GPTMAINFNC_EXT`	Kernel	Close outputfile and general cleanup.

2.16.8 gptgetvardouble

int gptgetvardouble(const char *name, double *val)
Obtains the value of a variable set in the inputfile or a predefined constant.

name	Name of the variable.
val	The value is stored in this variable.
Return value:	Zero if the variable is present, nonzero otherwise.

Using this function to pass parameters to a custom element is strongly not advised. Specifying the parameters on the command-line of the element always results is much clearer elements and inputfiles. Even when the same parameter needs to be specified several times.

Please note that some variables have a predefined value. Examples are m and q, which represent the mass and charge of an electron respectively.

Example: Start a particle with the default charge and mass having the specified energy.

```
1.  void foo_init(gptinit *init)
2.  {
3.    gptparset *ppp ;
4.    double mass, charge, gammabeta ;
5.    double Wr[3], WGBr[3] ;
6.
7.    if( gptgetargnum(init)!=2 )
8.      gpterror( "Syntax: %s(setname,gammabeta)\n", gptgetname(init) ) ;
9.
10.   ppp = gptgetparset( gptgetargstring(init,1) ) ;
11.   energy = gptgetargdouble(init,2) ;
12.
13.   if( gptgetvardouble("m",&mass)!=0 )
14.     gpterror( "Variable m is not present.\n" ) ;
15.   if( gptgetvardouble("m",&charge)!=0 )
16.     gpterror( "Variable q is not present.\n" ) ;
17.
18.   Wr[0] = Wr[1] = Wr[2] = 0.0 ;
19.   WGBr[0] = WGBr[1] = 0.0 ;
20.   WGBr[2] = gammabeta ;
21.
22.   gptaddpar(ppp,Wr,WGBr) ;
23. }
24.
```

3 Custom GDFA Programs

All GDFA data-analysis routines are designed for normal use. However, just like custom GPT elements, sometimes something different is required. In that case, it is possible to write custom GDFA analysis routines, named GDFA Programs or Progs.

This chapter describes the procedure to add custom GDFA progs on both a PC and a Linux machine.

3.1 Introduction

A custom GDFA program acts just like all built-in GDFA programs. It can calculate any statistic based on particle coordinates, velocities, electromagnetic fields or a combination of these. The generated interface code is essentially similar to the interface of custom GPT elements, as explained in section 1.4.

Every custom GDFA-Program has access to all arrays output by GPT: particle velocities, particle positions, electromagnetic fields at these positions and n, m and q of the macro-particles. Furthermore, a GDFA Program can read additional variables written by **`outputvalue`** or the **`gptoutputdouble`** or **`gptoutputdoublearray`** functions in a custom element.

Because a GDF-file is a hierarchical structure, the GDFA-Program is run on every group in the file. When data is retrieved, for example particle x-coordinates, only the current group is searched. However, additional variables and MR (multiple run) variables are searched down the hierarchy.

As an example, a routine is added calculating the average of x times z named **`avgxz`** in the file **`avgxz.c`**. The equation performed by the **`avgxz`** routine is:

$$\mathbf{avgxz} = \frac{\sum n_i x_i z_i}{\sum n_i}$$

where n_i is the number of elementary particles represented by macro particle *i*. The equations is a weighted average of x times z, reducing to $\Sigma xz / N$ when all n*i* are constant.

The details of the code are explained in the following sections, but when you are in a hurry, it is often sufficient to just modify the example code listed in Listing 3-i. The procedure to add a custom Linux program is explained in section 3.5 on page 72.

3.2 Overview of a GDFA program

Every GDFA program has a function name, appended by **_func**. When run, this function is called for every group in the GDF-file. The function is responsible for retrieving data from the GDF-file, calculating its specific statistics and returning the results.

Every GDFA program starts like the file listed below: A comment line between **/*** and ***/**, a **#include** line to include the standard GDFA routines, followed by the function itself. In this case, the name of the function is **example**.

```
1.  /* example.c: Just return 1 */
2.
3.  #include "gdfa.h"
4.
5.  int example_func( double *result )
6.  {
```

Obtaining particle coordinates is done using the **gdfmgetarr** function. For example, to retrieve the array of *x*-coordinates, the following lines can be used.

```
1.  int num ;
2.  double *x ;
3.  gdfmgetarr( "x", &x, &num )
```

After the **gdfmgetarr** function, the variable **num** is set to the number of particles **and x[0]**…**x[num-1]** contain the actual particle coordinates. When an attempt is made to retrieve a non-existing array, the **gdfmgetarr** function returns nonzero (TRUE in C). As typically a number of arrays is retrieved, which should all exists and have identical lengths, often the following scheme is applied:

```
1.  int num, tmpnum ;
2.  double *x, *y, *z ;
3.  if( gdfmgetarr( "x", &x, &num ) || num<1  ||
4.      gdfmgetarr( "y", &y, &tmpnum ) || tmpnum!=num ||
5.      gdfmgetarr( "z", &z, &tmpnum ) || tmpnum!=num ) return(1) ;
```

The **return(1)** statement indicates an error condition and instructs the GDFA kernel to skip to the following group.

Alternatively, a single variable can be read from the GDF-file using the **gdfmgetval** function. For example, when MR is used to scan the variable **current**, it can be obtained using the following lines

```
1.  double current ;
2.  if( gdfmgetval( "current", &current ) ) return(1) ;
```

When MR files are nested, this code will still work, independent on the ordering of the MR commands. This is because the **gdfmgetval** functions searched down the hierarchical GDF structure until the specified variable is found. If it is not found, an error condition is returned.

After the proper variables and arrays are obtained, the statistics can be calculated. Typically, such a calculation takes the following form when for example the weighted sum of *q* is required.

```
1.  double sum = 0.0 ;
2.  for(i=0 ; i<num ; i++) sum  = sum + nmacro[i]*q[i] ;
```

or

```
2.  for(i=0 ; i<num ; i++) sum += nmacro[i]*q[i] ;
```

The **nmacro** array contains the number of elementary particles represented by every macro-particle. The full C language and math library can be applied for the calculation of these statistics, but typically the functions and operators as listed in Table 1-A on page 5 are sufficient. The **+=** operator adds the right-hand-side to the left-hand-side as a convenient notation.

The calculation result must be stored in the ***result** variable. Technically, the variable **result** is a pointed to the actual variable, but interpreting ***result** as the variable to be set works fine for non C experts. Continuing the above example, the example function would end with:

```
1.  *result = sum ;
2.  return(0) ;
```

where the **return(0)** statement indicates that the calculation result is valid.

3.3 GPTwin wizard

To add a custom element using GPTwin, you must first select the "Progs" button on the Elements Toolbar (View/Elements), as shown in Figure 3-1. Then you can insert a new element from the menu (Elements/New) or right-click in the Elements Toolbar to display a pop-up menu.

Figure 3-1: GPT-Elements and GDFA-Programs toolbar.

To create a template for the **`avgxz`** program, the dialog as shown in Figure 3-2 must be filled in as indicated. Multiple arrays can be selected using ctrl-click, and please make sure the **`nmacro`** array is selected as well. This array contains the number of elementary particles every macro-particle represents and is used to weight the data.

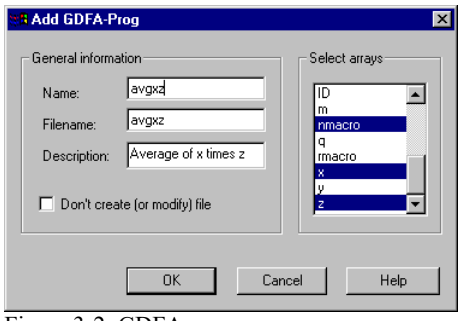


Figure 3-2: GDFA program avgx

Pressing the OK button generates the template file. It is a full working GDFA-program, but the proper equations still needs to be specified. After the custom GDFA Program is generated, it is listed in the Elements Toolbar. Double-clinking the element shows the source code of the program. Although this is a good time to make changes to the generated source-code, custom programs can always be modified. The element with the proper equations is listed in Listing 3-i on page 71.

Compiling the custom element can be done using the Menu (Elements/Compile GPT-Progs) or by right-clicking in the Elements-Toolbar. Both actions open and run a file named "**`build.bat`**". If one or more errors appear in the messages window, they should be corrected and "**`build.bat`**" must be run again using the Menu (File/Run) or the Run button on the Standard Toolbar. When no error messages appear, a new GDFA executable is compiled including the **`avgxz`** program: From this point forward, **`avgxz`** can be used on the GDFA command-line just like all built-in progs.

3.4 The final avgxz program

The GDFA program with the correct equations is shown in Listing 3-i. The equations are straightforward, but it is important to realize that the variable **sumn** is zero when spacecharge calculations are switched off in GPT. In that case, the simplified equation is used assuming all n_i's are equal.

Listing 3-i: The final **avgxz** program. In GPT version 2.50, the equations can also be written as:

```
*result = gdfamean2(namcro,x,z,num) ;
```

```
1.  /* avgxz.c: Average of x times z */
2.
3.  #include "gdfa.h"
4.
5.  int avgxz_func( double *result )
6.  {
7.    /* Declarations */
8.    int       i, num, tmpnum ;
9.    double    *nmacro, *x, *z ;
10.
11.   /* Private variables */
12.   double    sumn, sumnxz, sumxz ;
13.
14.   /* Get selected arrays from GDFA kernel */
15.   if( gdfmgetarr( "nmacro", &nmacro, &num ) || num<1  ||
16.       gdfmgetarr( "x", &x, &tmpnum ) || tmpnum!=num ||
17.       gdfmgetarr( "z", &z, &tmpnum ) || tmpnum!=num ) return(1) ;
18.
19.
20.   /* This calculates the weighted average of x times z */
21.   sumn = 0.0 ; sumnxz = 0.0 ; sumxz = 0.0 ;
22.   for(i=0 ; i<num ; i++)
23.   {
24.     sumn   += nmacro[i] ;
25.     sumnxz += nmacro[i]*x[i]*z[i] ;
26.     sumxz  += x[i]*z[i] ;
27.   }
28.
29.   if( sumn!=0.0 )
30.     *result = sumnxz/sumn ;
31.   else
32.     *result = sumxz/num ;
33.
34.   /* Return without error */
35.   return( 0 ) ;
36. }
```

3.5 Linux procedure for adding a GDFA program

To add a custom analysis program to the Linux version of GDFA, the following steps have to be taken:

- Type **`cd ~/gpt/progs`** (where **`~/gpt`** is the GPT installation directory).
- Write the actual code. A good starting point is a copy of one of the built-in programs provided in the **`sources`** directory.
- Add your program to the file **`proglist`**. This is an ascii file with two words on every line. The first word specifies the name of your program; this is the name of the new GDFA function (without **`_func`**). This is also the name that needs to be specified on the GDFA command-line. The second word in **`proglist`** provides the filename without extension containing the C code containing the program. For an element named **`avgxyz`** with the source code in file **avgxyz.c** you should add the line "**`avgxyz avgxyz`**" to **`proglist`**.
- Type **`make`** twice. This compiles your new function and it creates the code needed by GDFA to efficiently interface with the new function.
- When an existing function is modified, **`make`** needs to be run only once to create a new GDFA executable.

GDFA is now ready to run with your new program. Whenever you modify your custom data analysis routine, only the last **`make`** needs to be repeated.

To remove an element, delete it from **`proglist`** and run **`make`** twice.

3.6 Retrieving data

3.6.1 gdfmgetarr

`int gdfmgetarr(const char *name, double **result, int *len)`
Retrieves an array of floating point data in the current group.

`name`	Name of the array to obtain. When it is not found, all parent groups are searched also.
`result`	Pointer to requested array. Because this represents the actual data, not a copy, you should not modify the results.
`len`	Length of requested array.
Return value	Nonzero on error or when the array is not found.

Example:

```
1. double sum, *x ;
2. int i, len ;
3. if( gdfmgetarr("x",&x,&len) !=0 ) /* Error condition */
4.
5. sum=0.0 ;
6. for(i=0 ; i<len ; i++) sum += x[i] ;
```

For non C programmers, the **`&`** sign is a bit difficult to explain. However, the example code allows **`gdfmgetarr`** to fill-in both the **`x`** and **`len`** variables. In a way, you could interpret **`x`** and **`len`** as additional return values.

3.6.2 gdfmgetval

`int gdfmgetval(const char *name, double *result)`
Retrieves a floating-point value in the current group.

`name`	Name of the value to obtain. When it is not found, all parent groups are searched also.
`result`	Pointer to requested value. Because this represents the actual data, not a copy, you should not modify the result.
Return value	Nonzero on error or when the value is not found.

3.7 Helper routines

There are a number of built-in routines available in GDFA to simplify common data analysis. They are listed below.

3.7.1 gdfamean

`double gdfamean(double *n, double *a, int num) ;`
Calculates the weighted average of **`a`**.

`n`	**`nmacro`** array. Can be all zero.
`a`	Input array
`num`	Length of **`n`** and **`a`**
Return value:	Weigthed average of **`a`**.

The return value is: $\frac{\Sigma \mathbf{n}_i \mathbf{a}_i}{\Sigma \mathbf{n}_i}$

This method is sensitive to outliers. For a robust measure, see **`gdfamedian`**.

Example: Weighted average of x.

```
1.  int avgx_func( double *result )
2.  {
3.    /* Declarations */
4.    int num, tmpnum ;
5.    double *x, *nmacro ;
6.
7.    /* Get selected arrays from GDFA kernel */
8.    if( gdfmgetarr( "x", &x, &num ) || num<1  ||
9.        gdfmgetarr( "nmacro", &nmacro, &tmpnum ) || tmpnum!=num ) return(1) ;
10.
11.   /* Store result and return without error code */
12.   *result = gdfamean(nmacro,x,num) ;
13.   return( 0 ) ;
14. }
```

3.7.2 gdfamean2

`double gdfamean2(double *n, double *a, double *b, int num) ;`
Calculates the weighted average of **`a`** times **`b`**.

`n`	**`nmacro`** array. Can be all zero.
`a`	First input array.
`b`	Second input array.
`num`	Length of **`n`**, **`a`** and **`b`**
Return value:	Weighted average of **`a`**.

The return value is: $\frac{\Sigma \mathbf{n}_i \mathbf{a}_i \mathbf{b}_i}{\Sigma \mathbf{n}_i}$

This method is sensitive to outliers.

3.7.3 gdfamedian

`int gdfamedian(double *n, double *x, int num, double *result);`
Calculates the weighted median value

`n`	Pointer to weight array
`x`	Pointer to data array
`num`	Length of weight and data array
`result`	Pointer to median of x
Return value	Nonzero on error (such as num=0)

This method is robust and insensitive to outliers.

3.7.4 gdfasubavg

```
void gdfasubavg(double *n, double *dest, double *src, int num) ;
```
Fills **dest** with **src** while subtracting the weighted average.

n	**nmacro** array. Can be all zero.
dest	Output array. Must be pre-allocated by the user.
src	Input array.
num	Length of **n** and **a**

This function centers the **src** array around zero. A subsequent call to **gdfamean(n,dest,num)** will always result in 0.

3.7.5 gdfacmpdouble

```
int gdfacmpdouble(const void *val1, const void *val2) ;
```
Compare two doubles in ascending order for use in the **qsort** C-library routine.

val1	Pointer to double, declared as void for ANSI compliance.
val2	Pointer to double, declared as void for ANSI compliance.
Return value:	Ascending order.

When an array of floating point values need to be sorted, the C library function **qsort** can be used. For convenience, this comparison function can be specified when the array is to be sorted in ascending order. The function is obsolete given the std::sort algorithm in C++, but remains for compatibility reasons.

Exaple: Sort array **a** in ascending order.
```
double a[num] ;
qsort(a,num,sizeof(*a),gdfacmpdouble);
```

3.7.6 gdfavariation

```
int gdfavariation(double *n, double *x, int num, double fraction, double
*result);
```
Calculates the smallest variation (max-min) in x containing the specified fraction

n	Pointer to weight array
x	Pointer to data array
num	Length of weight and data arrays
fraction	Fraction <0,1>
result	Pointer to variation in x containing the specified fraction of the particles
Return value	Nonzero on error

This method is robust and insensitive to outliers.

3.7.7 getepsfraction

```
double getepsfraction(int Ndim, double ***vecs, int num, double *nmacro,
  double *nq, double *eps, double fraction);
```
Estimates the smallest hypervolume with elliptical shape containing the specified fraction

Ndim	Number of dimensions
vecs	Array[Ndim] of pointers to pointers to data such that (*(vecs[i])[j] points to the date for dimension i and point index j.
num	Length of data arrays
nmacro	Pointer to weight array
nq	Output array of per-particle nmacro for particles inside hyperellipsoidal volume, 0 for all other paricles. Memory must be provided by calling routine.
eps	Outout array of per-particle 'volume'. Memory must be provided by calling routine.
fraction	Fraction <0,1>
Return value:	Volume of the hyperellipsoid containing specified **fraction** of the particles

This routine estimates the smallest ellipsoidal (hyper)volume containing the specified **fraction** of the particles. It is the main building block for all robust data analysis in N-dimensional phase-space. The algorithm starts by running principal component analysis on the covariance matrix of the given data. This gives the orientation of an hyperellipsoid. The ellipsoid is than scaled, but not deformed, such that a small fraction of the particles fall outside its bounds. These particles are not taken into account any further, and the process is iteratively repeated until the desired fraction is reached. The particles that remain, via the **nq** and **eps** arrays, and the volume of the final (hyper)ellipsoid are returned.

The following example calculate the smallest ellipse containing 50% of the particles in xy-space.

```
1.  /* xy_50.c: Calculate smallest ellipse in xy containing 50% of the particles */
2.
3.  #include <stdlib.h>
4.  #include "gdfa.h"
5.  #include "robust.h"
6.
7.  #define FRACTION 0.5
8.
9.  int xy_50_func( double *result )
10. {
11.   /* Declarations */
12.   int num, tmpnum ;
13.   double *nmacro, *x, *y ;
14.   double **(vecs[2]) = {&x,&y} ;
15.
16.   /* Get selected arrays from GDFA kernel */
17.   if( gdfmgetarr( "nmacro", &nmacro, &num ) || num<3  ||
18.       gdfmgetarr( "x", &x, &tmpnum ) || tmpnum!=num ||
19.       gdfmgetarr( "y", &y, &tmpnum ) || tmpnum!=num ) return(1) ;
20.
21.   /* Allocate memory for extra arrays */
22.   double *nq =(double *)malloc( num*sizeof(double)) ;
23.   double *eps=(double *)malloc( num*sizeof(double)) ;
24.
25.   /* Calculate */
26.   *result = getepsfraction(2,vecs,num,nmacro,nq,eps,FRACTION) ;
27.
28.   /* Cleanup and return */
29.   free(eps) ;
30.   free(nq) ;
31.   return( 0 ) ;
32. }
```

3.7.8 stdsqrt

double stdsqrt(double x) ;
Calculate square-root, return 0 on negative argument

x	Argument
Return value:	**sqrt(x)**, or 0 when **x**<0.

Use this function when a negative argument in a square root doesn't signal an error condition but indicates that the result must be zero. For example, calculating the normalized energy deviation with the standard equation:

$$\mathrm{std}(\gamma) = \sqrt{<\gamma^2> - <\gamma>^2}$$

works perfectly well, unless all γ's are equal. In that case, there is a 50% chance that due to the finite numerical precision, the argument of the square root is negative. And that will trigger a spurious floating point exception and terminate program execution. Using **stdsqrt** solves the problem because if the argument is negative the result should be zero.

Example: Standard normalized energy deviation.

```
1.  int nstdG_func( double *result )
2.  {
3.    /* Declarations */
4.    int num, tmpnum ;
5.    double *G, *nmacro ;
6.
7.    /* Get selected arrays from GDFA kernel */
8.    if( gdfmgetarr( "G", &G, &num ) || num<1  ||
9.        gdfmgetarr( "nmacro", &nmacro, &tmpnum ) || tmpnum!=num ) return(1) ;
10.
11.   /* Store result and return without error code */
12.   *result = stdsqrt( gdfamean2(nmacro,G,G,num)-
13.                      gdfamean(nmacro,G,num)*gdfamean(nmacro,G,num) ) ;
14.   return( 0 ) ;
15. }
```

References

1 Brian W. Kernighan, Dennis M. Ritchie, *The C programming language*, Second edition, AT&T Bell Laboratories, Murray Hill, New Jersey (1988).

2 William H. Press, et all. *Numerical Recipes*, Second edition, Cambridge University Press, Cambridge, UK (1992).

Index