Tracking in magnetic fields

Advanced FLUKA Course

Magnetic field tracking in FLUKA

FLUKA allows for tracking in arbitrarily complex magnetic fields. Magnetic field tracking is performed by iterations until a given accuracy when crossing a boundary is achieved.

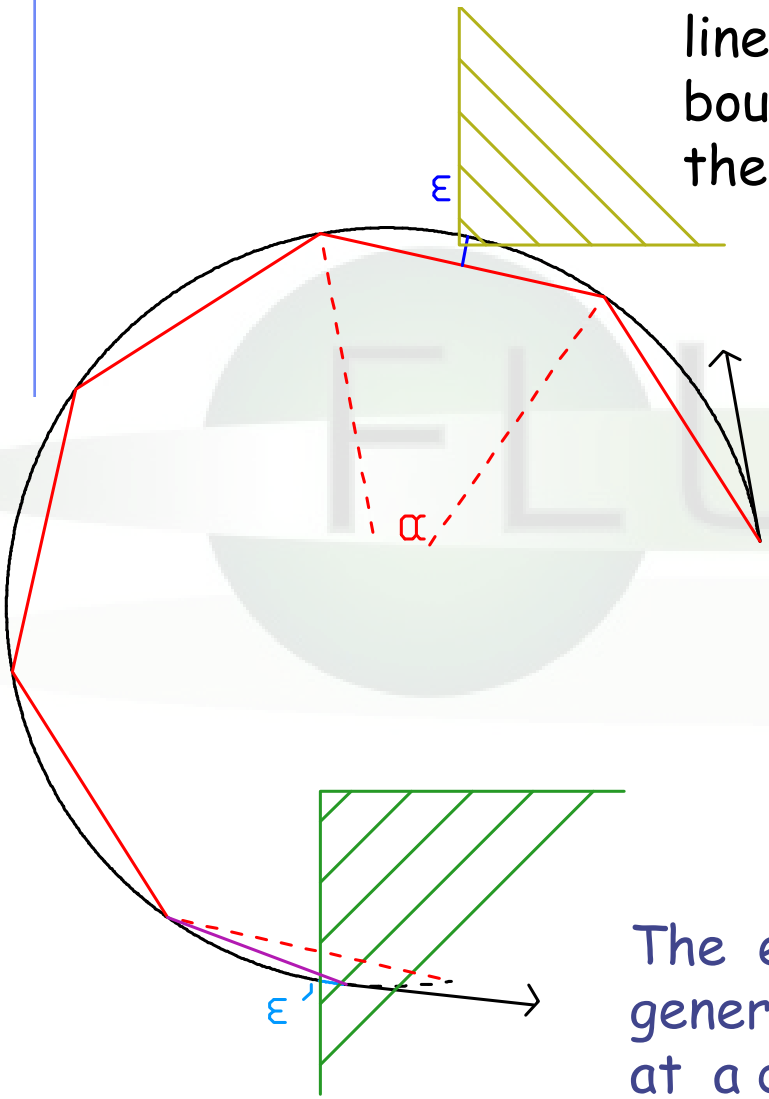
Meaningful user input is required when setting up the parameters defining the tracking accuracy.

Furthermore, when tracking in magnetic fields FLUKA accounts for:

- The precession of the mcs final direction around the particle direction: this is critical in order to preserve the various correlations embedded in the FLUKA advanced MCS algorithm
- The precession of a (possible) particle polarization around its direction of motion: this matters only when polarization of charged particles is a issue (mostly for muons in Fluka)
- The decrease of the particle momentum due to energy losses along a given step and hence the corresponding decrease of its curvature radius. Since FLUKA allows for fairly large (up to 20%) fractional energy losses per step, this correction is important in order to prevent excessive tracking inaccuracies to build up, or force to use very small steps

Magnetic field tracking in FLUKA

The true step (black) is approximated by linear sub-steps. Sub-step length and boundary crossing iteration are governed by the required tracking precision



The red line is the path actually followed,
the magenta segment is the last substep, shortened because of a boundary crossing

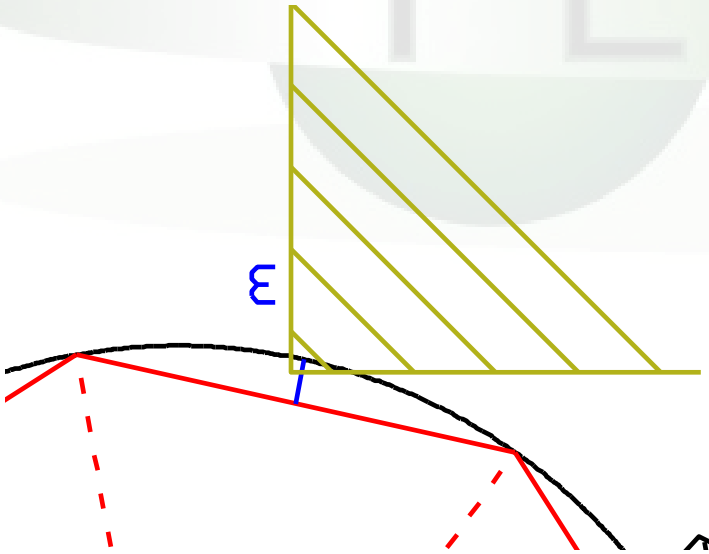
- α= max. tracking angle (MGNFIELD)
- ε = max. tracking/missing error (MGNFIELD or STEPSIZE)
- ε' = max. bdrx error (MGNFIELD or STEPSIZE)

The end point is ALWAYS on the true path, generally NOT exactly on the boundary, but at a distance < ε' from the true boundary crossing (light blue arc)

Setting the tracking precision

MGNFIELD	α	ε	Smin	B_x	B_y	B_z

- α largest angle in degrees that a charged particle is allowed to travel in a single sub-step. Default = 57.0 (but a maximum of 30.0 is recommended!)
- ε upper limit to error of the boundary iteration in cm (ε' in fig.). It also sets the tracking error ε. Default = 0.05 cm

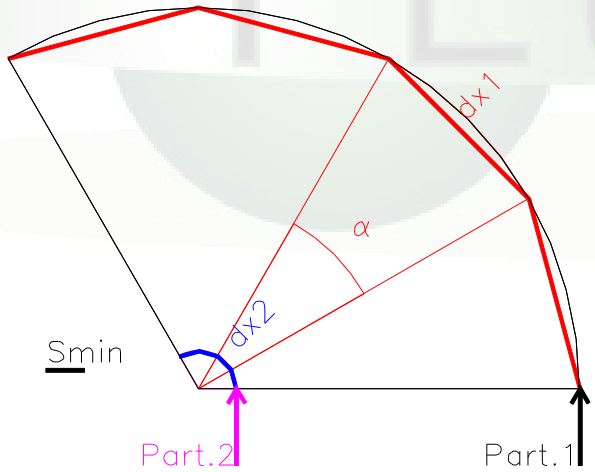


IF α and /or ε are too large, boundaries may be missed (as in the plot).
IF they are too small, CPU time explodes..
Both α and ε conditions are fulfilled during tracking
→ Set them according to your problem
→ Tune ε by region with the STEPSIZE card
→ Be careful when very small regions exists in your setting : ε must be smaller than the region dimensions!

Setting the tracking precision

MGNFIELD	α	ε	Smin	B_x	B_y	B_z

- Smin minimum sub-step length. If the radius of curvature is so small that the maximum sub-step compatible with α is smaller than Smin, then the condition on α is overridden. It avoids endless tracking of spiraling low energy particles. Default = 0.1 cm



Particle 1: the sub-step corresponding to α is > Smin -> accept
Particle 2: the sub-step corresponding to α is < Smin -> increase α

Smin can be set by region with the STEPSIZE card

Setting precision by region

```
STEPSIZE  Smin/ε  Smax    Reg1    Reg2    Step
```

- Smin: (if what(1)>0) minimum step size in cm Overrides MGNFIELD if larger than its setting.
- ε (if what(1)<0) : max error on the location of intersection with boundary.
  - The possibility to have different "precision" in different regions allows to save cpu time
- Smax : max step size in cm. Default:100000. cm for a region without mag field, 10 cm with mag field.
  - Smax can be useful for instance for large vacuum regions with relatively low magnetic field
  - It should not be used for general step control, use EMFFIX, FLUKAFIX if needed

Possible loops in mag.fields

- Although rare, it is ***PHYSICALLY*** possible that a particle ***loops for ever*** (or for a very long time). Imagine a stable particle generated perpendicularly to a uniform B in a large enough vacuum region: it will stay on a circular orbit forever !
- Suppose now that the orbit enters in a non-vacuum region (here we can at least loose energy..) but the boundary is missed due to insufficient precision. This results again in a never-ending loop.

Luckily, it almost never happens. ***Almost.***

The magfld.f user routine

This routine allows to define arbitrarily complex magnetic fields:
(uniform fields can be defined trough the MGNFIELD card)

SUBROUTINE MAGFLD (X, Y, Z, BTX, BTY, BTZ, B, NREG, IDISC)

.

Input variables:

x,y,z = current position

nreg = current region

Output variables:

btx,bty,btz = cosines of the magn. field vector

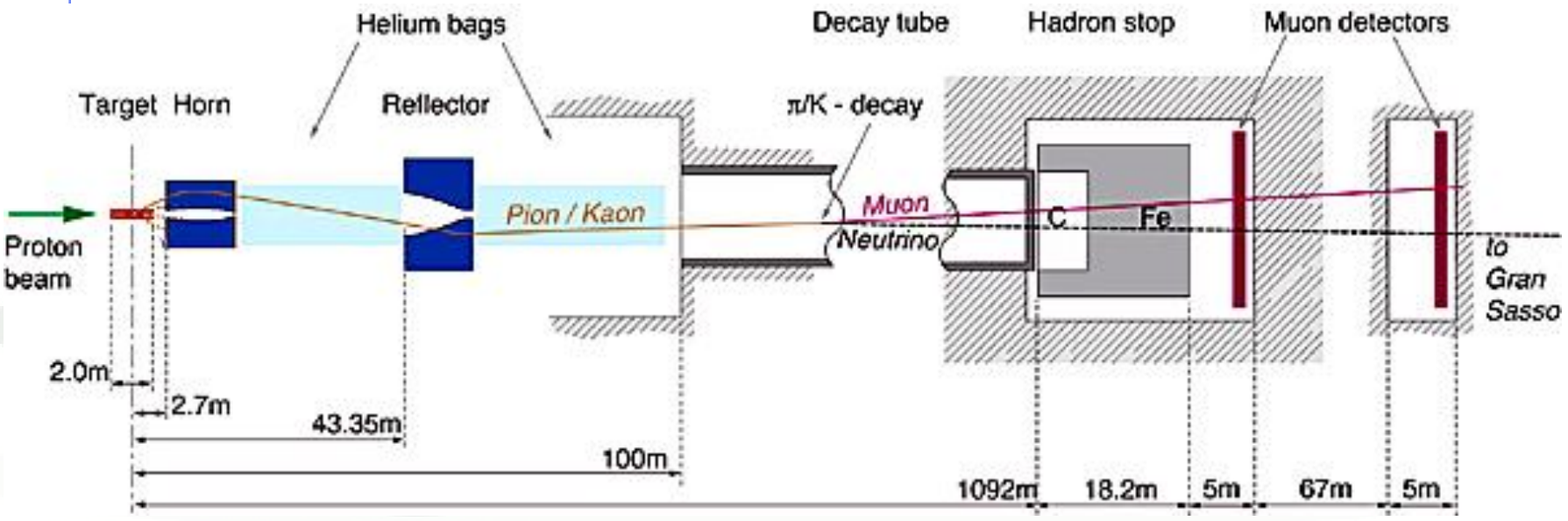
B = magnetic field intensity (Tesla)

idisc = set to 1 if the particle has to be discarded

- All floating point variables are double precision ones!
- BTX, BTY, BTZ must be normalized to 1 in double precision
- Magfld .f is called only for regions where a magnetic field has been declared through ASSIGNMAT

Example: magnetic field in CNGS

Cern Neutrino to Gran Sasso

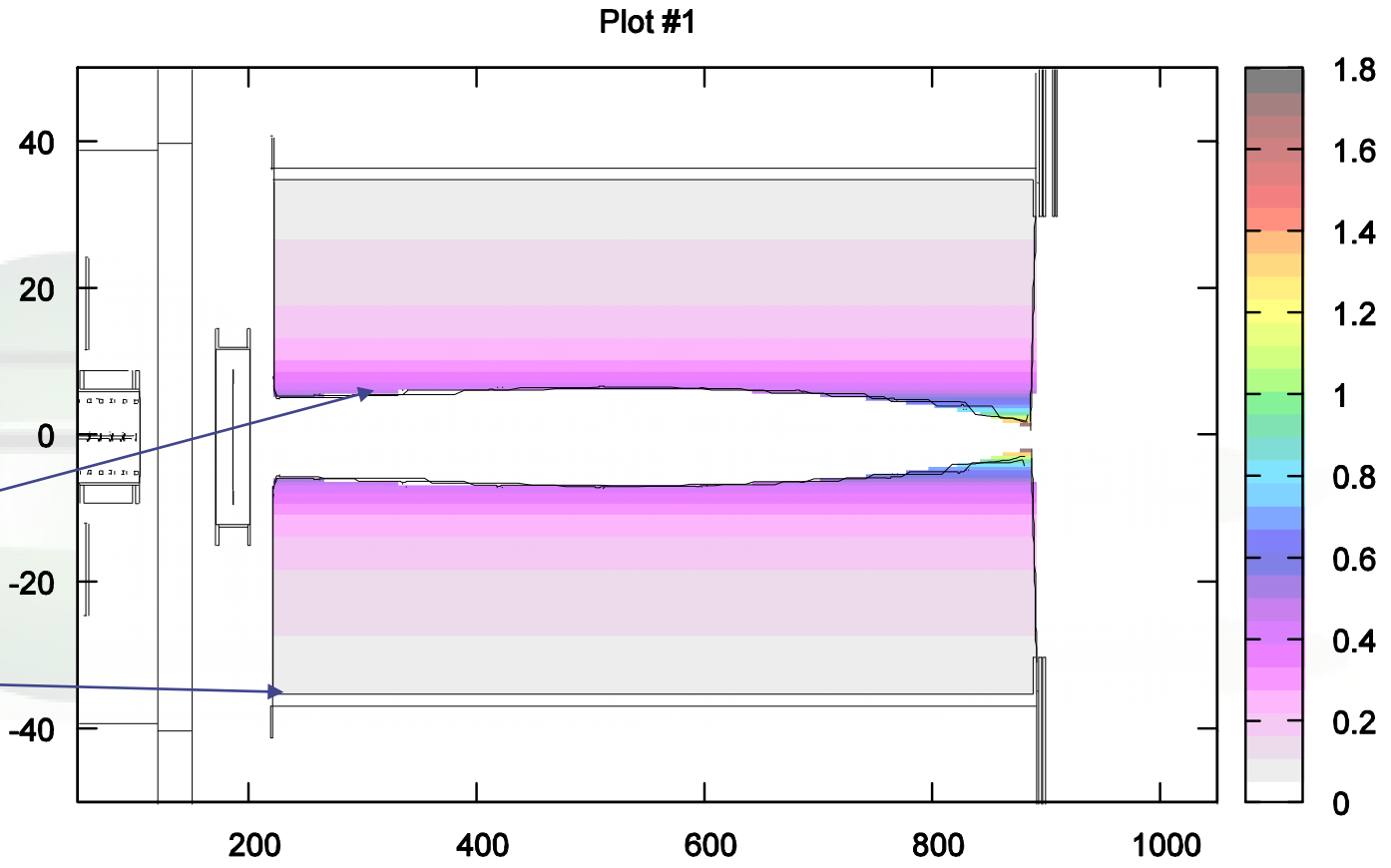


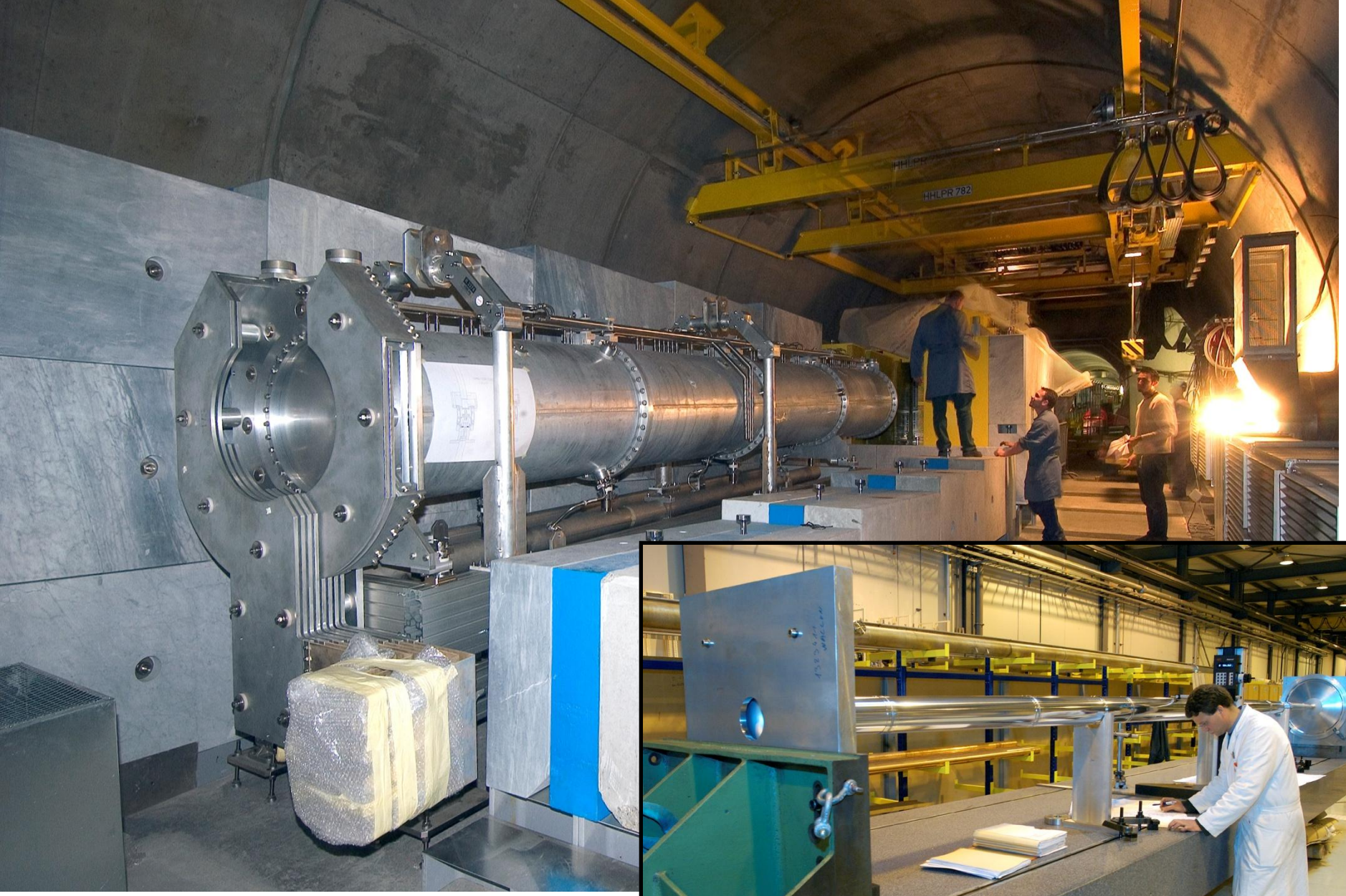
The two magnetic lenses (blue in the sketch) align positive mesons towards the Decay tunnel, so that neutrinos from the decay are directed to Gran Sasso, 730~km away
Negative mesons are deflected away
The lenses have a finite energy/angle acceptance

Example : the magfld.f routine

Magnetic field intensity in the CNGS horn

A cuurent ≈150kA, pulsed, circulates through the Inner and Outer conductors The field is toroidal, B÷1/R




HHLPR 782

magfld: example

```
SUBROUTINE MAGFLD ( X, Y, Z, BTX, BTY, BTZ, B, NREG, IDISC )

      INCLUDE '(DBLPRC)'
      INCLUDE '(DIMPAR)'
      INCLUDE '(IOUNIT)'

      INCLUDE '(NUBEAM)'

      IF ( NREG .EQ. NRHORN ) THEN
            RRR = SQRT ( X**2 + Y**2 )
            BTX =-Y / RRR
            BTY = X / RRR
            BTZ = ZERZER
            B   = 2.D-07 * CURHOR / 1.D-02 / RRR
      END IF
```

Standard FLUKA includes : KEEP THEM

This gives a versor ⊥ radius in a plane ⊥ z axis

In this case, the cosines are automatically normalized. Otherwise, user MUST ensure that BTX**2+BTY**2+BTZ**1=ONEONE

B intensity depending on R and current

USEFUL TIP

This is a user defined include file, containing for example COMMON /NUBEAM/ CURHORN, NRHORN, ……

It can be initialized in a custom usrini.f user routine, so that parameters can be easily changed in the input file

magfld: example contnd

Different fields in different regions:

```
IF ( NREG .EQ. NRHORN ) THEN
                ......
ELSE IF ( NREG .EQ. NRSOLE ) THEN
        BTX = ZERZER
        BTY = ZERZER
        BTZ = ONEONE
        B   = SOLEB
ELSE IF ( NREG .EQ. NRMAP ) THEN
        CALL GETMAP ( X, Y, Z, BTX, BTY, BTZ, B)
ELSE
        WRITE ( LUNOUT, *) 'MGFLD, WHY HERE ?
        WRITE ( LUNOUT, *) NREG'
         STOP
END IF
```

This gives a perfect solenoid field

Intensity calculated at initialization

Get values from field map

Add a bit of protection.

The user can add more routines, they have to be included in the linking procedure

Always :

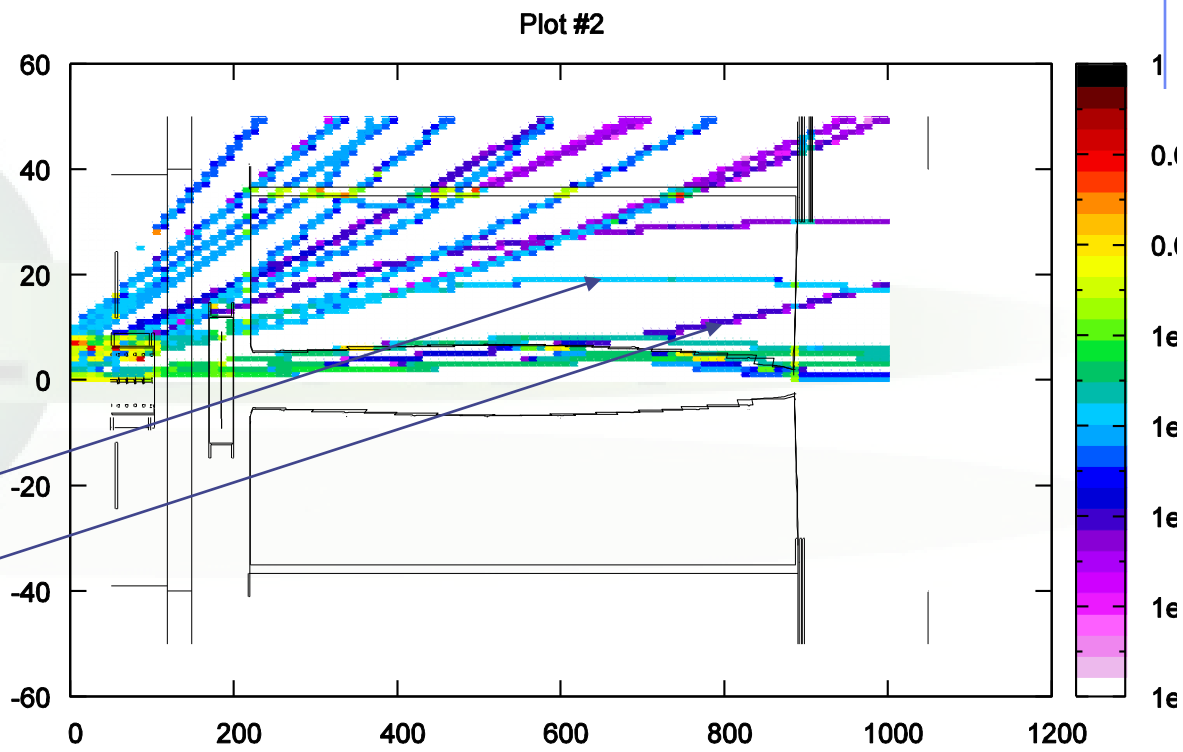
include the three standard FLUKA INCLUDEs

use FLUKA defined constants and particle properties for consistency

Possible, not explained here : call C routines

magfld: results

charged particle tracks
in the CNGS geometry
1 event
USRBIN R-Z



Focused
De-focused
Escaping..many

The user initialization routines

- usrglo.f called before all initialization, if a USRGCALL card is issued
- usrini.f called after all initialization, if a USRICALL card is issued
- usrein.f called at each event, before the showering of an event is started, but after the source particles of that event have been already loaded on the stack. No card is needed

Very useful to initialize and propagate variables common to other user routines

- Associated OUTPUT routines:
- usrout.f called at the end of the run if USROCALL is present
- usreout.f called at the end of each event , no card needed

Initialization routines -II

- usrglo.f knows nothing about the simulations, but can provide informations to the other initialization stages.
- usrini.f knows everything about the problem. Here one can, for instance, use informations about materials, regions etc.
- usrein.f is useful when doing event-by-event user scoring , it can for instance reset and reinitialize event-dependent user quantities

The USRGCALL and USRICALL cards can be issued many times if more parameters are needed

The USRICALL card accepts input BY NAMES

usrini.f :example

```
SUBROUTINE USRINI ( WHAT, SDUM )

INCLUDE '(DBLPRC)'
INCLUDE '(DIMPAR)'
INCLUDE '(IOUNIT)'
      .....
DIMENSION WHAT (6)
 CHARACTER SDUM*8
      ....
 CHARACTER  MAPFILE(8)
 INCLUDE '(NUBEAM)'

IF ( SDUM .EQ. 'HORNREFL' ) THEN
    NRHORN = WHAT (1)
    CURHORN = WHAT (2)
ELSE IF (SDUM .EQ. 'SOLENOID') THEN
     SOLEB = WHAT (2)
      NRSOLE = WHAT(1)

    contnd
```

Default declarations

Here we store our variables

Here we initialize region numbers And parameters for the magfld.f routine

usrini.f: example contnd

```
ELSE
     MAPFILE=SDUM
     MYUNIT=21
     CALL OAUXFI ( MAPFILE,  MYUNIT, 'OLD' , IERR)
     CALL READMAP(MYUNIT)
     CLOSE (21)
      NRMAP= WHAT (1)
END IF
RETURN
```

Use the SDUM field to read the name of the magnetic field map file

Open the field map

Call a user procedure that reads and stores the field map to be used by magfld.f

This usrini needs 3 cards to initialize all parameters:, like i.e.

```
USRICALL   MyHorn    150000.                    HORNREFL
USRICALL   MySole     1. 3                      SOLENOID
USRICALL   Mapped                                myflmap
```

The region names in the what's are automatically parsed and converted to region numbers by FLUKA
(same would happen with materials, scoring ..)

Roto-translation routines:

SUBROUTINE DOTRSF (NPOINT, XPOINT, YPOINT, ZPOINT, KROTAT)

…

SUBROUTINE DORTNO (NPOINT, XPOINT, YPOINT, ZPOINT, KROTAT)

…

SUBROUTINE UNDOTR (NPOINT, XPOINT, YPOINT, ZPOINT, KROTAT)

…

SUBROUTINE UNDRTO (NPOINT, XPOINT, YPOINT, ZPOINT, KROTAT)

…

DIMENSION XPOINT (NPOINT), YPOINT (NPOINT), ZPOINT (NPOINT)

…

The **DOTRSF** routine executes the KROTAT_th transformation as defined by ROT-DEFI on NPOINT points, defined by the X,Y,ZPOINT arrays, ***with a (possible) translation*** included

DORTNO does the same ***without the translation*** (eg for velocity vectors)

UNDOTR performs the ***inverse*** transformation, ***with a (possible) translation*** included

UNDRTO performs the ***inverse*** transformation, ***without the translation***

Numerical precision

- Floating point representation

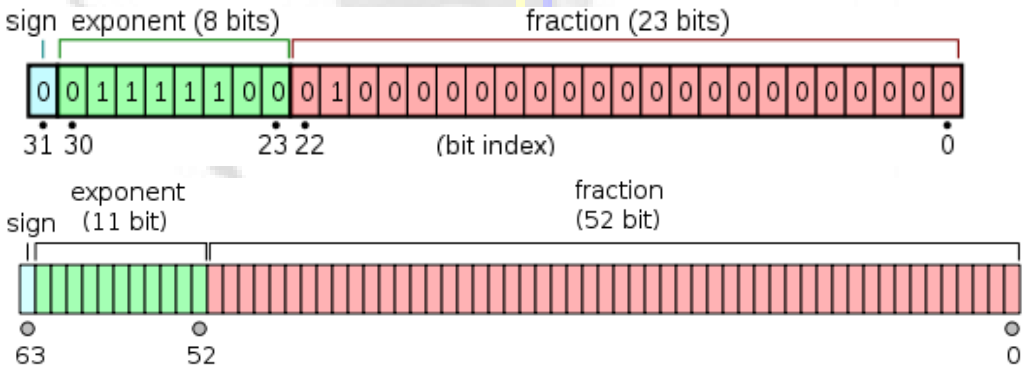
$$\pm d_0 d_1 d_2 \ldots d_{p-1} \times \beta^e$$

  where: β=base, $0.dddd$=significant

- Represents the number

$$\pm (d_0 + d_1 \beta^{-1} + \ldots + d_{p-1} \beta^{-(p-1)}) \beta^e, \qquad (0 \le d_i < \beta)$$

- Bits required: $\log_2(e_{max} - e_{min} + 1) + \log_2(\beta^p) + 1$
- Real numbers might not be exactly represented as a floating-point number. Example:
  with β=2 the number 0.1 has an infinite representation and with p=24 will be represented as: 0.10000000149011611938476562 5
- IEEE representation:
  - Single precision (32bit):
  - Double precision (64bit):



Floating point: Accuracy

- Cancellation: subtraction of nearly equal operands may cause extreme loss of accuracy.
- Conversions to integer are not intuitive:
  converting (63.0/9.0) to integer yields 7,
  but converting (0.63/0.09) may yield 6.
  This is because conversions generally truncate rather than round.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a denormal value or zero. If a denormal number results, precision will be lost.
- Testing for safe division is problematic: Checking that the divisor is not zero does not guarantee that a division will not overflow and yield infinity.
- Equality test is problematic: Two computational sequences that are mathematically equal may well produce different floating-point values. Programmers often perform comparisons within some tolerance

Minimizing Accuracy Problems

- Use double precision whenever possible.
- Small errors in floating-point arithmetic can grow when mathematical algorithms perform operations an enormous number of times. e.g. matrix inversion, eigenvalues…
- Expectations from mathematics may not be realized in the field of floating-point computation. e.g. $\sin^2\theta+\cos^2\theta = 1$.
- Always replace the $x^2-y^2 = (x+y)(x-y)$
- Equality test should be avoided: replace with "fuzzy" comparisons (if (abs(x-y) < epsilon) ...)
- Adding a large number of numbers can lead to loss of significance, use Kahan algorithm instead
- For the quadratic formula use either

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

when $b^2>>4ac$, then $\sqrt{(b^2-4ac)}\approx|b|$ therefore will introduce cancelation

END

usrglo.f :example

```
SUBROUTINE USRGLO ( WHAT, SDUM )

INCLUDE '(DBLPRC)'
INCLUDE '(DIMPAR)'
INCLUDE '(IOUNIT)'
     .....
DIMENSION WHAT (6)
CHARACTER SDUM*8
INCLUDE '(NUBEAM)'

 IF ( WHAT(1) .GT. ZERZER )  THEN
    ROTTRG   =  WHAT(1)
    LTGMISA = .TRUE.
    TRATARG = ZERZER
    IF ( WHAT(2) .GT. ZERZER ) TRATARG = WHAT(2)
 RETURN
```

Default declarations

Here we store our variables

Suppose we have a lattic.f routine
That rotates the target to simulate misalignment : here a flag and the rotation / translation amounts are set