The FLUKA environment

for user routine programming

Overview

- Why user routines
- How to link them into a custom executable
- Accessible information (in COMMON)
- Available tools (core routines)
- Debugging practice

Why user routines

- FLUKA offers a rich choice of **built-in options (cards)** for *scoring* most useful quantities and for applying *variance reduction* techniques, without requiring the users to write a single line of code. These should be fully explored and exploited **first**.

- Nevertheless, one may still need
  - to simulate a specific scenario, which cannot be set up by cards only (typically → *source* routine)
  - to extract information that is not directly obtainable

- Pre-defined user routines offer deeper flexibility, at the cost of (Fortran) programming effort and post-processing custom analysis.

Where to look

```
/usr/local/fluka/
├── include
└── src
    ├── tools
    └── user
├── …
```

- *User routine* templates sit in `src/user`
- The user should copy the needed ones into own working directory and customize them.

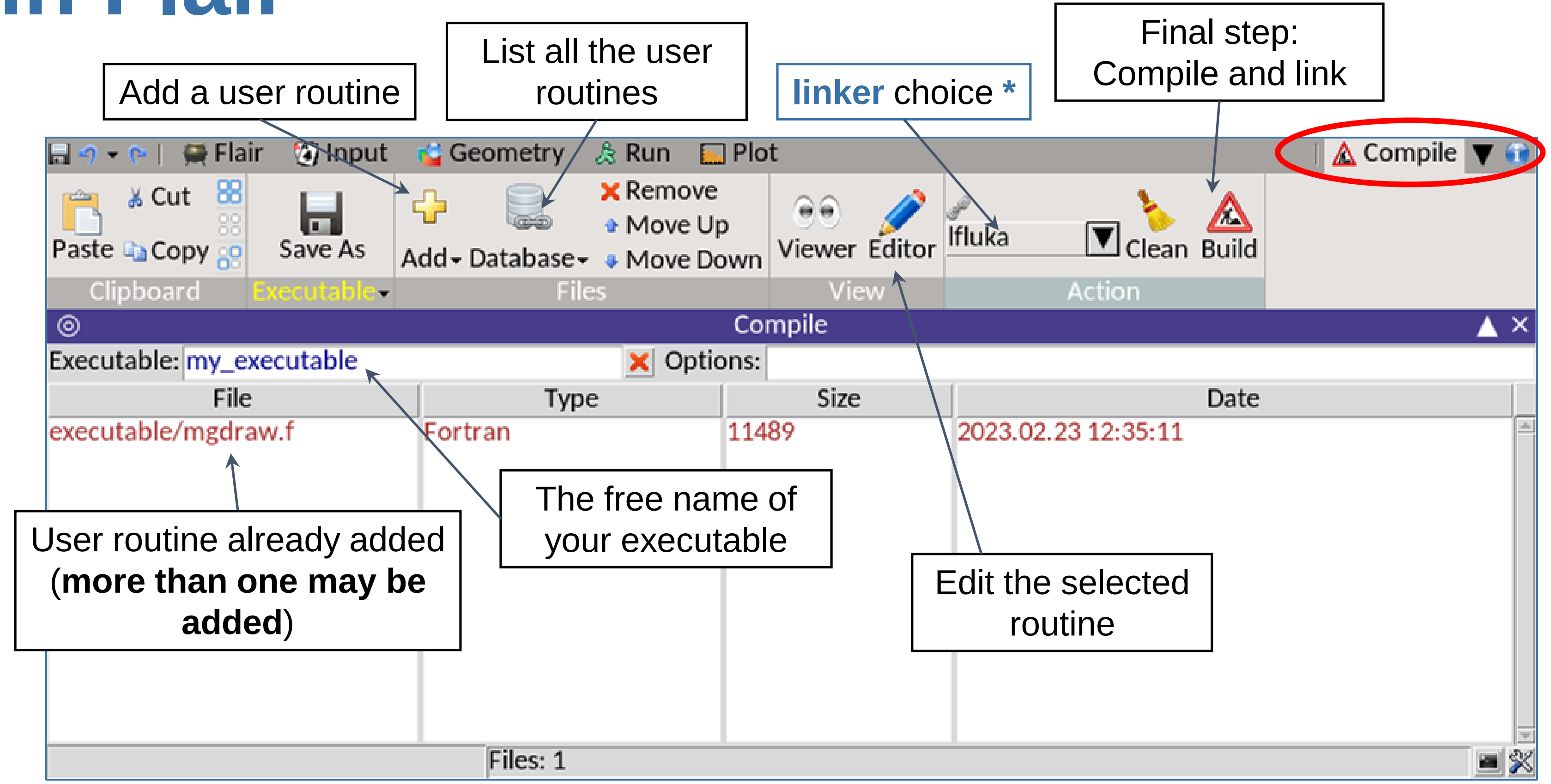
- Plenty of *variables* containing relevant information are accessible by the inclusion of the files in the `include` directory (e.g., `INCLUDE 'trackr.inc'`), which make them available through COMMON blocks.

The Compile tab in Flair

Most user routines need to be activated by input cards.

The Database button includes the Scan Input capability that automatically highlights the user routines implied by your input file.

A single executable shall embed all user routines of your choice.



Compile

Executable: my_executable Options:

File	Type	Size	Date
executable/mgdraw.f	Fortran	11489	2023.02.23 12:35:11

Files: 1

** Note the linker alternative **ldpmqmd**, which is required to include the DPMJET and RQMD libraries (for ions above 125 MeV/n and hadrons above 20 TeV).*

User routine scope [I]: inside the simulation loop

in `src/user`

SOURCE GENERATION
source_newgen.f
source.f
(soevsv.f) (lppsok.f)

accessing (almost) everything
mgdraw_empty.f
mgdraw.f

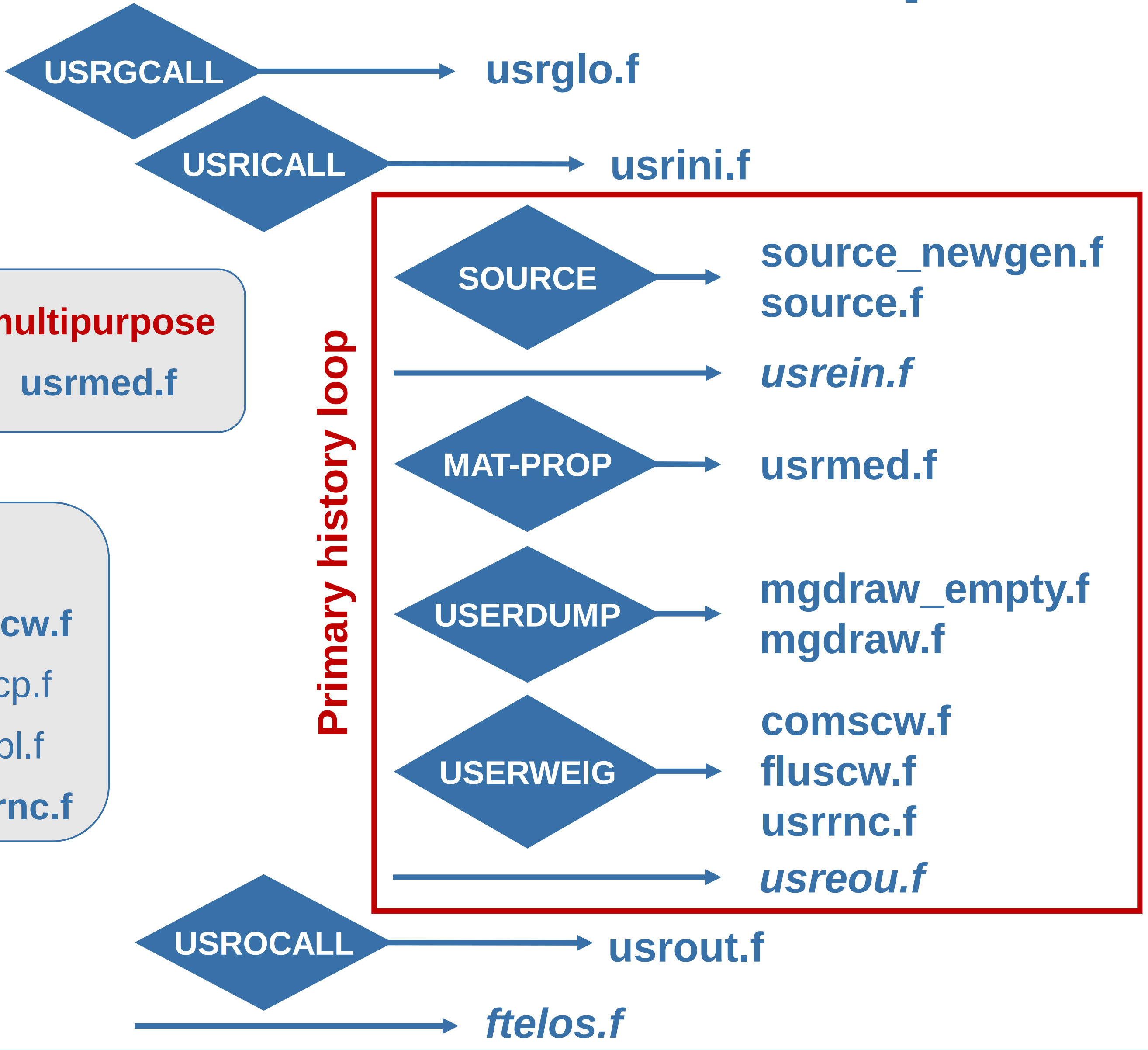
multipurpose
usrmed.f

INITIALISATION
usrglo.f
usrini.f
usrein.f

OUTPUT
usreou.f
usrout.f
ftelos.f

SCORING

comscw.f	**fluscw.f**
endscp.f	fldscp.f
musrbr.f	lusrbl.f
fusrbv.f	**usrrnc.f**



usrein.f, *usreou.f*, *ftelos.f* are examples of user routines always invoked with no need for an activating card (but by default they do nothing)

User routine scope [II]

in `src/user`

BIASING

ubsset.f

usimbs.f

udcdrl.f

MAGNETIC / ELECTRIC FIELDS

magfld.f

elefld.f

PHYSICS

formfu.f

MATERIALS

usrhsc.f

OPTICAL PHOTONS

abscff.f	dffcff.f
frghns.f	ophbdx.f
pshckp.f	queffc.f
rflctv.f	rfrndx.f
ustckv.f	wvlnsh.f

Accessing particle stack

mdstck.f

stupre.f

stuprf.f

LATTICE GEOMETRY

lattic.f

The shared variable scope

`in include`

just a small selected subset

beamcm.inc	beam particle properties (from BEAM and BEAMPOS)
sourcm.inc	user variables from a user-written source
souevt.inc	recording of the current source particle(s)
caslim.inc	number of primary particles followed
flkstk.inc	main particle stack of FLUKA
emfstk.inc	particle stack for electrons, positrons and photons interactions
genstk.inc	properties of secondaries created in a hadronic event
fheavy.inc	special stack for nuclear fragment products
resnuc.inc	residual nucleus properties
flkmat.inc	material properties
ltclcm.inc	lattice cell identification
trackr.inc	properties of the particle currently transported
paprop.inc	intrinsic particle properties (mass, charge, mean life, …)
scohlp.inc	scoring card identification

dblprc.inc sets the encyclopaedic scene

DouBLe PReCision common block included in all FLUKA routines, containing the declaration

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

and setting plenty of useful mathematical and physical constants, available to the user.

Users are strongly encouraged to adhere to the FLUKA style by *not spoiling double precision* (use real numbers in D-scientific notation or integer numbers) and adopting the constants defined in this file for maximum accuracy, e.g.:

0, 1, 2, 2/3

```
PARAMETER ( ZERZER = 0.D+00 )
PARAMETER ( ONEONE = 1.D+00 )
PARAMETER ( TWOTWO = 2.D+00 )
PARAMETER ( TWOTHI = TWOTWO / THRTHR )
```

π, 2π

```
PARAMETER ( PIPIPI = 3.141592653589793238462643383279D+00 )
PARAMETER ( TWOPIP = 6.283185307179586476925286766559D+00 )
```

e

```
PARAMETER ( ENEPER = 2.718281828459045235360287471353D+00 )
```

Speed of light c in cm/s

```
PARAMETER ( CLIGHT = 2.99792458D+10 )
```

Avogadro number N_A (mol-1)

```
PARAMETER ( AVOGAD = 6.0221367D+23 )
```

Boltzmann constant K (J/K)

```
PARAMETER ( BOLTZM = 1.380658D-23 )
```

Electron mass m_e (GeV/c2)

```
PARAMETER ( AMELCT = 0.51099906D-03 )
```

Conversion factor 1’000 from GeV to MeV

```
PARAMETER ( GEVMEV = 1.0D+03 )
```

iounit.inc

Pre-defined input/output unit numbers included in all FLUKA routines and reserved for FLUKA.

standard output unit:

```
      PARAMETER ( LUNOUT = 11 )
```

standard error unit:

```
      PARAMETER ( LUNERR = 15 )
```

Hence, your scoring card output unit numbers in the .inp file should start from 21.

The user can write there own messages from user routines, e.g.:

```
      WRITE ( LUNOUT, * ) ' My routine has been called!'
      WRITE ( LUNERR, * ) ' Invalid value in my routine!'
```

caslim.inc

/CASLIM/ is needed to decide when to stop the run

*	TRNLIM	=	if cpu_time_left < t_{lim} the run will be ended
*	Tpmean	=	is the average time needed for the following of one beam particle
*	Tprmax	=	is the maximum time needed for the following of one beam particle
*	Trntot	=	the cumulative time needed to follow the beam particles
*	Ncases	=	maximum number of beam particles to be followed (modulo 1,000,000,000)
*	Mcases	=	maximum number of beam particles to be followed in excess of 1,000,000,000, divided by 1,000,000,000
*	Ncase	=	current number of beam particles followed (modulo 1,000,000,000), i.e. **current event number**
*	Mcase	=	current number of beam particles followed in excess of 1,000,000,000, divided by 1,000,000,000
*	Ncoinc	=	flag used by the detect option to know if the Ncase particle has or has not to be considered in coincidence with the previous one (if they have the same Ncoinc/Mcoinc they belong to the same event)
*	Mcoinc	=	flag as Ncoinc, accounting for Ncase > 1,000,000,000
*	Lpseed	=	if .true. seeds will be printed for any history
*	Levtdt	=	if .true. a few data will be printed at each history
*	Lcrrfl	=	if .true. rfluka.stop must be created and the run stopped

flkstk.inc

FLuKa STacK contains all the information on the particles currently in the stack, waiting for transport

```
*    WTFLK(NPFLKA)    =  particle statistical weight                        NPFLKA  = stack pointer
*    PMOFLK(NPFLKA)   =  particle (laboratory) momentum (GeV/c)
*    TKEFLK(NPFLKA)   =  particle (laboratory) kinetic energy (GeV)
*    XFLK, YFLK, ZFLK (NPFLKA)      =  particle position x-coordinate, y-coordinate, z-coordinate
*    TXFLK, TYFLK, TZFLK (NPFLKA)   =  particle direction x-component, y-component, z-component
*    TXPOL, TYPOL, TZPOL (NPFLKA)   =  particle polarization x-component, y-component, z-component
*    DFNEAR(NPFLKA)   =  distance to the nearest boundary
*    AGESTK(NPFLKA)   =  age of the particle (seconds)
*    EKPSTK(NPFLKA)   =  kinetic energy of the last inelastic interaction parent
*    AKNSHR(NPFLKA)   =  Kshort component of K0/K0bar
*    CMPATH(NPFLKA)   =  cumulative path travelled by the particle since it was produced (cm)
*    ILOFLK(NPFLKA)   =  particle nature
*    LOFLK(NPFLKA)    =  particle generation
*    NRGFLK(NPFLKA)   =  particle region number
*    NLATTC(NPFLKA)   =  particle lattice cell number
```

genstk.inc

GENerator STacK contains all the information on the secondary particles *(secondaries)* created in discrete events

```
*    NP                 =  number of (light) secondaries        IP = 1, …, NP
*    Kpart (ip)         =  nature of the IPth secondary
*    Cxr, Cyr, CZr (ip)  =  x-axis, y-axis, z-axis direction cosine of the IPth secondary
*    Tki  (ip) =  laboratory kinetic energy of IPth secondary (GeV)
*    Plr  (ip) =  laboratory momentum of the IPth secondary (GeV/c)
*    Wei  (ip) =  statistical weight of the IPth secondary
```

fheavy.inc

/FHEAVY/ is the storage for **heavy secondaries** created as a result **of a nuclear reaction**

*	`NPHEAV`	=	number of secondaries	**`IP = 1, …, NPHEAV`**
*	`KHEAVY(IP)`	=	**type** of the IPth secondary (3 = ^{2}H, 4 = ^{3}H, 5 = ^{3}He, 6 = ^{4}He, 7-12 = as specified by **`IBHEAV`** and **`ICHEAV`**)	
*	`INFHEA(IP)`	=	possible extra info for the IPth secondary	
*	`CXHEAV(IP)`	=	direction cosine of the IPth secondary with respect to x-axis	**direction**
*	`CYHEAV(IP)`	=	direction cosine of the IPth secondary with respect to y-axis	
*	`CZHEAV(IP)`	=	direction cosine of the IPth secondary with respect to z-axis	
*	`TKHEAV(IP)`	=	**kinetic energy** of the IPth secondary	
*	`PHEAVY(IP)`	=	momentum of the IPth secondary	
*	`WHEAVY(IP)`	=	**statistical weight** of the IPth secondary	
*	`AGHEAV(IP)`	=	"age" of the IPth secondary with respect to the interaction time	
*	`AMHEAV(`*`KP`*`)`	=	atomic masses of the twelve types of evaporated or fragmented or fissioned particles	
*	`AMNHEA(`*`KP`*`)`	=	nuclear masses of the twelve types of evaporated or fragmented or fissioned particles	
*	`ANHEAV(`*`KP`*`)`	=	name of the kp-type heavy particle	**`KP = KHEAVY(IP)`**
*	`ICHEAV(`*`KP`*`)`	=	**charge number** of the kp-type heavy particle	(e.g., **`ICHEAV(5)=2`**)
*	`IBHEAV(`*`KP`*`)`	=	**mass number** of the kp-type heavy particle	

resnuc.inc

/RESNUC/ contains the information on the **residual nucleus** created as a result of a **nuclear reaction**

```
*    Icres   =  residual nucleus atomic number
*    Ibres   =  residual nucleus mass number
*    Amnres  =  residual nucleus nuclear mass (ground state)
*    Ammres  =  residual nucleus atomic mass
*    Eres    =  residual nucleus total energy
*    Ekres   =  residual nucleus kinetic energy
*    PxRES, PyRES, Pzres  =  residual nucleus momentum components
*    Ptres2  =  residual nucleus squared momentum
*    Angres  =  residual nucleus angular momentum (GeV/c fm)
*    AnxRES, ANyRES, ANzres  =  residual nucleus angular momentum components
*    Icestr  =  residual nucleus atomic number before evaporation
*    Ibestr  =  residual nucleus mass   number before evaporation
*    Tvestr  =  residual nucleus excitation energy before evaporation
*    Anestr  =  residual nucleus angular momentum before evaporation
```

trackr.inc

TRACK Recording contains all the information on the particle being currently tracked and its current step

* `Ntrack`	=	number of track segments (normally 1, unless in a magnetic/electric field)	
* `Mtrack`	=	number of energy deposition events along the track	
* `Xtrack, Ytrack, Ztrack (I)`	=	x, y, z coordinate of the end point of the Ith track segment	`I = 0, …, NTRACK`
* `Ttrack(I)`	=	length of the Ith track segment	`I = 1, …, NTRACK`
* `Dtrack(I)`	=	energy deposition of the jth deposition event	`I = 1, …, MTRACK`
* `Dptrck(I)`	=	momentum loss of the jth deposition event	`I = 1, …, MTRACK`
* `Jtrack`	=	**nature** of the particle (for recoils or kerma deposition it can be outside the allowed particle id range, assuming values like 208: "heavy" recoil, 211: EM below threshold, 308: low energy neutron kerma; in those cases the id of the particle originating the interaction is saved inside **`J0trck`**, which otherwise is 0)	
* `J0trck`	=	see above	
* `Etrack`	=	**total energy** of the particle	
* `CxTRCK, CyTRCK, Cztrck`	=	**direction cosines** of the current particle	
* `Wtrack`	=	**weight** of the particle	
* `Cmtrck`	=	cumulative curved **path since particle birth**	
* `Atrack`	=	**age** of the particle	
* `Ltrack`	=	**generation number**	

evtflg.inc

EVenT FLaGging indicates the last interaction type

*	LELEVT	=	Elastic interaction
*	LINEVT	=	**Inelastic interaction**
*	LDECAY	=	Particle decay
*	LDLTRY	=	Delta ray production (Moller and Bhabha included)
*	LPAIRP	=	Pair production
*	LBRMSP	=	Bremsstrahlung
*	LANNRS	=	Annihilation at rest
*	LANNFL	=	Annihilation in flight
*	LPHOEL	=	Photoelectric effect
*	LCMPTN	=	Compton effect
*	LCOHSC	=	Rayleigh scattering
*	LOPPSC	=	Optical photon scattering
*	LELDIS	=	Electromagnetic dissociation
*	LRDCAY	=	Radioactive decay
*	LSRPHO	=	Synchrotron radiation emission

Available tools in the FLUKA library

- `CALL FLABRT ('calling routine name','my message')`
  to abort FLUKA. To be used when an user routine reaches an unacceptable state (or for debugging!)

- `CALL OAUXFI ('file name', LUNRDB, 'OLD', IERR)`
  to open an auxiliary file (sitting in some default locations) for reading its content

- Random number generators:

`... = FLRNDM (XDUMMY)`	uniformly distributed in [0-1)
`CALL FLNRRN (RGAUSS)`	Gaussian distributed (μ=0, σ=1)
`CALL FLNRR2 (RGAUS1,RGAUS2)`	Gaussian distributed uncorrelated pair
`CALL SFECFE (SINT,COST)`	sine and cosine of uniformly distributed azimuthal angle
`CALL RACO (TXX, TYY, TZZ)`	isotropically distributed 3D direction
`CALL SFLOOD ( XXX, YYY, ZZZ, UXXX, VYYY, WZZZ )`	position and direction on a unit sphere to generate uniform and isotropic fluence **inside**

Available tools in the FLUKA library [II]

Get the region number from region name

```
CALL GEON2R ( REGNAM, NREG, IERR )
* Input variable:
* Regnam = region name (CHAR*8)
*
* Output variables:
* Nreg = region number
* Ierr = error code
* (0 on success, 1 on failure)
```

(… and vice-versa, do you really need it?*)

```
CALL GEOR2N ( NREG, REGNAM, IERR )
* Input variable:
* Nreg = region number
*
* Output variables:
* Regnam = region name (CHAR*8)
* Ierr = error code
* (0 on success, 1 on failure)
```

All **regions** are internally **treated as numbers**, both in FLUKA and user routines.

*When coding these, you should **`CALL GEON2R`** to *translate your region name into the respective number and save the latter for runtime use*. This has to be done only once the first time your routine is called (use **`IF (LFIRST) THEN`**).

Name based declaration in the inputfile

```
*Black hole
BLKBODY      5 +blkbody -void
*Void around
VOID         5 +void -target
*Target
TARGET       5 +target
```

Region numbers and names echoed in .out

```
  1   BLKBODY      1   BLCKHOLE   OFF                  0.00000E+00     9.99852E+04
                   (   1  BLCKHOLE OFF )
  2   VOID         2   VACUUM      OFF                 0.00000E+00     9.99852E+04
                   (   2  VACUUM   OFF )
  3   TARGET      12   COPPER      OFF                 0.00000E+00     9.99852E+04
                   ( 12   COPPER   OFF )
```

Available tools in the FLUKA library [III]

- Adaptive Gaussian quadrature

```
EXTERNAL FINTEG
 IOPT = 3
 ACCURA = 1.0D-4
 ... = FLGAUS ( FINTEG, XA, XB, ACCURA, IOPT, NXEXP )
```

Gives the integral over the (XA,XB) interval of the product between X**NXEXP and the FINTEG function, to be coded by the user as a separate `DOUBLE PRECISION FUNCTION FINTEG (X)`

- Real solutions of 3rd order equation

```
SUBROUTINE RADCUB ( AA0, AA1, AA2, AA3, X, X0, NRAD )
```

Computes real solutions of the equation: A0*X3 + A1*X2 + A2*X + A3 = 0

The solutions are put in the array X; if there is only one real solution it is put into X(1), while X(2) and X(3) are set to -1.D32. If A0=0 (and A1=0) the routine computes standard solutions of a second (or first) degree equation. If it no real solution exits, the whole array X is set to -1.D32. It is possible to compute solutions with a scale factor X0 to avoid loss of significance with very large or very small numbers. The NRAD flag records the number of real solutions found.

Available tools in the FLUKA library [IV]

`DOUBLE PRECISION FUNCTION GAMFUN ( X )`

Calculates the double precision complete **Gamma function** for double precision argument X

`SUBROUTINE RORDIN ( RVECT, ICORR, LEN )`

Rearranges a real array RVECT in increasing order

`SUBROUTINE RORDDE ( RVECT, ICORR, LEN )`

Rearranges a real array RVECT in decreasing order

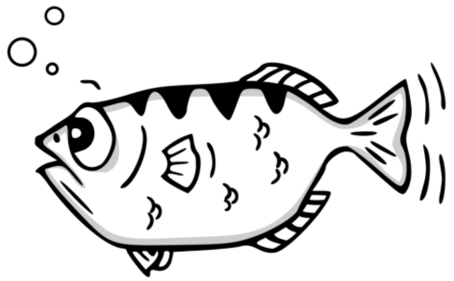
`DOUBLE PRECISION FUNCTION FLGNDR ( X, LMAX, PLGNDR )`

Function for **LeGeNDRe polynomials**

Computes P_{lmax} (x) and stores all values P_i (x) for i=0,l_{max} in the PLGNDR array

Debugging!

- Linux offers various tools that are integrated in the FLUKA environment.

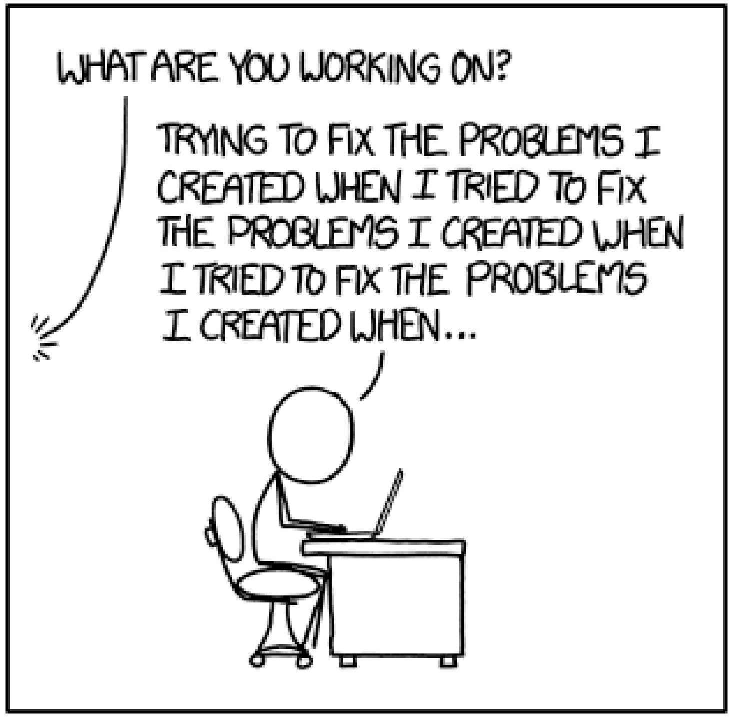


GDB: The GNU Project Debugger
https://sourceware.org/gdb/

$\longrightarrow$

cgdb
the curses debugger

CGDB (its frontend)
https://github.com/cgdb


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

https://xkcd.com/1739/

Instructions via terminal

```
[me@localhost myFolder]$ /usr/local/fluka/bin/rfluka -g cgdb -e executable/fluka.x -N0 -M1 test.inp
```

Normal call **Debug me!** Executable to be debugged

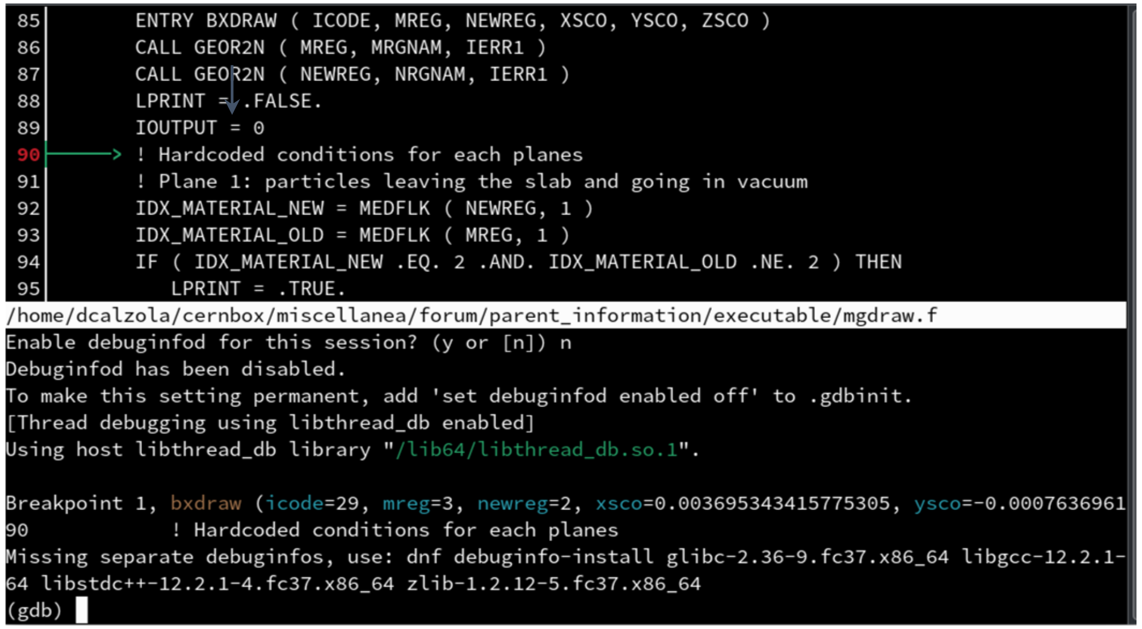
Debugging (example)

```
[me@localhost myFolder]$ /usr/local/fluka/bin/rfluka -g cgdb -e executable/fluka.x -N0 -M1 test.inp
```

```
New UI allocated
(gdb) b 86
Breakpoint 1 at 0x407b60: file mgdraw.f, line 90.
```

We ask for a breakpoint on line 86, it accepted it on line 90

```
(gdb) r
Starting program: myFolder/executable/fluka.x myFolder/test.inp
```



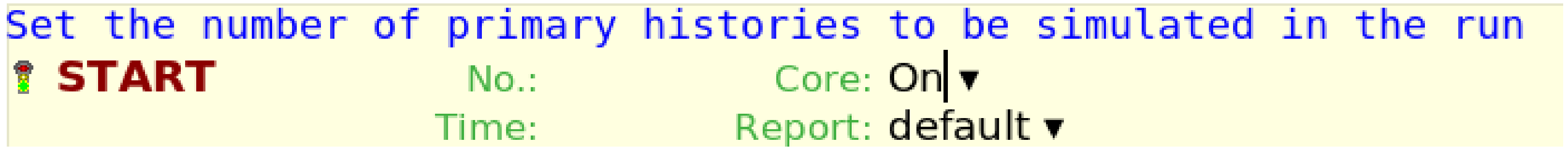
```
85        ENTRY BXDRAW ( ICODE, MREG, NEWREG, XSCO, YSCO, ZSCO )
86        CALL GEOR2N ( MREG, MRGNAM, IERR1 )
87        CALL GEOR2N ( NEWREG, NRGNAM, IERR1 )
88        LPRINT = .FALSE.
89        IOUTPUT = 0
90 ----> ! Hardcoded conditions for each planes
91        ! Plane 1: particles leaving the slab and going in vacuum
92        IDX_MATERIAL_NEW = MEDFLK ( NEWREG, 1 )
93        IDX_MATERIAL_OLD = MEDFLK ( MREG, 1 )
94        IF ( IDX_MATERIAL_NEW .EQ. 2 .AND. IDX_MATERIAL_OLD .NE. 2 ) THEN
95           LPRINT = .TRUE.
/home/dcalzola/cernbox/miscellanea/forum/parent_information/executable/mgdraw.f
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, bxdraw (icode=29, mreg=3, newreg=2, xsco=0.003695343415775305, ysco=-0.0007636961
90             ! Hardcoded conditions for each planes
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.36-9.fc37.x86_64 libgcc-12.2.1-
64 libstdc++-12.2.1-4.fc37.x86_64 zlib-1.2.12-5.fc37.x86_64
(gdb) 
```

Debugging window: in the top part there is the source, while instruction are given in the bottom. You can:

- **[s]**tep to the next instruction
- go to the **[n]**ext line
- **[b]**ack**[t]**race: prints a stack trace, listing each function and its arguments
- **[p]**rint variable value
- …and much more!

Post-mortem debugging

You can ask FLUKA to produce a core dump when it ends unexpectedly.
To do so, you need to set WHAT(4) of the **START** card equal to 1:



Set the number of primary histories to be simulated in the run
START No.: Core: On
Time: Report: default

It may happen that on your system core generation is inhibited by default. To overcome the latter, – on Fedora – one needs to fill the */proc/sys/kernel/core_pattern* system file with the *core.%p* content, by the *echo* command:

```
echo "core.%p" > /proc/sys/kernel/core_pattern
```

When a core dump is produced, the user can perform a post-mortem debugging invoking on the terminal:

```
gdb myfailingexecutable core.12345
```

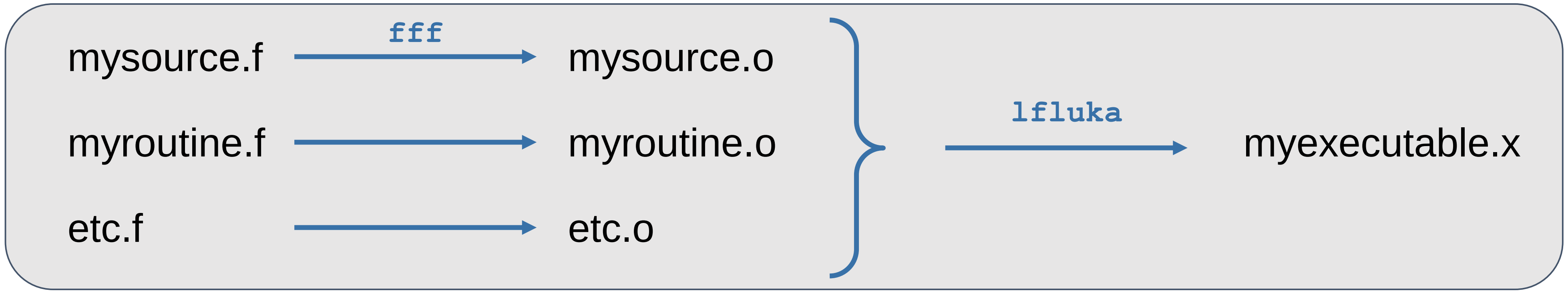
Looking at the backtrace normally helps to identify the failing routine/line, even in the absence of the FLUKA source code.

FLUKA

What is available for users

- Once the user routine has been written or modified, the user needs to:
  a. Compile each source routine into an object file: `/usr/local/fluka/bin/fff`
  b. Link each object file to the fluka executable*: `/usr/local/fluka/bin/lfluka`

- As good practice, try to keep everything in your working directory



Do everything outside the installation folder

```
fff xxx.f
fff yyy.f
lfluka -o myfluka xxx.o yyy.o
```

- For simulations requiring the DPMJET and RQMD packages, add the option *-d*, or use *ldpmqmd* instead of *lfluka*