Pluto Script Manual

July 10, 2023, Pluto version `6.01-devel`

I. Fröhlich[1]

[1] Institut für Kernphysik, Goethe-Universität, 60438 Frankfurt, Germany

The idea behind scripting with Pluto ("PlutoScript") is to combine objects like histograms and ntuples via a flexible batch language with the event loop and the data base. The commands of the batch script are compiled by Pluto via the class `PBatch` in advance and can be executed as often as required without additional delay. This execution can be done inside the event loop which allows for modifications of particles, e.g. to write acceptance filters or include smearing effects.
Furthermore, it is also possible to call many public available Pluto-methods of the `PUtils` class and all methods of the class `PParticle` (i.e. `TLorentzVector`).
As the script can combine ROOT-objects (histograms and ntuples) with the event loop, particle properties can be converted very quickly to histograms and ntuples.
And finally, the script can be packed together with acceptance and resolution matrices to form a universal, common format for a detector description.

Contents

1 The syntax

1.1 Execution of the script commands

The first examples deal with the usual “hello world” for which the `echo` command is used (which works in a similar way as the echo of the unix shell).

In order to execute script commands via the ROOT command shell the quickest method is to use the global batch singleton:

```
makeGlobalBatch()->Execute("echo Hello world");
```

Additional new batch objects can be constructed as well:

```
PBatch * batch = new PBatch();
batch->AddCommand("....");
batch->AddCommand("....");
batch->Execute();
```

In this case, the commands are compiled instantly after `AddCommand`, and are executed later with `Execute`. After the script has been compiled without error, it can be executed as often as required.

In order to add command lines to the event loop the `Do()` method of the `PReaction` (here defined as `r`) can be used:

```
r->Do("echo Yet another event....");
```

These commands are executed in each event *after* the particles of the event have been sampled.

In the next sections, the following notation is used: commands in quotation marks are Pluto batch commands which can be encapsulated into the event loop or any of the other execution methods as described above; other commands are ROOT-CINT commands.

1.2 Command line syntax

One command line can be composed of several single commands. Like in C++ the semi-colon is used as the delimiter:

```
"command1; command2; ..."
```

There are cases where the script is line-sensitive, i.e. it makes a difference if several commands are written in one single line or splitted into several lines.

1.3 Variables

Variables are always `Double_t`'s and assigned automatically without any definition or constructor:

```
"myvar = 0.2;"
```

Therefore, one should keep in mind that *typos* (e.g., a wrong variable name) are not catched.

To get a list of variables, one can access the Pluto global data base via:

```
makeDataBase()->ListEntries(-1,1,"*name,batch_value");
```

It is possible to obtain the pointer to the variable via:

```
Double_t* makeStaticData()->GetBatchValue(varname,makeflag);
```

with `varname` as the name of the (new) batch variable. If `makeflag` is set, the variable is constructed, in the other case a NULL is returned if the variable does not exist.

For example

```
makeStaticData()->GetBatchValue("myvar",0);
```

will return the pointer to the variable from the example above. Consequently, the content of this variable can be changed also via the ROOT macro/shell. This allows to communicate with the script (e.g. for the control and setting of filters).

> All variables are globals. They can be changed and acessed over the entire Pluto system.

Variable names must contain alphanumeric characters only plus the "_", they must not start with a digit.

> It is not recommended to use variables with a trailing "_". Such names are reserved for system variables.

In addition, variables may start with a hash ("#"), these are interpreted by the `PReaction` loop as a filter variable.

1.4 The echo command

Variables can be printed via the `echo` command, by adding a "$" in front of the variable name, e.g.:

```
"echo The value of myvar is $myvar"
```

If a definition has a trailing "$", the corresponding variable cannot be replaced by an operation or calculation. In this cases, only the (precalculated) variable can be used.

The format of the output can be controlled with the same syntax as in the c printf syntax, by using a format string which is in this case, however, embedded in the variable name. The format string must have a trailing '%', like in c, but it must be completed with a '%' as well, in order to have a delimiter between the format and the variable.

Syntax:

```
"echo The value of myvar is $%format%myvar"
```

The format can be, e.g. 'i' (for integers), 'x' (for hex numbers), 'c' (conversion to a character) but also of more more complex nature, like '.2E' (float with 2 digits after the point) or '0#.8x' (8-byte hex). For a complete list of features reference is made to the c-manual.

1.5 Arithmethic operations

The batch script can use all operations which are allowed by `TFormula`. This means, all normal operations like "+", "-", "*", "/", but also boolean, are accessible. In particular, all functions of `TMath` can be used.

This syntax has been enhanced to include Pluto variables and additional features (see Sec. 2) into the `TFormula` syntax.

```
"echo $myvar; myvar = myvar + 0.1; echo $myvar;"
```

In this example, each time the line is executed, the variable `myvar` is increased by 0.1.

1.6 Conditions and branches

1.6.1 Tests using If

With the `If` command a conditional execution can be realized.

Syntax:

```
"if condition; commands...;"
```

There are some differences to the "normal" c-like `if`: there are no curly brackets, and there is no else. The `condition` can be either a variable or the result of a calculation. The commands after the `if` are executed, if the value of the condition is non-zero. This is quite clear for boolean calculations as they are translated into 0 or 1. All other non-zero floating point values are treated as 1. As described above, all functions and arithmetic operations which are allowed by `TFormula` can be used.

One simple example would be a test for a value:

```
"if myvar < 0.3; echo myvar too small;"
```

1.6.2 The else command

If a test failed, the `else` command can catch this situation and continue the execution. The `else` command can be used in conjuntion with `if`:

```
"if condition; commands...; else; other commands..;"
```

or even stand alone:

```
"commands...; else; other commands..;"
```

This construction can be helpful, if the `commands` require objects which are not present or empty. *Pluto does not catch empty particle objects, it stops the command chain and returns a false (e.g. to the `PProjector`), so if you need a error message one has to use the `else` command.*

1.6.3 Conditions on equality

The equality operator ("==") can be used as usual:

```
"myvar = 1; if(myvar == 1); echo myvar is one"
```

As the variables are always floating points and therefore can be unprecise, the tilde comparator ("~") provides some "equality" feature:

```
"myvar = 1; if(myvar ~ 1); echo myvar is similar to one"
```

The result of this operator is true, if the difference between the two operands is smaller than 0.5.

Tests for zero's can be done in the following ways:

```
"myvar = ...; if(myvar);  echo myvar is not zero";
"myvar = ...; if(!myvar); echo myvar is zero"
```

1.6.4 Goto

With `goto` local jumps inside the script can be realized. Syntax:

```
"label: ..."
"goto label;"
```

In the following example the `goto` is used to construct a “for-loop”:

```
"myvar = 0.0; loop: myvar=myvar + 0.1;
   if myvar < 0.7; echo $myvar; goto loop;"
```

1.6.5 Gosub and return

This allows to jump to subroutines, e.g. to do common calculations.

Syntax:

```
"gosub label;"
"..."
"label: ..."
"..."
"return;"
```

1.6.6 Exit

Stops the script and returns to the Pluto event loop (or CINT). Syntax:

```
"exit;"
```

2 Additional features of the scripting language

2.1 `PParticle`-objects

In addition to floating point variables the script can use `PParticles` objects. Since particles are inherited from Lorentz vectors, at least their momenta and the masses (or energies) have to be defined. This can be done by the following constructor functions.

2.1.1 The P3M constructor

This constructor uses the 3-momentum (p_x, p_y, p_z) and the invariant mass as arguments. Syntax:

```
"mypar = P3M(px,py,pz,mass);"
```

The following example creates a particle with eta mass and 2 GeV momentum in z-direction:

```
"mypar = P3M(0,0,2.,0.547);"
```

2.1.2 The P3E constructor

This constructor uses the 3-momentum (p_x, p_y, p_z) and the energy as arguments. Syntax:

```
"mypar = P3E(px,py,pz,e);"
```

2.1.3 Copy constructor

New particle objects are created automatically. Syntax:

```
"newpar = mypar;"
```

Here, `newpar` is a new object, and not a reference to `mypar`.

2.2 Access to `PParticle` methods

The script can use all public methods of `PParticle` (and therefore also of the `TLorentzVector`) which uses only `void`'s, `Int_t`'s, or `Double_t`'s as arguments/return values.

The following example sets the particle id of the above-created vector to the id of the η meson:

```
"mypar->SetID(17);"
```

The more commonly used feature is to read observables of a particle. Some usual examples (e.g. to fill histograms or ntuples) are:

```
"mass  = mypar->M();"
"angle = mypar->Theta();"
```

For a complete list, reference is made to the ROOT manual.

2.3 Additional build-in methods

2.3.1 `obj->Angle(target)`

Opening angle between particle `obj` and particle `target`.

2.3.2 `obj->Boost(target)`

Boosts the particle `obj` into the rest frame of particle `target` (replaces `obj->Boost(-target->GetBoostVector())`).

2.3.3 `obj->Rot(target)`

Rotate particle `obj` such, that `target` would point to z-Direction.

2.3.4 `composite->GetBeam()`

Returns a `PParticle` object (the beam) from a composite object (like `[p+p]`)

2.3.5 `composite->GetTarget()`

Same as above.

2.4 Access to the particle stream of the event loop

The link to the Pluto event loop is done via the `PProjector` class, which internally executes one or more batch objects. By using the `Do()` method as mentioned in Sec. 1.1, a `PProjector` is generated automatically, which consequently connects the particles of the event loop with the batch script.

This method works both for complete chains of the `PReaction` and single decay steps in `PChannel`. In the latter case, particles can be modified before their consecutive decay.

In this subsection, at first place the syntax to access the particle stream will be explained. The connection to histograms will be further described in Sec 3.

> The script can access only particles which are stored in the file. If the "tracked only" option of `PReaction` is used, the unstable (i.e. decayed) particles can not be read by the script.

The particle objects of the event loop are marked with square brackets. The general syntax is:

```
"mypar = [id,num]"
"myvar = [id,num]->...()"
```

with `id` as the Pluto particle name, and `num` as the number of the individual particle of the species `id` in the current event, counting from 1.

If `num` is not defined or equal to zero, the "current position" is used. This is by default the first particle. Inside loops (for a further description see 2.6) the current position is the loop position.

Examples:

```
"[eta,1]" //first eta in event
"[eta]"   //current position
```

There is a placeholder ("$*$") for all particle species, i.e. with

```
"mypar = [*,num]"
```

any particle can be accessed, where "num" has to be the number of the particle.

It is, however, also possible to use variable position numbers:

```
"num=1; mypar = [id,$num]"
```

2.5 Adding particles to the stack

2.5.1 New branches

By default, Pluto has only one particle stack (i.e. a `TBranch`) with the name "Particles". It is, however, possible to create new branches with the command `Branch`:

```
"branchid = Branch(newname);"
```

with `newname` as the name of the `TBranch` as it will appear in the ROOT-file, and `branchid` as the number of the branch (the default branch has 0 as its number).

2.5.2 Pushing particles

New particles, which are generated by `P3M` or `P3E`, or the copy constructor, can be added to the stack of the event loop by the `push` command. Syntax:

```
"push(mypar);"
```

In this case, the particle is pushed on the default branch. A complete example which reads data from a text file and adds particles can be found in 3.5.2.

An alternative is to use a method:

```
"mypar->Push(branchid);"
```

E.g. this loop pushes all particles on a second `TBranch`:

```
"foreach(*); [*]->Push(Branch(particles_sim));"
```

2.6 Particle loops

Pluto has two build-in commands for easy looping over the particle array (a self-made loop is discussed in 2.8.4).

2.6.1 The `foreach` loop

Syntax:

```
"foreach(id); ... [id] ..."
```

This executes the commands after `foreach` until the end of the line, repeating it for each particle of the species `id` of the current event. For each execution the current position of `[id]` is shifted by one and replaced by the next particle, as discussed above. Via this command loop, operations and calculations can be easily repeated for each occurence of a particle named `id`.

Example(s):

```
"foreach(p); mom = [p]->P(); echo proton momentum $mom"
"foreach(*); id = [p]->ID(); echo found particle with $id"
```

2.6.2 The `formore` loop

Syntax:

```
"loop: ..."
"...[id]..."
"formore(id); goto loop;"
```

The commands after `formore` are called, if particle objects of the species `id` are left over. This allows to construct longer command lists, which spans over more than one line.

2.7 Access to system values

The batch script can be used to read and change Pluto system variables.

Change Pluto system variables only when you know what you are doing

2.7.1 `_system_version`

Pluto version number.

2.7.2 `_system_weight_version`

=1 (default): New version, weight is $1/N_{ev} \cdot \Pi b_i \cdot \Pi W_j$ with the branching ratios b_i and the weights of all models W_j.

=0: Old version, where weight is set by the seed particle and propagated.

2.7.3 `_system_unstable_width`

Limit (in GeV) of the width where particles are treated as stable particles.

2.7.4 `_system_thermal_unstable_width`

Limit (in GeV) of the width for enabling the mass sampling in thermal models. If the particle width is below the limit, the mass is fixed, and only the energy is sampled. For particles with widths equal or larger the defined limit, 2-dimensional sampling is applied. I.e., the 1-dimension sampling is disabled with:

```
*(makeStaticData()->GetBatchValue("_system_thermal_unstable_width")) = -1;
```

2.7.5 _system_alloc_verbosity

=1 (default): Print allocation infos.

=0: Quiet mode.

2.7.6 _system_printout_percent

[1]

Defines the printout steps in percent (default: 20%).

2.7.7 _system_max_failed_events

`PReaction` will stop the event loop if the number of failed events is larger (default 10000).

2.7.8 _system_total_events_to_sample

[2]

Number of events to be sampled in total (taken from `Loop()`). Readonly!

2.7.9 _system_total_event_number

[3]

Number of events which have beed sampled already. Readonly!

2.7.10 _system_particle_stacksize

This defines the size of the stack in `PReaction` for bulk decays, embedded particles, etc. Default: 500.

2.7.11 _system_force_m1n

Uses the ROOT based model M1N for decays with 1 unstable and 2 stable daughters instead of M3.

[1] since v5.43
[2] since v5.43
[3] since v5.43

2.7.12 `_event_vertex_x / ..._y / ..._z`

Default event vertex. Can be changed/smeared inside event loop.

2.7.13 `_event_plane / _event_impact_param`

Parameters for fireball sampling.

2.8 Access to data base values

The script is able to read basic (static) particle properties from the data base (values are read-only). Syntax:

```
"... = id.varname;"
```

where `varname` is the data base variable name. Some usefull variable names are listed below:

2.8.1 mass

Static pole mass of a particle. Example(s):

```
"mass = eta.mass; echo $mass;"
"[rho0]->SetM(rho0.mass);"
```

2.8.2 pid

Pluto particle id. Example (with the definition above in Sec. 2.2):

```
"mypar->SetID(eta.pid);"
```

2.8.3 width

The static width of the particle.

2.8.4 npar

Number of particle objects in the current event. This allows to build loops by hand:

```
"cur = 0; myloop: if !(cur ~ p.npar); cur = cur + 1;
    echo proton $cur; [p,$cur]->Print(); goto myloop"
```

2.8.5 cpos

Current position of the particle object (e.g. in foreach loop)

2.9 Access to `PUtils`

The script offers to call all methods which are available in the `PUtilsREngine` class (which is a wrapper to `PUtils`). This can be used, e.g., to access the random number generator:

```
"var = sampleFlat();"
"echo The number is $var;"
```

The result of `var` is a random number between 0 and 1. Via this means it is possible to use various methods, which are, e.g., essential for acceptance filtering and momentum smearing. Some other usefull example is the sampling of a gaus function:

```
"var = sampleGaus(10.,0.1);"
```

which also can make use of variables as arguments:

```
"mean = 10;sigma=0.1;"
"var = sampleGaus(mean,sigma);"
```

2.10 Access to the Pluto models

Calculations and numbers can be retreived also by all build-in Pluto models (and models of the plug-ins, after their activation).

Syntax:

```
"var = {modelname}->X(...)";
```

with `modelname` as the unique name of the model , as returned by `makeDistributionManager->Print("root")` (in addition with curly brackets) and `X(...)` as one of the methods of the class `PChannelModel`, such as:

2.10.1 `SampleTotalMass()`

This returns the sampled mass of particle models. The following example:

```
"rhomass = {rho0_bw}->SampleTotalMass();"
"rho = P3M(0,0,0,rhomass); push(rho)"
```

adds a rho with the mass-dependent Breit-Wigner model to the particle stack.

2.10.2 `GetWeight(mass)`

Returns the weight of the model. Can be used, e.g., to fold the event weight ("`_w`") with a local weight of a single model.

2.10.3 `GetWidth(mass)`

Returns mass-dependent width of particle models.

2.10.4 `GetBR(mass)`

Returns mass-dependent branching ratios of decay models. The following example re-weights a (free) ρ^0 with the branching ratio of a di-electron decay:

```
"rhomass = [rho0]->M(); _w = {rho_picutoff_e-_e+}->GetBR(rhomass);"
```

3 Connection to ROOT objects

The batch script can be used to fill histograms and/or ntuples. This will be outlined in the following section.

3.1 Filling histograms in the event loop

Syntax:

```
r->Do(histo,"command");
```

with `histo` as a pointer to a 1-, 2-, or 3-dimensional histogram, and `command` as the batch command which has to calculate `_x, _y, _z` as the required position on the axes to fill the histogram. For the weight the current event weight is used, which can be accessed or changed via the variable `_w`.

Example(s):

```
//Missing mass of the p2 and pi0 pair:
r->Do(histo1,"miss= [p + p]- ( [p,2]+ [pi0] );_x=miss->M()");
//Theta of the first proton
r->Do(histo3,"_x= ([p,1]->Theta() * 180.)/TMath::Pi()");
```

N.b. that the histogram is not filled, if one of the particle objects is empty. No error message is dumped. This is not a bug, it is a feature! See 1.6.2 for a message possibility.

3.2 Filling ntuples in the event loop

Syntax:

```
r->Output(ntuple,"var1 = ...; var2 = ...; ...");
```

where `var1, var2, ...` are the variables of the ntuple.

Example(s):

```
TNtuple *ntuple = new TNtuple("ntuple","eta events",
                      "eta_px:eta_py:eta_pz:eta_m");
PReaction r("3.5","p","p","p p eta");
r.Output(ntuple,"eta_px = [eta]->Px(); eta_py = [eta]->Py();
                 eta_pz = [eta]->Pz(); eta_m  = [eta]->M()");
```

3.3 Filling text files in the event loop

The `echo` command can be redirected to a text file. Syntax:

```
r->Output("filename.txt","echo $var1 ....");
```

This works also with multiple lines. After the `Output()` has been used, all echo commands are redirected to this file until `CloseFile()` is used. Example:

```
r->Output("bla.lst","a=[p,1]->P(); echo $a");
r->Do("a=[p,2]->P(); echo $a");
r->Do("echo newevent");
r->CloseFile();
r->Do("echo stdout");
```

3.4 Input via ntuples

Syntax:

```
r->Input(ntuple);
r->Do("... = var1; ... = var2; ....");
```

The writing of histograms, ntuples, and the reading of ntuples can be combined.

Example(s):

```
PReaction r;
r.Input(ntuple);
r.Do("myeta = P3M(eta_px,eta_py,eta_pz,eta.mass)");
r.Do("cm = P3E(0.000000,0.000000,4.337961,5.376545) ;
      myeta->Boost(cm);");
r.Do(histo,"_x= myeta->CosTheta();");
r.Loop()
```

3.5 Prologue and Epilogue interfaces

3.5.1 Input via text files

Text files in any possible format can be read in a generic way with the `readline` command. Syntax:

```
"readline{pattern}";
```

where `pattern` is line pattern similar to `scanf` but with the variables indicated by an .

Example(s):

```
PProjector *input = new PProjector();
input->AddInputASCII("omega.txt",
    "readline{@px @py @pz @mass}; [w]->SetXYZM(px,py,pz,mass)");
r.AddPrologueBulk(input);
```

Similar to the output, following `readline` commands read from the same file (e.g. when each particle appears in a single line):

```
PProjector *input = new PProjector();
input->AddInputASCII("multiple.txt",
    "readline{@p1x @p1y @p1z @p1mass};");
input->Do("readline{@p2x @p2y @p2z @p2mass};");
r.AddPrologueBulk(input);
```

This can be used to convert event files with muliple lines to TNtuples or ASCII files with different formats.

The following example merges two lines into one single line (in this case just an addition):

```
r.Input("multiple.lst", "readline{@a1,@a2,@a3};");
r.Do("readline{@b1,@b2,@b3};");
r.CloseFile();
r.Do("c1=a1+b1; c2=a2+b2; c3=a3+b3;");
r.Output("merged.lst","echo $c1,$c2,$c3");
```

3.5.2 Example: pushing particles from text file to the event loop

```
PReaction my_reaction("eta_decays");

PProjector *input = new PProjector();
input->AddCommand("myeta = P3M(0.,0.,0.,eta.mass)");
input->AddCommand("myeta->SetID(eta.pid)");
input->AddInputASCII("eta_sample.txt",
    "readline{@px @py @pz @mass}; myeta->SetXYZM(px,py,pz,mass)");
input->AddCommand("push(myeta)");
```

3.6 Evaluation

Histograms (or `TGraph(2D)`-objects) can be evaluated via the `Eval` method. The `Eval` method has 1, 2 or 3 arguments for 1-, 2-, or 3-dimensional histograms in axis coordinates (not bin number).

The return value is the content of the respective bin.

Example(s):

```
r->Do(histo1,"val = Eval(0.2)");
r->Do(histo2,"val = Eval(0.2, 0.4)");
```

In case of a 1-dimensional histogram the argument can be replaced by the content of `_x`:

```
r->Do(histo,"_x = 0.2; val = Eval()");
```

NB: If `Eval` is used, the entire `Do`-Line is set to “read-only”, i.e. the histogram is not filled in this access.

3.7 Random numbers

The `GetRandom` method can be used to sample random numbers from a histogram. Arguments (1, 2 or 3, depending on the dimension of the histogram) are the variables which are filled. The variables must have been instanced before.

Example(s):

```
r->Do(histo1,"x = 0; GetRandom(x)");
r->Do(histo2,"x = 0; y = 0; GetRandom(x, y)");
```

In case of a 1-dimensional histogram the argument can be replaced by a return value:

```
r->Do(histo,"x = GetRandom()");
```

NB: If `GetRandom` is used, the entire `Do`-Line is set to "read-only", i.e. the histogram is not filled in this access.

The `GetRandomX`/`GetRandomY` method can be used to sample random numbers from a single line (in one direction) of a histrogram, e.g.:

```
r->Do(histo1,"x = 0; GetRandomY(x)");
```