FunRootAna - analysis in functional approach

Tomasz Bold
AGH University of Krakow, Dept. of Physics and Applied Comp. Sciences

AGH
NATIONAL SCIENCE CENTRE POLAND
UMO-2023/51/B/ST2/00920

Functional container

Functional programming is regaining popularity in specific domains.
This is due to several multi-paradigm languages like F#, Scala, Kotlin,…
The functional approach shines in processing collections (e.g. Spark) where units of computations are projected into filter-map-reduce paradigm.
The C++ views, debuted in standard c++20, provide similar "look and feel" but allow to mutate the containers. A simpler solution is provided by **FunRootAna**. The functionality is tailored for typical analyses.
Common elementary analysis tasks are:
- filter objects,
- extract quantity out of object,
- accumulate,
- …

All of these are best approached assuming immutability of the data (e.g. Spark RDD). FunRootAna provides **functional API** for any c++std container. Main features of FunRootAna functional container are:
- **complete set of functionalities,**
- **lazy evaluation,**
- **immutable container,**
- **convenience macros to reduce C++ boiler plate.**

Construction Lazy Functional Container:

lazy_view(container) // for any container with begin & end
lazy_view(array, size) // for plain array
one_own(value) // for single value
geometric/arithmetic/iota/random_streams // infinite sequences

Operations Lazy Functional Container:

map(function) // transforms
filter(predicate) // select according to function
take/skip(N, stride) **take/skip_while**(predicate) // elements range selection
foreach(procedure)
chain(other) // concatenation
cartesian(other) // all possible pairs
zip(other) // pairwise combined
all/any/count(predicate) // predicates
…. // and more

Operations can be chained and lazy evaluation used whenever possible:
e.g. data.**filter**(F(_>4)).**filter**(F(_%3 == 0)).**map**(F(_*_)).**filter**(F(_+2)) - no-op
None of the operation mutate containers.

N code lines ⟹1 per plot

The effectiveness of data exploration depends on flexibility of processing system. E.g. to define and fill a histogram one should require a single line.
FunRootAna streamlines commonly tedious tasks such that the construction/registration/usage of one histogram/efficiency plot/graph takes only a single line.

Creation (similar to ROOT):
HIST1("name", "title", nbins, min, max)
HIST1V("name", "tile", std_vector_of_bin_edges)
+ HIST2, HIST3, EFF1, EFF2, EFF3, PROF1, PROF2, (with V variants) & GRAPH

These are responsible for booking/registering/discovering histograms.
Thanks co combination of macros & static lambdas convenient object-singleton pattern. Typical usage:
data_in_lfv >> HIST1("data", ";unit", 100, 0, 100);

Histogram context HCONTEXT:
Scope context switcher streamlines generation of histogram varianats
Filling operator >>:
Provided set of operators streamlining filling from various datum.

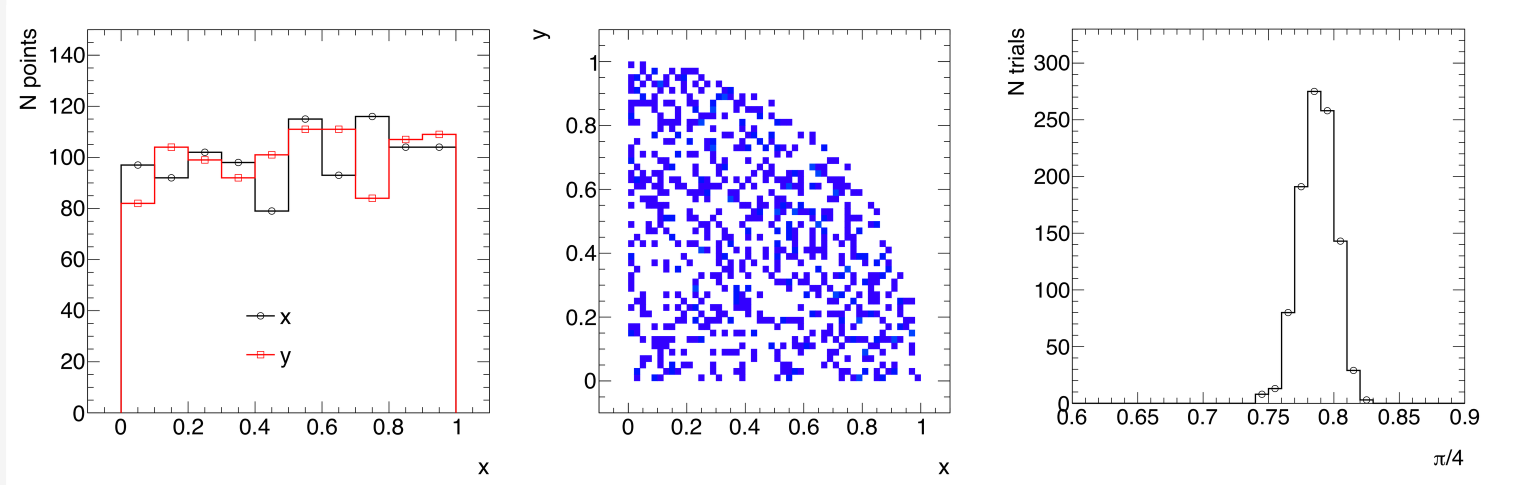
x >> HIST1(…) // x any elementary type, double, int, std::string, …
std::make_pair/tuple(1, 5) >> HIST1(…)/HIST2(…)/PROF1(…)/EFF1(…) // pairs & tuples
std::optional<double> y{};
y >> HIST1(…) // optional handled as argument (skips fill operation)
lazy_c.map(F(_.a_value)) >> HIST1(…)
lazy_c.map(F(std::make_pair(_.a_value, _.b_value))) >> HIST2(…)
// lazy collections can fill the histograms as well

ROOT Trees reading supported n the same manner

Example analysis code*

```
const size_t N = 1000;
auto randToUniform = F( (_ % 1000)/1000; ); // shorter variant
auto x_vec = lfv::crandom_stream().take(N).map(randToUniform).stage();
auto y_vec = lfv::crandom_stream().take(N).map(randToUniform).stage();
auto x = lazy_view(x_vec);
auto y = lazy_view(y_vec);
x >> HIST1("x", "", 10, 0, 1);
y >> HIST1("y", "", 10, 0, 1);
auto points2d = x.zip(y);
points2d >> HIST2("x_vs_y", ";x;y", 100, 0, 1, 100, 0, 1);
auto inCircle = points2d.filter(F(std::hypot(_.first, _.second) < 1)); // no-op
inCircle.size() / static_cast<double>(N) >> HIST1("pi_over_4", "", 100, 0, 1);
```

Estimate value of π via trivial MC integration



*in fact this calculation can be done in single, not so long line

Future

Functionality provided in FunRootAna is sufficient to perform typical analyses. The build in collection immutability is a good start. ROOT histograms filling requires attention currently but will be made safe in future ROOT versions (global lock solution to slow). Additional functionality for concurrent trees reading is envisaged as well.
In typical analysis, an evaluation of systematic effect can be carried out together with main analysis. A support for automatic handling of that aspect is planned.