REST v2.2 Introduction

Introduction

REST is an event-oriented analysis framework. Based on ROOT, it defines specific classes to keep data, read config, and run processes. Most of these classes are designed for gas detectors with micropatterned readout.

REST does analysis through some modular process classes. They in chain convertes raw data to the final analysis result. One can easily modify the process chain in the config file and do his own analysis. We will frequently introduce new or update the old processes as the analysis work of PandaX-III requires. The user can also write his own processes at any time.

(we need to add some development history here)

We made many changes since then, and up to now REST is still under development. The latest version is V2.2, which is the basis of this guide.

Installing REST

We first start with the installation of REST. First check the environment. Following are some necessary development tools on linux.

- unzip
- cmake
- g++
- gcc
- python
- git

- subversion

Most of the Linux distributions should already contain these softwares. One can check if they are installed by typing their name directly in bash(for subversion type "svn"). If not installed, use `sudo apt-get install ...` or `sudo yum install ...` to install them.

In addition, we need to install the following packages referred to by REST.

- ROOT6 or higher
- tinyxml

REST's gas functionality also needs a library "garfield" (compiled with ROOT6). This can be switched on/off by setting cmake flags.

We have some shell scripts to install these packages. One can find them in ./scripts/installation.

After finishing installing these packages, we are able to install REST mainbody. There are some python scripts in the directory ./scripts. The script "scriptsInterface.py" provides a wizard for installation (requires package "python-tk"). Call it by typing `python scriptsInterface.py` .

One can also choose to manually install REST, by using `cmake` followed by `make install` in the root directory of REST. REST will be installed in the directory ./install by default. Cmake for REST can have those following flags: -DINSTALL_PREFIX=(install path); -DREST_GARFIELD=(ON/OFF); -DREST_WELCOME=(ON/OFF)

Finally the user needs to source the shell script before using REST software. The shell script is in the installation path named "thisREST.sh". It is recommended to add a line "source .../thisREST.sh" in users .bashrc file.

Check if REST is successfully installed by typing `rest-config --welcome` in the command line. A welcome message with version info should show up.

Try Some Examples

The main executable of REST is restManager and restRoot. By typing directly `restManager` it will show its usage. By typing restRoot the user can access to REST libraries and macros inside ROOT prompt. We have some example files for restManager in the directory ./examples/restManager. We first switch to that directory.

Generate a readout file

We first try to generate a readout file. A readout file saves the definition of the detector's readout system, including geometry, daq channel mapping, strip gain, etc. This file very important for the processes. Generate it by typing :

`restManager --c generateReadoutFile.rml --o readouts.root`

This command will make REST to save readout definition to a file "readouts.root", which can be used in the later data analysis work. The readout definition is all in the rml file. Later we will explain what it did.

A readout figure will showup after the generation. The user can also view the figure later on by typing `restViewReadout readouts.root PandaReadout_MxM` . Here "PandaReadout_MxM" is the name of the readout definition.

Process a raw data file

If you are in pandax-iii daq server, you can try to process some raw data file with the readout file we just generated. For example, to process a 7MM run data file, we can type the command:

`restManager --c multiCoboAnalysis.rml --i /data2/7MM/graw/CoBo_AsAd0_2017-12-23T17\:24\:04.657_0000.graw --o abc.root`

This gives REST names of the config file, input file and output file. If works smoothly, you can see numbers of processed events increasing.

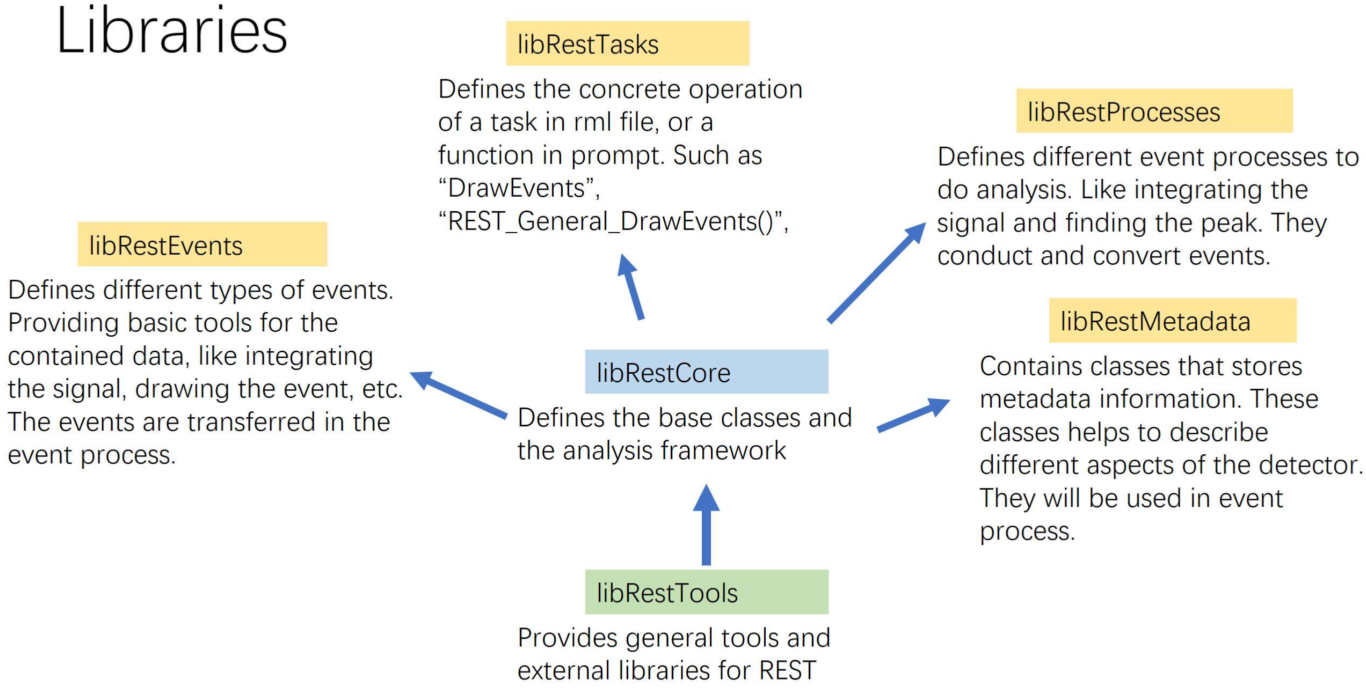
When it is done, use command: `restRoot abc.root` to see the content of generated ROOT file. The file should contain two trees and some saved classes. We will talk it later.

The REST Framework

REST is developed based on ROOT. It provides additional classes for the data analysis. The property of REST's analysis lies in its the event-orientation. An event is the collection of all physical existence due to one incident particle. Events are independent from each other, while their content is from a same random process in a same run. In REST we define several types of events, such as RawSignalEvent, SignalEvent, HitEvent and TrackEvent. We also define process classes to deal with these events, to make analysis for them or to change types between them. Some configuration and definitions are needed to make a process, so we define metadata classes to read config file with specific format, or to save/load the object into/from a ROOT file.

REST is organized with several libraries. The center of them is libRestCore. Inside this library we have several base classes which builds up the whole framework of REST. libRestCore relies on the library libRestTools, which provide some basic tools(string tools, output tools, units tools) and some external codes. Up on libRestCore, other REST libraries such as libRestProcesses, libRestMetadata, libRestEvents and libRestTasks are built.

Libraries



Here we will talk about some important classes in libRestCore. The methods in these class will only get their names mentioned. One can find their detailed usage in REST class reference.

TRestMetadata : basic functionality of REST

We first talk about the most important class - TRestMetadata. This abstract class brings basic funcitionality to REST, and is the base class of many REST classes. Data member of a TRestMetadata-inherted class is defined inside an rml file, an xml encoded configuration file. With this configuration file the inherted class can be initialized. The concrete operation of its initialization is defined in the method InitFromConfigFile() by the inherted class. This method is called during start up of the class by another method LoadConfigFromFile(), which parses the rml file or section as preparation.

TRestMetadata itself is inherted from the class TNamed. It allows the save/load functionality into/from a ROOT file. In many cases loading from ROOT file will be much quicker than reading and parsing an rml file.

TRestMetadata inherted class should have a name and a title. They are from TNamed class. In addition, we also define two basic attritubes: verbose level and storage. They controls the amount of words printed on screen and whether the class should be saved. All those will automatically be set from the rml config file.

One major type of the inherted class is called "metadata". They contain data of, for example, the geometry of a simulation, the properties of a gas, the readout pattern used to "pixelize" data, etc. Usually we will first instantiate and save a metadata class with an rml file. In pratical use, we can just read it from the saved ROOT file.

Another family of TRestMetadata inherted class is called "application". Their rml file gives, for example, the parameters of an analysis, the targets of a plot, the processes to load of an analysis, etc. Currently, REST cannot load application classes from saved ROOT file.

TRestMetadata also provides some utilities for the inherted class. The most commonly used methods are: GetParameter(), GetElement(), GetChar(), GetDataMemberWithName(), etc. It also defines leveled string output tools: fout, essential, info, debug, etc. See them in the REST class reference page.

TRestManager : managing REST applications

This class, as its name suggests, manages all other REST applications, including TRestRun, TRestProcessRunner, and so on. A REST application has a poniter to its manager, thus it can easily get access to its sibling applications. For example, a process class can get access to its sibling TRestRun, and acquires matadata in it.

TRestManager performs initialization for its managed applications by a strategy called sequential start up. For example, inside the rml configuration file, there is a section declared as "TRestManager". And in this section there is some child sections declared by different application names(here we have "TRestRun", "TRestProcessRunner"). TRestManager will try to instantiate objects of corresponding applications by calling the method TClass::GetClass(). Then it call the applications' LoadConfigFromFile() method giving them the defined child sections. If the application also contains TRestMetadata-inherted class which can be initialized through rml file/sections, its section will have its own child section(here we have "TRestReadout"). And this grandchild section is given to the grand-resident class in that application. This is sequential startup.

```
<TRestManager ... >
  <TRestRun ... >
    <TRestReadout ... >
      ...
    </TRestReadout>
  </TRestRun>
  <TRestProcessRunner ...>
```

```
  ...
</TRestProcessRunner>
<addTask .../>
<globals>
  ...
</globals>
</TRestManager>
```

In short, we perform sequential startup by constructing a same hierarchy in rml file with REST classes. This helps to make the code and conifig file easier to read.

An xml section declared as "globals" is also in the TRestManager section, the content of it will be expanded into all other sections in the same level. They will not override the one which are already defined.

There is also an xml section declared as "addTask". This line actually tells TRestManager the real work with those initialized applications. "addTask" section can either call a TRestTask type application which has a default behavior after initialization(talked later), or be a C++ style command for TRestManager to execute. For example, we can use:

`<addTask command="TemplateEventProcess->RunProcess()" value="ON"/>`,

and TRestManager will invoke the method "RunProcess()" in the application named "TemplateEventProcess". This application should be defined in previous sections.

TRestRun : operating files and managing data

TRestRun is an application class hosted by TRestManager. This class is designed for file operation and data handling. Usually, when REST is running, a instantiated TRestRun object opens a saved ROOT file to get the metadata and eventdata. This object therefore serves as a data provider in the framework. We call its public methods GetNextEvent(), GetEntry(), ImportMetadata(), GetMetadata(), etc. to get data from it. When the framework finishes its run, TRestRun will also help to save those data in files. We use it with the methods FormFormat(), MergeProcessFile(), CloseFile(), FormOutputFile(), etc.

For example, we have a TRestRun object containing some readout definition. During the event process when we need to use that readout definition, we first get the TRestRun object from the manager of the process class, then we can use its method GetMetadata(). We give input the class name(TRestReadout), and will receive the corresponding object.

TRestEvent & TRestEventProcess : data and analysis

TRestEventProcess is another important application class. It is a base class for all REST pre-defined processes. TRestEvent is directly inherted from TObject. It is a base class for all REST pre-defined event types.

Besides the functionality of reading configuration from file, TRestEventProcess defines extra interfaces for its inherted class to do the job. The method InitProcess() and EndProcess() are used as preparation/completion steps before/after the process loop. The method ProcessEvent() is the main method of the loop. It receives an input TRestEvent and returns a new output TRestEvent. The input and output events are in concrete type inherted from TRestEvent.

TRestEventProcess can also output analysis result to a tree. This kind of analysis result is called "observable". For example, TRestRawSignalAnalysisProcess will add a branch in the output tree called "BaseLineSigmaMean". This observable is the value of average baseline RMS of the input TRestRawSignalEvent. By switching on/off the observable in the config file, user can change the output of this process. Observbles can only be double type, and the user must call the method TRestAnalysisTree::SetObservableValue() during the process.

A special kind of TRestEventProcess is called ExternalProcess. They usually open a/several raw data files (e.g. binary signal readout file) and directly generate output event. Their input event is null and they cannot save any observables in the tree.

REST uses chained process to do anslysis. For exapmle, In a practical detection run we get the raw readout waveform. The data file is first converted into type TRestRawSignalEvent by some processes. Then we go forward with the envolution of event type. The defined process TRestSignalZeroSuppresionProcess converts the type TRestRawSignalEvent into TRestSignalEvent type (cutting out the baseline and extracting the pulse). We redord the observables(baseline level, rms, etc) during the process. To go further we also have TRestSignalToHitProcess, which maps the channel id with their physical location according to an external readout file. The signal will be convered into 3D hit points. Finally we are able to draw a hit map or an energy spectrum with spacial cut.

In this way, an event is conducted and analized by a chain of processes. It is conducted with the changing representation of TRestEvent. During the process, we extract its information, mix our definition, and change its representation into the very processed type. We could finally get a close reconstruction to the event's physical truth.

TRestProcessRunner : running analysis in an efficient way

All TRestEventProcess objects are managed by TRestProcessRunner, which enables multi-threading, chain validation, output handling, etc. This application class makes several copies of the process chain and keeps each of them in a thread. Any number of threads are supported with siginificant improvement of process efficiency. During the event process, the thread first asks TRestProcessRunner for an input event, which in term asks its sibling TRestRun object for the next event from input file. After processing, the thread will ask TRestProcessRunner to make a save for its output event. The runner copies the thread's tree's branch address to its own tree, and call TTree::Fill() afterwards. However, REST deals with data files with usually very large size, so heavy IO stress are exerted on disk. As a result, too much threads will not be helpful to the higher efficiency.

TRestProcessRunner is able to save a snapshot of the values of the data members for each managed TRestEventProcess objects in a corresponding branch. This is regarded as an alternative of saving observables as analysis result. The user can switch off the saving by using an annotation like `//!` at the end of the data member definition. This saving supports not only double type, but also int, vector, map, etc., which makes the analysis more flexible.

We also implemented a test run functionality in the runner. It tries to give an input event to the process chain and receives the output event. It will then know the memory address of the output event and therefor be able to config the output tree. In old times we must instantiate a TRestEvent object for the output of an event process, and copy the data from input event(if the process dosen't change the event type). Now we can simply copy the address of the input event and directly operate it. This will also improve the efficiency and simplify the code.

Using REST

REST in all provides two main executables, several ROOT scripts, several alias calling the scripts, plus a shell script containing REST system infomation.

`restManager` is the main program of REST. It runs with the first argument specifying the rml config file or ROOT script name. And with the following arguments giving some parameters. In rml config mode, the usage is like:

`restManager --c CONFIG_FILE [--i INPUT_FILE] [--o OUTPUT_FILE] [--j THREADS] [--e EVENTS_TO_PROCESS] [--v VERBOSELEVEL]`.

Here we must give the rml config file to the program. Other arguments with squared brackets are optional. If given, they will overwrite the corresponding parameters in rml config file. "restManager" calls TRestManager object to parse rml config file and handle the objects and tasks defined in the xml section.

In scripts executing mode, the usage is: `restManager TASK_NAME ARG1 ARG2 ARG3`. User just needs to specify the script name and REST will automatically find and run it from the "macro" directory in installation path. Alias are set together with those ROOT scripts. The commands `restXXX` is actually an alias of the command "restManager XXX", which also executes a script. The usage is like:

`restViewEvents abc.root TRestRawSignalEvent` or `restManager ViewEvents abc.root TRestRawSignalEvent`

Several pre-defined ROOT scripts are already installed there. We have a list of them in the appendix. The arguments are auto detected from the definition of the C++ function in script. REST also provides a default help message for the function is wrong number of input argumets are given.

The program "restManager" runs once and quits after finish. On the other hand, `restRoot` will not quit but provide a prompt after finish. "restRoot" is identically ROOT with additional REST libraries and ROOT scripts loaded. As a result, it can not only operate REST objects or data trees saved in TFile, but also run the methods in pre-defined ROOT scripts like "REST_Printer_SignalEvent()". For example, in prompt, calling `TASK_NAME(ARG1,ARG2,ARG3)` will be equal to the previous call: `restManager TASK_NAME ARG1 ARG2 ARG3`.

Finally we have a shell script `rest-config`. It provides some basic infomation of REST, including installation date/directories, branch, commit id, compilation flags, etc.

Running with a ROOT script

REST defines some ROOT scripts which can both be used at bash, ROOT prompt, and rml file. In a script we define a C++ function for ROOT, and the application TRestTask can be instantiated from that script. TRestTask first calls gInterpreter to load the file, and then forms a ROOT command calling the function in file.

TRestTask provides basic functionalities of helping message and rml parsing for traditional ROOT scripts. If that's not enough, we can define TRestTask-inherted class ourselves inside the scripts, and override the helping or rml parsing methods in it.

When using `restRoot`, the program will load all the scripts and user can get access to all the defined C++ function in prompt. When using `restManager`, without rml file, REST will try to instantiate a TRestTask class either from REST library(if this TRestTask-inherted class is defined manually) or by loading the file. If it is with rml file, The instantiate TRestTask object will also perform an rml parsing afterwards. This make different calling resulting the same, as they both referrs to a same ROOT script. It also saves a lot of time for developer adapting these different usage.

For example, we can write a ROOT script named "REST_ViewEvents.hh" in the directory ./macros. Inside the script we define a C++ function "REST_Viewer_GenericEvents()" and a class "REST_ViewEvents". When we use the line `<addTask type="ViewEvents" filename="abc.root" value="ON"/>` in an rml file, TRestManager will try to instantiate a TRestTask-inherted class with class name "REST_ViewEvents". The class will set its datamember "filename" to the value "abc.root", and then do the task of showing an event. It calls the defined function "REST_Viewer_GenericEvents()" in file, giving the argument of the input file name. This is equivelent to the call REST_Viewer_GenericEvents("abc.root") inside restRoot prompt.

Rml reference

Though encoded in standard xml format, all REST config files use .rml extension to identify themselves. rml file makes a clear hierarchy of C++ objects used during analysis. This kind of configuration file not only tells the newcomer the job it is carrying, but also gives him a good view of the entire REST framework.

Firstly, the user needs to study a little xml stuff. We start with a template rml file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<A_ROOT_SECTION>
  <ClassName name="userGivenName" title="User given title" >
    <parameter name="parName" value="parVal" />    <SomeCommand field1="value1" field2="value2"/>
    <ContainedClassName field1="value1" field2="value2" ... >
      <SomeCommand field1="value1" field2="value2" />
      comments or words
    </ContainedClassName>
  </ClassName>
  <AnotherClass ...>
    ...
  </AnotherClass>
  <SomeCommand field1="value1" field2="value2"/>
</A_ROOT_SECTION>
```

The first line is universal, telling the text viewer this file is xml encoded. Then here comes a **section**. An xml section is a sealed text structure starting with `<decalre` and ending with `</decalre>` or `/>`. Sections in an xml file usually have multiple nesting relationship. Here as the indentation suggests, we have a root section decalred "A_ROOT_SECTION". It has three child sections "ClassName", "AnotherClass", "SomeCommand". In tinyxml xml sections are also called **element**, and the declaration are also called "value". In old version xml sections are also called **KeyDefinition** or **KeyStructure**.

Xml sections can also have some **field values** (also called xml **attributes**), like in the template `field1="value1"` or `name="userGivenName"`. In REST these attributes can also be written in a child section declared with **parameter**. In the template if we add an attribute `parName="parVal"` in the thrid line, this is equivalent to the forth line: `<parameter name="parName" value="parVal" />`.

The symbol `<`, `>`, `&`, `"` shall appear in the main text of xml encoded file. Use escape string `<`, `>`, `&`, `"` respectively.

In REST, the section decalration `include`, `for`, `variable` and `myParameter` and the symbols `{`, `}`, `[`, `]` are reserved for the software. They will be preprocessed by the software before the method "InitFromConfigFile()" in matadata classes.

variable and myParameter

The xml sections "variable" and "myParameter" are for the keyword replacement. They are defined with a line like: `<variable name="PITCH" value="3" overwrite="false" />`. xml sections with same or lower hierarchy than this definition section will have its attributes replaced. If this definition section is in the "globals" section, then all xml sections in the file can see it.

To replace "variables", we must mark the corresponding keyword, while to replace "myParameter" we need not. Both the following symbols can mark the keyword: `{}`, `[]` an `${}`. For example, we add a line after the previous "variable" definition: `<myParameter name="pitch" value="{PITCH}" />`. This marks out the keyword "PITCH" and it will be replaced by the word "3". Now we defined a "myParameter" with name "pitch" and value "3". Then we add another line: `<addPixel id="0" origin="(pitch,pitch/4+pitch)" size="(20,20)" rotation="45" />`. In this line all the apperrance of "pitch" will be replced by the word "3". The expression will be executed.

REST works together with system environmental variable. By switching true or false for the "overwrite" attribute, a variable definition will use the text defined vale or the system environmental variable. By marking the keyword with `$ENV{}` REST will search for system environmental variable to replace it.

include definition

It is possible to link to other rml files in any section. The included file must also be xml encoded. REST will open the file and searches for the section with the same declaration(or type attribute) and name attribute as the current section. If found, the external section will be expanded into the current section. Variable in that file will be imported together.

There are two ways to make a include definition. One can either specify the external file in the element attribute: `<addProcess type="TRestRawSignalAnalysisProcess" name="sAna" value="ON" file="processes.rml"/>` or in the element's child:
`<addProcess type="TRestRawSignalAnalysisProcess" name="sAna" value="ON">`
` <include file = "processes.rml" />`
`</addProcess>`

These two include definitions will order REST to find a section in the file process.rml declared as "addProcess" or "TRestRawSignalAnalysisProcess" and named with "sAna". The section can both be a root element or a child element of root element(cannot be grand-child element). If found, REST will expand its attributes and child elements to the element "addProcess".

for loop expansion

The definition of FOR loop is implemented in RML in order to allow extense definitions, where many elements may need to be added to an existing array in our metadata structure. The use of FOR loops allows to introduce more versatil and extense definitions. Its implementation was fundamentally triggered by its use in the construction of complex, multi-channel generic readouts by TRestReadout.

The for loop definition is as follows, where *pitch* and *nChannels* are previously defined myParameters, and *nCh* and *nPix* are the *for* loop iteration variables.

```
<for variable = "nCh" from = "0" to = "nChannels-2" step = "1" >
  <readoutChannel id = "{nCh}" >
    <for variable = "nPix" from = "0" to = "nChannels-1" step = "1" >
      <addPixel id = "{nPix}" origin = "((1+{nCh})*pitch,pitch/4+{nPix}*pitch)" size = "(pixelSize,pixelSize)" rotation = "45" />
    </for>
    <addPixel id = "nChannels" origin = "({nCh}*pitch,pitch/4+(nChannels-1)*pitch+pitch/2)" size = "(pitch+pitch/2,pitch/2)" rotation = "0" />
  </readoutChannel>
</for>
```

REST will recongize the fields "variable", "from", "to", "step" in the header of the for loop definition. The variable "nCh", definded at the header of the for loop definition, will be updated in each loop and be used to replace values of the loop content. During the loop, REST will add the new content element at the front of the for loop element(add a new sibling). After the loop, REST will delete the for loop element, leaving purely the loop content.

an example

We used generateReadoutFile.rml in the ./example directory to generate a readout file. We now open it and see what it did.

```
... <TRestManager ...>
  <globals>...</globals>
  <TRestRun>...</TRestRun>
  <addTask .../>
  <addTask .../>
</TRestManager>
```

The root section is declared "TRestManager". It contains a scetion declared "globals", a scetion declared "TRestRun" and two sections declared "addTask". This section has a corresponding class in REST with same class name.

In the "TRestManager" section, obviously, the "globals" section are providing some global setting for others. The "TRestRun" section has also a corresponding class in REST. This class mainly deals with file IO and data transmission. So we are adding TRestRun class inside TRestManager class. Lets see what's in it.

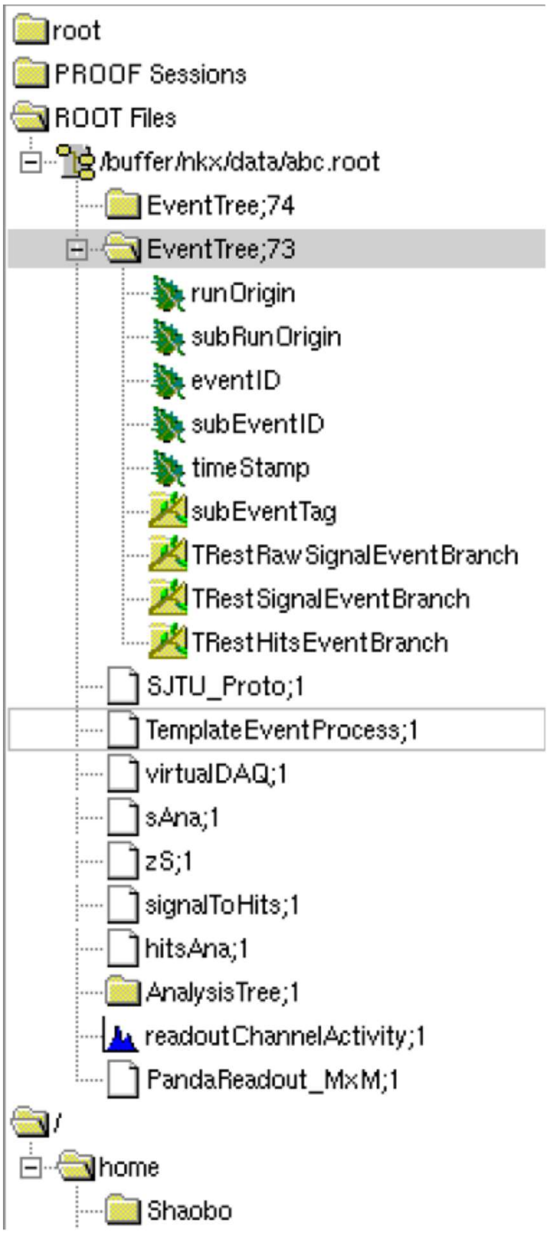
```
<TRestRun ...>
  <TRestReadout ...>
    <readoutModule .../>
    <readoutPlane ...>
      <addReadoutModule .../>
      <addReadoutModule .../>
      <addReadoutModule .../>
      ...
    </readoutPlane ...>
  </TRestReadout>
</TRestRun>
```

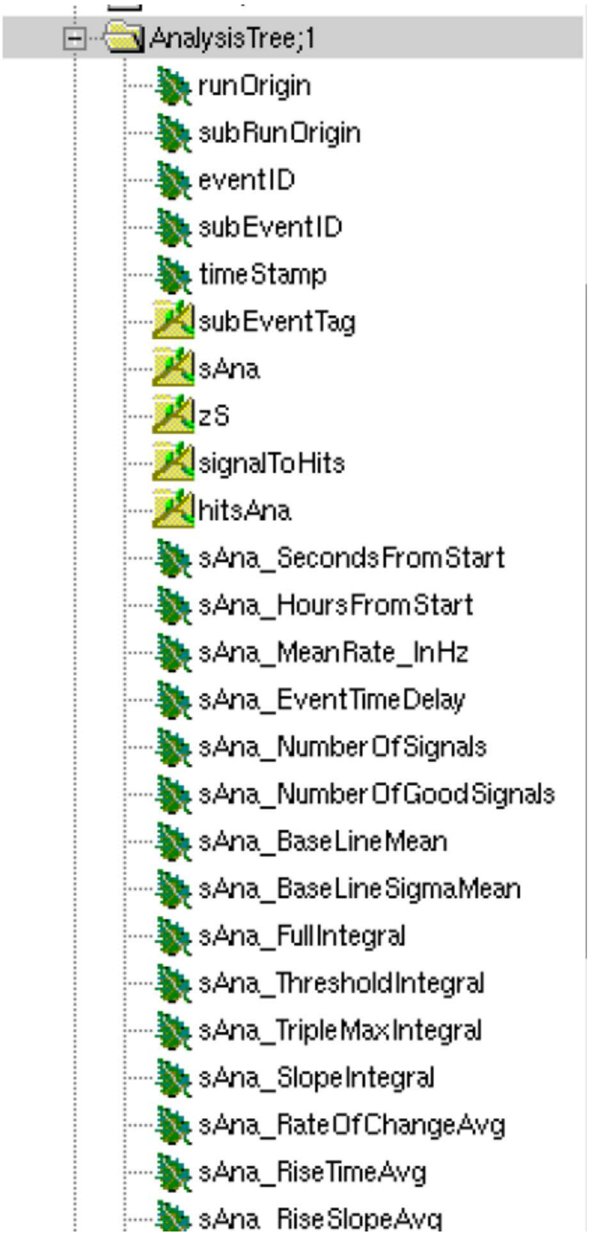
Its easy guess that we are adding a TRestReadout class inside TRestRun class. This class is what contains real readout definition. Inside its section, there are several operations. First we define a readout module, which in our experiment is the MicroMegas. And then we define a readoutplane, adding several of this kind of readout module in it, giving their physical position and some other infomation. All together they form an one-plane readout system. This TRestReadout class is now saved inside TRestRun class.

Previous work is done within the initialization of these classes. Now these classes are ready and we need to tell REST what to do. So finally, in "addTask" section, we give the command readoutrun->FormOutputFile(). and readoutrun->CloseFile(). Here "readoutrun" is the name(not class name) of the previously defind TRestRun class. Obviously here we are telling REST to "save file and close". The two lines of command are actually a method in the class TRestRun which TRestManager will invoke.

REST data format

REST saves an event tree, an analysis tree, some metadata objects, and some ROOT analysis objects in a same output file of the analysis run, as shown in the figure.





The trees are of the type TRestAnalysisTree, which is typically a ROOT tree plus a list of observables. Inside these trees there are six branches saving some basic information of the event, including run ID, event ID, event time, event tag, etc.

For the event tree, we also save the output event of each process in sequence. The event branches are C++ object branches with a same structure as the class definition. Here in the figure we can see three event branches are saved: A TRestRawSignalEventBranch, a TRestSignalEventBranch, and TRestHitsEventBranch.

For the analysis tree, we also save the observables as well as a snapshot of each processes. In the figure there are four process branches called "sAna", "zS", "signalToHits" and "hitsAna". If we open them, we can see some leaves inside corresponding to the processes' data member. Analysis tree is separated from the event tree for the convenience of drawing. Usually event tree contains large amount of data and makes it slow to draw some thing, which needs to go through all the entries of the tree.

Several metadata objects are also saved in file. They are used for recovering the setup of that analysis run. For example, REST can read the saved TRestReadout object (here named "PandaReadout_MxM") and get the setup of readout structure used at that time.

In addition, REST allows processes to save some ROOT analysis objects in the file. Here the TH1D "readoutChannelActivity" is saved by the process "sAna" (of type TRestRawSignalAnalysisProcess). We can directly draw it.

Analysis with REST

Here we will talk about the detailed options of rml config file when using REST. We usually have a "TRestRun" section, a "TRestProcessRunner" section, a "globals" section and some "addTask" section under the root section "TRestManager" in an rml file to be loaded by `restManager`. "TRestRun" section and "TRestProcessRunner" section corresponds to the two REST application objects managed by TRestManager. They cooperate with each other.

name, title and verbose level

TNamed class introduces name and title as datamembers for the objects of its inherted class. In addition, TRestMetadata introduces verbose level option. We need to set all three of them as basic information for any TRestMetadata inherted class objects. Use xml attributes or child sections "parameter" to set them. For example:

```
<TRestProcessRunner name="TemplateEventProcess" verboseLevel="info">    <parameter name="title" value="A Template of REST Analysis" />
  ...    </TRestProcessRunner>
```

There are five verbose levels one can choose, and the parameter "verboseLevel" can either be a number value or a string value.

level	number	string	Description
REST_Silent	0	silent	show minimized information of the software, as well as error messages
REST_Essential	1	essential	+show some essential information, as well as warnings
REST_Info	2	info	+show most of the infomation
REST_Debug	3	debug	+show the debug messages, pause at each processes to show the details
REST_Extreme	4	extreme	show everything

The default verbose level for a TRestMetadata inherted class is silent/info before/after loading the rml file(calling the method "LoadConfigFromFile()").

setting run information

Physically, a "run" is a continuous data taking during which all the detector's configuration maintains a constant. We can set run information in TRestRun and they can be saved in output file.

Below is a list of run info datamember in the class TRestRun. They can be set with a line like: <parameter name="experiment" value="PandaX-III"/> under the section "TRestRun".

item	type	rml parameter name	Description
fRunNumber	int	runNumber	first identificative number
fParentRunNumber	int	~~~~	~~~~
fRunClassName	TString	~~~~	~~~~
fRunType	TString	runType	Stores bit by bit the type of run. e.g. calibration, background, pedestal, simulation, datataking
fRunUser	TString	user	To identify the author it has created the run. It might be also a word describing the origin of the run (I.e. REST_Prototype, T-REX, etc)
fRunTag	TString	runTag	A tag to be written to the output file
fRunDescription	TString	runDescription	A word or sentence describing the run (I.e. Fe55 calibration, cosmics, etc)
fExperimentName	TString	experiment	The experiment name

adding metadata

REST metadata objects is managed by TRestRun. They provides, for example, gas definition or readout definition to the processes to use. So inside the "TRestRun" section we need to add metadata. It is possible to either import them from a ROOT file or to generate new.

To import them from file we need to use a line like:
<addMetadata name="PandaReadout_MxM" file="readouts.root"/> ,
where we input the name of the matadata object and the name of the file. This is a recommended way to add metadata as it is faster. One can spend some time generate definition files for the detector, then other users need not to do it again.

To generate a new instance, we need to add a full definition of the class. The section declaration must be the metadata class name. The content of the section should follow the rule of the class. For example:

```
<TRestRun ...>
  <TRestReadout ...>
    <readoutModule .../>
    <readoutPlane ...>
      <addReadoutModule .../>
      <addReadoutModule .../>
      <addReadoutModule .../>
      ...
    </readoutPlane ...>
  </TRestReadout>
</TRestRun>
```

adding process and its observables

Now we are going to add processes and define the needed output observables. In the section "TRestProcessRunner" we add sections like <addProcess type="TRestRawSignalAnalysisProcess" name="sAna" value="ON" file="processes.rml"/> . We have to specify the type and name. In addition, we can use the option "value" to switch on/off the process in the analysis chain.

To add observables or to set parameters, we need to write lines like: `<observable name="FirstX" value="ON" />` or `<parameter name="resolutionReference" value="1.0" />` inside the "addProcess" section (as a child section of it). In case there are too many observables and parameter to define, which may cause a mess in the rml file, we can use an include definition here. REST already defines some useful observable/parameter sets for its process. For example for TRestSmearingProcess we have two parameter sets defined in the file "processes.rml", their names are "smear_1FWHM" and "smear_3FWHM" respectively. Use an attribute like: `file="processes.rml"` to include and expand this rml file. To change the parameter set just change the "name" attribute.

input file and external process

The input file must be given to TRestRun to run an analysis. If the input file has .root extension, then it will be regarded as REST data file and will be directly opened. If not, we must have an external process in TRestRun to extract events from it. The "addProcess" section for external file processes can both be in section "TRestRun" and section "TRestProcessRunner". There can only be one external process added, and it is running under single threaded mode.

changing saved branches

A parameter in section "TRestProcessRunner" changes the branches to save in the two output trees. The parameter line goes like:
`<parameter name="treeBranches" value="inputevent:processevent:outputevent:inputanalysis:outputanalysis"/>`
where we spilt five items with the symbol ":". The five items are:

inputevent: the event from input file
processevent: the output event of each processes
outputevent: the output event of the last process
inputanalysis: the existing analysis(observables, process snapshots) in input file, if input file is a REST data file
outputanalysis: the observables and process snapshots

inputevent, processevents, outputevent may have overlaps. For example, the output events of two processes are the same when a process just pass the pointer of its input event to its output event. In this situation, the two events will not be save twice.

By default REST saves all the five items in trees. If the user wants to save some disk space, he can choose to save analysis items only. Or if he only wants a view of the last processed event, he can choose to save the output event only.

changing event region

Three parameters in section "TRestProcessRunner" changes the region of events to process. They are: "firstEntry", "lastEntry" and "eventsToProcess". If they are all zero, then the input file will have its all events processed. If the parameter "eventsToProcess" is non-zero, then the process will cut down after it reaches the number. The parameter "firstEntry" and "lastEntry" is only effective when the input file is a REST data file. They determines the entry region in the tree to extract events to process. "lastEntry" will be overwritten by a non-zero "eventsToProcess".

search path for definition files

Usually we put include rml files and input definition files in a same directory as the main rml file. This makes it simple to specify the file name(otherwise we need to input the absolute path of it). Actually we can also add additional paths in "globals" section for REST to find these files. The following is an example:

`<parameter name="addonFilePath" value="$ENV{REST_PATH}/inputData/definitions/:$ENV{REST_PATH}/inputData/gasFiles/"/>`

In the directoty "definitions" REST saves many pre-defined rml files for readout, processes and gases. In the directory "gasFiles" REST saves many pre-generated gase files. The detailed organization of the directory "inputdata" can be found in the appendix. [REST pre-definition data]

output file auto-naming

REST also enables auto-naming of the output file. By using square brackets in the "outputFile" parameter in TRestRun section, the user can have REST trying to replace the strings. For example, we can have a the parameter written like:

`<parameter name="outputFile" value="RUN[RunNumber]_[Time]_[LastProcess].root" />`

The replace work is done as follows:

1. replace with system environmental variable(done in rml parsing step)
2. replace with the file info
3. replace with the process info
4. replace with the run info

There are some public keywords for replacement in REST:

source	item	Description
file info	Time	The last write time of the input file
file info	Date	The last write date of the input file
file info	Size	The size of input file
file info	Entries	The total entries of input file (if is REST data file, otherwise = 2e9)
process info	FirstProcess	The name of first event process
process info	LastProcess	The name of last event process
process info	ProcNumber	The number of processes
run info	see the table above	~~~~

In future we may add more keywords. The user can also add his own file info keywords from the input file name. What he needs is to add another parameter line like:
`<parameter name="inputFormat" value="run[RunNumber]_cobo[CoBoId]_[Fragment].graw"/>`
Therefore REST will match the input file name with the given format. For example, when file name is: "run00042_cobo1_0000.graw", then item "RunNumber" with value "00042", item "CoBoId" with value "1", and item "Fragment" with value "0000" will be added into file info.

Let's assume that the graw file is created in 2018-01-30 16:30, and the last event process is a TRestRawSignalAnalysisProcess with a name "sAna", then the output file name will be: "RUN00042_16:42_sAna.root".

the "addTask" command

Don't forget to add an "addTask" section after all the sections are completed. We invoke the method RunProcess() in TRestProcessRunner to run the analysis. Here we need a line in "TRestManager" section like:
`<addTask command="TemplateEventProcess->RunProcess()" value="ON"/>`.
Here "TemplateEventProcess" is the name of the TRestProcessRunner object defined previously.

It is also possible to directly use a line:
`<addTask type="processEvents" value="ON" />`.
This asks TRestManager to directly call start of TRestProcessRunner object.

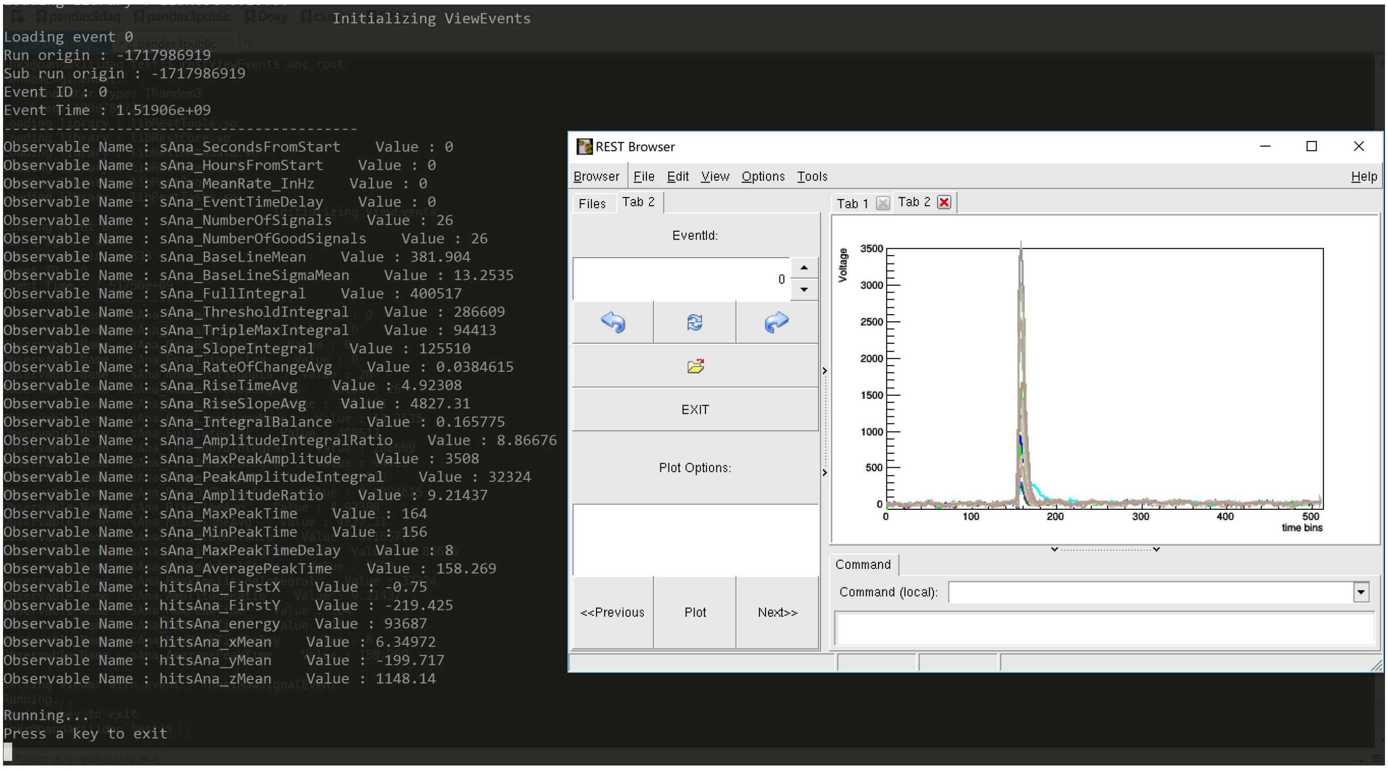
Browsing and viewing events

Events in REST data files are managed by TRestRun, whose graphical interface, in turn, is shown by by TRestBrowser. This class shows a TBrowser window during initialization. In the window there is a canvas showing the current event, and a control panel to switch between events.

In restRoot prompt, by using TRestBrowser, one can easily get accsess to the file's events, and don't need to manually instantiate a TRestEvent object and set the tree's branch address. He just needs to type:
`restRoot abc.root`
`TRestBrowser a`
`TRestxxxEvent*eve=(TRestxxxEvent*)a.GetInputEvent()`,
and will be free to operate this event.

By default TRestBrowser extracts the last event in file, and draws it in the canvas by using the viewer class TRestGenericEventViewer. This viewer just calls the default method TRestEvent::Draw(). Other viewers like TRestHitsEventViewer or TRestG4EventViewer are also available. Some pre-defined ROOT scripts can be used to draw these events in differently. The commands like: "restViewEvents abc.root", "restManager ViewHitsEvents hits.root", "REST_Viewer_LinearTrackEvent("track.root")" and "restManager --c Viewabc.rml" shall all work.

Here for example, we use the generated file in example, and call the command `restViewEvents abc.root`. The last event is TRestRawSignalEvent type in this file, and a TRestBrowser window will show up with some observable values on prompt.



In the right side it shows a combined plot of the event, which consists from many individual signals. In the left side we have a control panel which helps to switch next/previous/specific event/signal and open a new file. Different event viewers will define different interfaces of the control panel and the plot window.

Some viewer processes are also available in REST. The user can have a view of the events during the process. All the viewer processes are single thread only, and TRestProcessRunner will automatically roll back to single thread mode with a viewer process in process chain.

Plot the analysis result

It is also allowed to plot histograms for observables in output file. REST has an application class called TRestAnalysisPlot. It generates plot string according to an rml config file and calls the TTree::Draw() method to draw the histogram. It can also save the plots to a pdf file or ROOT file afterwards.

To use it, a "TRestAnalysisPlot" section is needed in "TRestManager" section. The template of rml config file for TRestAnalysisPlot can also be found in ./examples. It shall follow the rules below.

add input file and set plot mode

To add input files just use a section like: `<addFile name="filename.root" />` in the "TRestAnalysisPlot" section. Multiple input file is allowed. If the "addFile" section does not exist, TRestAnalysisPlot will ask the sibling TRestRun object for its output file as the input file.

In most cases REST saves a single output file in an analysis run. So these multiple files are from different runs, or analysis with different configurations. Usually we are interested in the difference of one observable between different runs. In this case we just set the plot mode to "compare" with section: `<parameter name="plotMode" value="compare" />`. Then REST will plot these same-observable-from-different-file in a same figure with different color. The "compare" plot mode is also the default plot mode.

In rare cases the multiple files are from a same run with same analysis configuration, then we need to set plot mode to "add". This will make REST plot the observables into a single histogram.

define a canvas

It is needed to define a canvas for TRestAnalysisPlot. Use a section like:
`<canvas size="(1000,800)" divide="(2,2)" save="plot.pdf" />`
to define it. The canvas can be devided into several sub-canvas and each of them will contain a plot figure. It can also be saved into a file. Most of the common figure formats are supported, as REST calls TCanvas::Print() method to make the save. The list of supported file formats shall be found in ROOT website.

add a plot with cut

Now we define and add a plot in "TRestAnalysis" plot section.

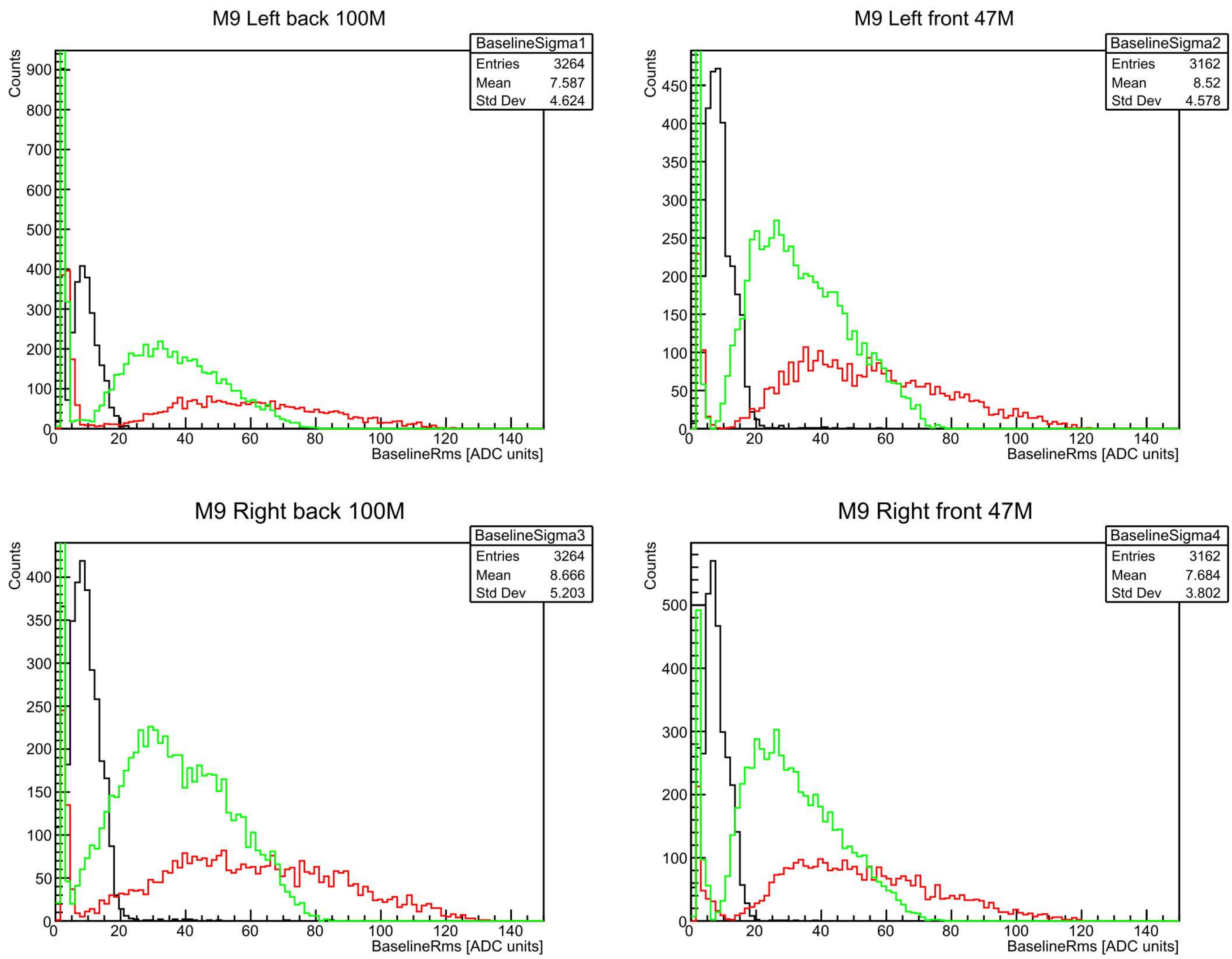
```
<plot name="Baseline" title="Baseline average" value="ON" >
  <parameter name="xlabel" value="BaselineRms [ADC units]" />
  <parameter name="ylabel" value="Counts" />
  <parameter name="logscale" value="false" />
  <parameter name="option" value="" />
  <source name="sAna_BaseLineSigmaMean" range="(0,1000)" nbins="100" />
  <cut source="sAna_NumberOfGoodSignals" condition="&gt;1" value="ON" /> </plot>
```

Here we can set xlabel, ylabel and logscale with corresponding parameters. We can also add additional options with parameter "option".

Then we add the source of the plot. Its name is "sAna_BaseLineSigmaMean", which is an observable in the analysis tree. Here we got a TH1 histogram of this observable. We can also define the region of this histogram, just as the template shows. TH2 and TH3 are also supported, by adding more lines of this kind of "source" section.

We also add cut for the plot. Here in the "plot" section we add a cut that the observable "sAna_NumberOfGoodSignals" should be greater than 1. Note that standard xml needs escape string to express the symbol `>`. Though this symbol still works out of some reason, we suggest using `>` instead of `>` for a good habit.

Finally with an "addTask" section in "TRestManager" section, we can make a plot like below



Reference of Metadata Classes

In this section we will talk about the usage of REST metadata classes. They load useful information about the experiment setup from rml config file. How to write those files is what the user should know.

TRestGas

The TRestGas metadata members description can be found detailed in the TRestGas class documentation. Here we provide few commands to show how to generate a new gas mixture and access its basic properties, as drift velocity, electron diffusion, and typical gas coefficients.

We start by definning a standalone TRestGas section inside our RML file. Our configuration file could be something like this.

(File : argonMixture.rml)

```
<TRestGas name="Argon-Isobutane 4Pct 10-10E3V/cm" title="Argon-Isobutane Mixture (4Pct Isobutane)">
  <parameter name="pressure" value="1" />
  <parameter name="temperature" value="293.15" />
  <parameter name="maxElectronEnergy" value="400" />
  <parameter name="W_value" value="26.145" />
  <parameter name="nCollisions" value="10" />
  <eField Emin="10" Emax="1000." nodes="20" />
```

```
  <gasComponent name="ar" fraction="0.96" />
  <gasComponent name="iC4H10" fraction="0.04" />
</TRestGas>
```

The *gasDataPath* defines the path where our gas file will be generated. In this case the file will be generated at the same location where we launch our program.

We can generate the gas file by specifying the full path to our configuration file and providing the name of the gas we want to use. In this case we will assume that we are launching the commands from the same directory where *argonMixture.rml* is found. We first start *ROOT* interpreter using *restRoot* to load all the REST libraries in the ROOT environment.

:~$ `restRoot`

[0] `TRestGas *gas = new TRestGas( "argonMixture.rml", "Argon-Isobutane 4Pct 10-10E3V/cm", true )`

This should start the generation of the gas file, the calculation will be running for a few hours. When it is finished a new .gas file will have been created at *gasDataPath*. The filename format defines in a unique way the parameters used in the TRestGas section. In this way, it is used by TRestGas to determine if the gas mixture we want to use is already existing as a .gas file.

If the gas file exists TRestGas will directly load the gas file. After the previous command is finished you can verify it by closing the session and starting again.

[1] `.q`

:~$ `restRoot`

[0] `TRestGas *gas = new TRestGas( "argonMixture.rml", "Argon-Isobutane 4Pct 10-10E3V/cm", true )`

This time the recently generated gas file will be loaded and no long calculation will be required.

We can now access the members of TRestGas to obtain the values for the drift velocity or the tranversal electron diffusion. The following command will print on screen the drift velocity.

[1] `cout << "The drift velocity at 100 V/cm is : " << gas->GetDriftVelocity( 100 ) << endl`

You can also find the information of the gas you are using by calling PrintMetadata().

[2] `gas->PrintMetadata()`

Although one must be careful with the definition range of the electric field and the pressure, we can retrieve the gas properties at different gas conditions. This code changes the value of the pressure to 10 atm and prints the drift velocity at that pressure value.

[3] `gas->SetPressure( 10 )`

[4] `cout << "The drift velocity at 100V/cm at 10 bar is : " << gas->GetDriftVelocity( 100 ) << endl`

Finally, we can quickly visualize the dependency of different gas properties as a function of the field. The following code will plot the drift velocity at 10 bar, for a field range between 1V/cm and 1000V/cm drawing 100 extrapolated points.

[5] `gas->PlotDriftVelocity( 1, 1000, 100 )`

Thats it. Other gas parameters and relevant information related to TRestGas can be found in the class documentation.

TRestReadout

We will address two different examples following basic readout topologies. You will find a variety of more complex examples at REST_v2/inputData/definitions/readouts.rml. More details about readout construction are available at the documentation of TRestReadout class. The class TRestMetadata describes detailed information on how to write RML files.

Example 1. A basic pixelated readout

In this example we generate a readout with a single readout plane, and one pixelated readout module placed inside. To achieve that each channel has a unique pixel definition.

```
// We define some environmental variables that we can later use as ${VARIABLE}
<globals>
  <variable name="PIX_SIZE" value="3" overwrite="true" />
  <variable name="CHANNELS" value="8" overwrite="true" />
</globals>

<TRestReadout name="pixelReadout" title="A basic pixel readout. ${CHANNELS}x${CHANNELS} channels. Pixel size : ${PIX_SIZE} mm" >

  // These parameters are later keywords inside the section
  // and will be substituted by their value.
  <myParameter name="nChannels" value="${CHANNELS}" />
  <myParameter name="pixelSize" value="${PIX_SIZE}" />
```

```
  // Mapping nodes is the number of nodes N, in a NxN grid.
  // This grid allows for faster channel/pixel finding algorithm.

  // If the mappingNodes value is 0. The value will be automatically assigned by REST.
  <parameter name="mappingNodes" value="0" />

  // This is just the module definition.
  <readoutModule name="pixelModule" size="(nChannels*pixelSize, nChannels*pixelSize)" tolerance="1.e-4" >

    // We use for loops to generate any number of channels given by the CHANNELS variable.
    // The loop variable must be placed between [] in order to be evaluated.
    <for variable="nChX" from="0" to="nChannels-1" step="1" />
      <for variable="nChY" from="0" to="nChannels-1" step="1" />

        // The readout channel id will be used to identify the channel and associate it to a daq id
        <readoutChannel id="[nChX]*nChannels+[nChY]" >
          // In this example we define one pixel per channel.
          // But we can define any number of pixels inside a channel
          <addPixel id="0" origin="([nChX]*pixelSize, [nChY]*pixelSize)" size="(${PIX_SIZE},${PIX_SIZE})" rotation="0" />
        </readoutChannel>

      </for>
    </for>

  </readoutModule>

  // The real readout implementation is done inside the readout plane.
  // The readout plane parameters define the active volume.
  <readoutPlane position="(0,0,-990)" units="mm" planeVector="(0,0,1)" chargeCollection="1" cathodePosition="(0,0,0)" units="mm" >

    // We can add any number of modules
    // name="pixelModule" is the name defined at the "readoutModule" section.
    // We define the module position inside the readout plane
    <addReadoutModule id="0" name="pixelModule"origin="(-nChannels*pixelSize/2,-nChannels*pixelSize/2)" rotation="0" />

  </readoutPlane>

</TRestReadout>
```

By using restRoot one can manually instantiate this TRestReadout object and save it. By using restManager to directly generate the ROOT file, one needs to add these sections in TRestRun and TRestManager section, and provide an "addTask" section.

Example 2. A multilayer stripped readout

In this example we define a stripped readout using single pixels with y-dimension much longer than x-dimension. We create two readout module definitions, one for each axis, and place each readout module at a different readout planes.

```
<globals>
  <variable name="PIX_SIZE" value="3" overwrite="true" />
  <variable name="CHANNELS" value="8" overwrite="true" />
</globals>

<TRestReadout name="strippedReadout" title="A basic pixel readout. ${CHANNELS}+${CHANNELS} channels. Pitch size : ${PIX_SIZE} mm" >
  <myParameter name="nChannels" value="${CHANNELS}" />
  <myParameter name="pixelSize" value="${PIX_SIZE}" />

  // In case of errors during the readout generation you might need to
  // define this value manually in the mapping nodes parameter.
  <parameter name="mappingNodes" value="nChannels" />

  // X-strips readout module definition
  <readoutModule name="stripsX" size="(nChannels*pixelSize, nChannels*pixelSize)" tolerance="1.e-4" >

    <for variable="nChX" from="0" to="nChannels-1" step="1" />
      <readoutChannel id="[nChX]" >
        <addPixel id="0" origin="([nChX]*pixelSize, 0)" size="(${PIX_SIZE},${PIX_SIZE}*nChannels)" rotation="0" />
      </readoutChannel>
    </for>

  </readoutModule>

  // Y-strips readout module definition
  <readoutModule name="stripsY" size="(nChannels*pixelSize, nChannels*pixelSize)" tolerance="1.e-4" >
```

```
<for variable="nChY" from="0" to="nChannels-1" step="1" />
  <readoutChannel id="[nChY]" >
    <addPixel id="0" origin="(0, [nChY]*pixelSize)" size="(${PIX_SIZE}*nChannels,${PIX_SIZE})" rotation="0" />
  </readoutChannel>
</for>
```

```
</readoutModule>
```

// We define a first readout plane

```
<readoutPlane position="(0,0,-990)" units="mm" planeVector="(0,0,1)" chargeCollection="1" cathodePosition="(0,0,0)" units="mm" >
```

// This readout plane includes the readout with the strips along Y-axis (X-position)

```
<addReadoutModule id="0" name="stripsX" origin="(-nChannels*pixelSize/2,-nChannels*pixelSize/2)" rotation="0" />
```

```
</readoutPlane>
```

// We define a second readout plane covering the same active volume.

```
<readoutPlane position="(0,0,-990)" units="mm" planeVector="(0,0,1)" chargeCollection="1" cathodePosition="(0,0,0)" units="mm" >
```

// This readout plane includes the readout with the strips along X-axis (Y-position)

```
<addReadoutModule id="0" name="stripsY" origin="(-nChannels*pixelSize/2,-nChannels*pixelSize/2)" rotation="0" />
```

```
</readoutPlane>
```

```
</TRestReadout>
```

For the moment, the process TRestHitsToSignalProcess is not able to process a multilayer readout plane, and/or charge collection sharing between different readout planes, covering the same active volume. Although few changes would be needed to adapt this process.

Readout generation and storage in a ROOT file

Here we assume the previous examples are defined in a file named *readouts.rml* and this file is found at the working directory.

The following code will instantiate the TRestReadout class using the pixelated and stripped definitions, and save them to a ROOT file.

// We start a ROOT session with REST libraries and scripts loaded by using restRoot
:~$ `restRoot`

// We give the filename and the readout names as arguments for the TRestReadout constructors
[0] `TRestReadout *pixRead = new TRestReadout( "readouts.rml", "pixelReadout");`

[1] `TRestReadout *stripRead = new TRestReadout( "readouts.rml", "strippedReadout");`

// We create a new ROOT file with "RECREATE" option or open an existing file with "UPDATE" option
[2] `TFile *f = new TFile( "readouts.root", "RECREATE" );`

[3] `pixRead->Write("pixel");`

[4] `stripRead->Write("strip");`

[5] `f->Close();`

// We exit from ROOT session
[6] `.q`

After executing this code we will have a *readouts.root* file with two different readouts, named *pixel* and *strip*.

The original readout name given at the RML file has been lost. And in order to reference it in ROOT or REST we will use the names given at write time, *pixel* and *strip*.

Recovering the TRestReadout saved on a ROOT file

We can easily recover the TRestReadout as any other ROOT structure. In order to quickly look inside a REST/ROOT file we can use the executable **restPrintFileContents** to check the existing objects (readouts) inside the file.

:~$ `restPrintFileContents readouts.root`

The following code recovers the TRestReadout structure

:~$ `restRoot`

[0] `TFile *f = new TFile( "readouts.root" );`

// We get a pointer to the pixelated readout
[1] `TRestReadout *r = f->Get("pixel");`

// We print the metadata information of this readout
[2] `r->PrintMetadata();`

// And we print it again with full detail, with info about channels and pixels positions.
[3] `r->PrintMetadata(1);`

[4] `.q`

Readout visualization

The readout visualization is still far from optimal, but a couple of ways are available in order to verify the task of readout design.

In a ROOT session we can call the method TRestReadoutPlane::GetReadoutHistogram to draw the pixel boundaries.

~$ `restRoot`

[0] `TFile *f = new TFile( "readouts.root" );`

// We get a pointer to the pixelated readout
[1] `TRestReadout *r = f->Get("strip");`

// We draw first the readout plane 0. A canvas inside ROOT is automatically generated
[2] `r->GetReadoutPlane(0)->GetReadoutHistogram()->Draw();`

// We can draw the other strips readout plane on top of the existing drawing
[3] `r->GetReadoutPlane(1)->GetReadoutHistogram()->Draw("same");`

[4] `.q`

Or, we can directly use the script *REST_Viewer_Readout* to draw one of the readout planes.

~$ `restRoot`

// By default the plane with index 0 will be drawn, if not specified
[0] `REST_Viewer_Readout( "readouts.root", "strip" );`

// We can also draw the other readut plane
[1] `REST_Viewer_Readout( "readouts.root", "strip", 1 );`

Readout validation

The construction of complex readouts requires to evaluate the proper channel spatial definition. Complex readouts will be composed of channels in which several pixels are combined and overlapped. The overlap between different pixels on the same readout channel will never suppose a problem. However, different channels overlap may affect the final response of the readout channels.

In order to test the readout we can produce a random virtual hit generation, with (x,y) coordinates inside the range of the readout modules in a given readout plane. We may then activate few test channels and draw only those hits which dropped in the activated channels. The script *REST_UTILS_CheckReadout* allows to perform this task. To produce a faster result we can focus in a small area of the readout, defined by the *region* parameter. We can activate the 128 first channels with a channel *mask* definition.

The following code shows the use of this script that works for any TRestReadout class stored previously in a ROOT file.

$~ `restRoot`

// We define the 128 bits mask to enable different channels
[0] `Int_t mask[4];`

[1] `mask[0] = 0x80000100;` // Channels 8 and 31 enabled

[2] `mask[1] = 0x000000FF;` // Channels from 32 to 47 enabled

[3] `mask[2] = 0x0;` // All channels disabled [From 64 to 95]

[4] `mask[3] = 0x0;` // All channels disabled [From 96 to 127]

// We define also a reduced region of the readout where we will launch random (x,y)
[5] `Double_t region[4];`

[6] `Double_t region[0] = 0.2;` // Xmin starts at 20% of full area

[7] `Double_t region[1] = 0.8;` // Xmax ends at 80% of full area

[8] `Double_t region[2] = 0.4;` // Ymin starts at 40% of full area

[9] `Double_t region[2] = 0.9;` // Ymax ends at 90% of full area

// The last two arguments, N and pId are optional.

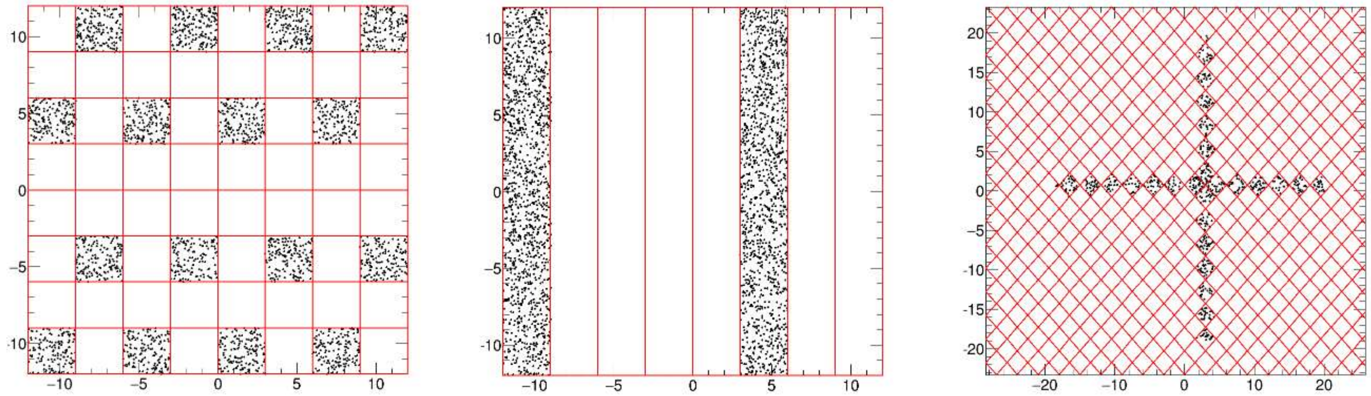
// The number of (x,y) coordinates to be generated.
Int_t N = 1E4;

// The plane readout index to be checked
Int_t pId = 0;

// This script will launch the generation of random (x,y) positions at the specified region,
// and it will draw only the hits on the activated channels (8, 31, 32-47).

[10] `REST_Tools_CheckReadout( "readouts.root", "pixel", region, mask, N, pId )`

These figures show the result of running the *REST_Tools_CheckReadout* script for different channels.



TRestG4Metadata

I don't know...

Start Your Own Analysis with REST

We may engage a new analysis request which REST is unable to handle. It is possible to modify/add processes to REST source code, or to develop a new project based on REST. The project can either be a program running separately, or a library added into REST libraries.

Adding a class to souce code

REST auto detectes source files in its directories. During the making, Cmake makes use of all the .h and .cxx files in the sub inc/src directories of each working directory. In concrete, the regular form of directory order is:

```
DIR
 ├──── CMakeLists.txt
 ├──── SUB-DIR-1
 │      ├──── inc
 │      │      └──── CLASS_A.h
 │      └──── src
 │             └──── CLASS_A.cxx
 └──── SUB-DIR-2
        ├──── inc
        │      ├──── CLASS_B.h
        │      └──── CLASS_C.h
        └──── src
               ├──── CLASS_B.cxx
               └──── CLASS_C.cxx
Or:
DIR
 ├──── CMakeLists.txt
 ├──── inc
 │      ├──── CLASS_A.h
 │      └──── CLASS_B.h
 └──── src
        ├──── CLASS_A.cxx
        └──── CLASS_B.cxx
```

Cmake first targets on a .cxx file. It regards the file name as the class name. Then it searches the .h file of this class. If found, it calls CINT to make a wrapper for the class with this .h file. ROOT CINT genertes a new .cxx file for the .h file, containing some streamer methods and reflection methods which overwrites those in TObject class. Each .h file can only contain one TObject inherted class whose name must be the same as the file name. Then Cmake includes that .h file and complies the two .cxx files calling gcc. This work is done for all .cxx files, after which CMake generates a library with the name of the directory.

As for the user, after adding a new class, he just needs to rerun the command `cmake [PATH]` followed by `make install` in the build directory.

To add a new process, we suggest copying the header file and the source file of our template dummy process. They are in the directory ./packages/userRESTLibrary, named "mySignalProcess". First rename the files and replace all the instance of "mySignalProcess" into your process name. Then add your data member and implement your InitFromConfigFile() method. It will convenient to use GetParameter() and StringToDouble() methods to set configurations from rml file. Then define your input and output event type, by using code: `fInputEvent = new TRestxxxEvent();` in method Initialize(). Finally implement your ProcessEvent() method which is for the main analysis loop. A new process is ready!

Adding new event class or metadata class shall be similar.

Referring to REST library in another project

In the directory ./packages/userRESTLibrary, we have a library project which can directly be compiled. We did the same modification as above, creating a new utilized process and metadata class. Then we can use the following command:

~/REST_v2$ `cd packages/userRESTLibrary`

// We create the build directory and enter it
~/REST_v2/packages/userRESTLibrary$ `mkdir build`
~/REST_v2/packages/userRESTLibrary$ `cd build`

// We create the compilation environment, setting the library to be installed at the same example directory.
~/REST_v2/packages/userRESTLibrary/build$ `cmake -DINSTALL_PREFIX=../ ..`

// We compile and install the library
~/REST_v2/packages/userRESTLibrary/build$ `make install`

After following those steps, we should have a *lib* directory inside the *packages* directory, and inside a library named *libMyRESTLibrary.so*. In order to use the generated library the system needs to be able to find it looking at LD_LIBRARY_PATH.

Add the following line to your *.bashrc* file and launch a new terminal.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/REST_v2/packages/userRESTLibrary/lib
```

Now REST will always start up referring to this new library.

Example : restG4

To make a separate program based on REST we also have an example: restG4. It is in the directory ./packages/restG4. It provides an executable named restG4 which refers to both REST and geant4 library.

To install it use the similar command as above. By default restG4 is installed in REST executable directory (${REST_PATH}/bin/).

Appendix

List of REST processes

We list here the different event data processes implemented in REST together with a brief funcionality description.

Data transformation processes

These processes are in charged of transforming data between different basic data types in REST.

As soon as we transform one data type to another we can make use of the dedicated data type processes. For example, if we have a geant4 event (TRestG4Event), we may transform it to a basic hits event (TRestHitsEvent) and continue processing the event using the basic *hit processes*.

REST process	Input type	Output type	Description
TRestRawSignalToSignalProcess	TRestRawSignal	TRestSignalEvent	Transforms a rawsignal into a signal event.
TRestSignalToHitsProcess	TRestSignalEvent	TRestHitsEvent	Converts a signal event into a hits event using TRestReadout.
TRestHitsToSignalProcess	TRestHitsEvent	TRestSignalEvent	Transforms a HitsEvent into SignalEvent using TRestReadout.
TRestHitsToTrackProcess	TRestHitsEvent	TRestTrackEvent	Hit clusterization into tracks by proximity (Accurate).
TRestFastHitsToTrackProcess	TRestHitsEvent	TRestTrackEvent	Hit clusterization into tracks by proximity (Aproximation).
TRestG4toHitsProcess	TRestG4Event	TRestHitsEvent	Transforms a geant4 event into a hits event.

Analysis processes

These are pure analysis processes. They do not transform the event data itself but add new observables/branches to TRestAnalysisTree. They may apply cuts to the event data, somehow deciding if an event should continue being processed. Any process can decide to stop processing an event by just returning a NULL pointer. This should be documented in each process class.

REST process	Input type	Output type	Description
TRestRawSignalAnalysisProcess	TRestRawSignal	TRestRawSignal	Adds analysis observables from raw signal event.
TRestHitsAnalysisProcess	TRestHitsEvent	TRestHitsEvent	Adds analysis observables from hits event.
TRestGeant4AnalysisProcess	TRestG4Event	TRestG4Event	Adds analysis observables from a geant4 event.
TRestTrackAnalysisProcess	TRestTrackEvent	TRestTrackEvent	Adds analysis observables from a track event.
TRestTriggerAnalysisProcess	TRestSignalEvent	TRestSignalEvent	Applies cuts using time window and energy threshold trigger definition.
TRestFindG4BlobAnalysisProcess	TRestG4Event	TRestG4Event	Finds the electron end blobs in a TRestG4Event. For events with at least 2-electron tracks.

Signal processes

These processes just modify the data inside a signal event, returning again a signal event data type. These kind of processes add signal noise to simulated data, filter noise from rawdata, shape the input signal, or suppress signal data points which are under threshold.

REST process	Input type	Output type	Description
TRestAddSignalNoiseProcess	TRestSignalEvent	TRestSignalEvent	Adds random noise to a signal event.
TRestSignalDeconvolutionProcess	TRestSignalEvent	TRestSignalEvent	Deconvolutes a signal using a given input response signal.
TRestSignalGaussianConvolutionProcess	TRestSignalEvent	TRestSignalEvent	Convolutes the input signal with a gaussian.
TRestSignalShapingProcess	TRestSignalEvent	TRestSignalEvent	Shapes the input signal with a given input response signal.
TRestFindResponseSignalProcess	TRestSignalEvent	TRestSignalEvent	Selects clean signals from input to be used as response for deconvolution.
TRestSignalZeroSuppresionProcess	TRestRawSignalEvent	TRestSignalEvent	Keeps only points which are found over threshold.

Hit processes

These processes just modify the data inside a hits event, returning again a hits event data type. These kind of processes normalize hits energy, fiducialize hits in a given volume, drift and diffuse hits, or reduce the hit number using merging algorithms.

REST process	Input type	Output type	Description
TRestElectronDiffusionProcess	TRestHitsEvent	TRestHitsEvent	Spatially diffuses input hits using gas properties defined in TRestGas and the active TPC volume/geometry defined in TRestReadout.
TRestFiducializationProcess	TRestHitsEvent	TRestHitsEvent	Only hits inside readout active volume definition survive.
TRestAvalancheProcess	TRestHitsEvent	TRestHitsEvent	Statistical gain increase per hit.
TRestHitsNormalizationProcess	TRestHitsEvent	TRestHitsEvent	Re-scales the hits energy by a constant factor.
TRestHitsRotateAndTraslateProcess	TRestHitsEvent	TRestHitsEvent	Rotates and translates a hit distribution along its energy center.
TRestHitsReductionProcess	TRestHitsEvent	TRestHitsEvent	Merges hits by proximity reducing the effective number of hits.
TRestHitsShuffleProcess	TRestHitsEvent	TRestHitsEvent	Hits are disordered in the hits queue.
TRestSmearingProcess	TRestHitsEvent	TRestHitsEvent	Hits energy is re-scaled with a random factor by a given resolution.

Track processes

These processes operate over track event data, or a set of hits that have been grouped into tracks. We may find the minimum path between the hits inside each track, reduce the number of hits inside each track, find the track ends, or project the hits over the main trajectory effectively linearizing the track.

REST process	Input type	Output type	Description
TRestFindTrackBlobsProcess	TRestTrackEvent	TRestTrackEvent	Finds the track end blobs in a TrackEvent .Tracks should have been pre-processed with path minimization and reconnection processes.
TRestTrackLinearizationProcess	TRestTrackEvent	TRestLinearTrackEvent	Projects the hits into the track to get dE/dx profile.
TRestTrackPathMinimizationProcess	TRestTrackEvent	TRestTrackEvent	Finds the minimum path between hits inside each track.
TRestTrackReconnectionProcess	TRestTrackEvent	TRestTrackEvent	Improves physical track description after track minimization.
TRestTrackReductionProcess	TRestTrackEvent	TRestTrackEvent	Reduces the number of hits inside a track by merging closer hits.
TRestTrackToHitsProcess	TRestTrackEvent	TRestHitsEvent	It recovers back a track event into a hits event.

Rawdata processes

These processes read rawdata written in binary format and extract the signal event data to write it into a TRestRawSignalEvent.

REST process	Input type	Output type	Description
TRestRawToSignalProcess	rawdata	TRestRawSignalEvent	Used to encapsulate rawdata to signal processes.
TRestAFTERToSignalProcess	rawdata	TRestRawSignalEvent	Transforms AFTER data into raw signal event.
TRestCoBoAsAdToSignalProcess	rawdata	TRestRawSignalEvent	Transforms CoBoAsAd data into raw signal event.
TRestMultiCoBoAsAdToSignalProcess	rawdata	TRestRawSignalEvent	Transforms CoBoAsAd data into raw signal event. General version using several CoBoAsAd cards. Event data might be splitted between different data files. The process receives a list of all the files in a given run.
TRestFEMINOSToSignalProcess	rawdata	TRestRawSignalEvent	Transforms FEMINOS data into SignalEvent.
TRestMultiFEMINOSToSignalProcess	rawdata	TRestRawSignalEvent	Transforms FEMINOS data into SignalEvent. General version using several Feminos cards. Full event data is containned in one single file.

Viewer processes

These processes can be connected during intermediate event processes to visualize the input/output of different processes. Still many missing viewer processes should be developed in this section.

REST process	Input type	Output type	Description
TRestRawSignalViewerProcess	TRestRawSignalEvent	TRestRawSignalEvent	Visualizes a raw-signal event
TRestSignalViewerProcess	TRestSignalEvent	TRestSignalEvent	Visualizes a signal event

List of ROOT scripts

Pre-defined ROOT scripts are also called REST macros. They can be found in ./macros directory.

Call these macros directly in bash with "Command"; Call them in restRoot prompt by "Function name".

The macros could be changed frequently in future.

general macros

These macros provides basic data analysis functionality, including fit, integrate, etc, for observables in the anslysis tree

Command	Function name	Input arguments	Description
restCreateHisto	REST_General_CreateHisto()	TString varName TString rootFileName TString histoName int startVal = 0 int endVal = 1000 int bins = 1000 Double_t normFactor = 1	Creates and saves a histogram for a given observable in the data file.
restFit	REST_General_Fit()		Fits a given observable in the file and prints the result on screen.vis
restIntegrate	REST_General_Integrate()		Integrates a given observable in the file and prints the result on screen.
restIntegrateSmearing	REST_General_IntegrateSmearing()		Integrates with smearing a given observable in the file and prints the result on screen.

viewer macros

Command	Function name	Input arguments	Description
restViewEvents	REST_Viewer_GenericEvents()	TString fName TString EventType = ""	Shows a TRestBrowser which visualizes events in file by calling TRestGenericEventViewer
restViewG4Event	REST_Viewer_G4Event()	TString fName	Shows a TRestBrowser which visualizes TRestG4Event by calling TRestG4EventViewer
restViewGeometry	REST_Viewer_Geometry()	TString fName	Shows Geometry info saved in the given file by calling TGeoManager
restViewHitsEvent	REST_Viewer_HitsEvent()	TString fName	Shows a TRestBrowser which visualizes TRestHitsEvent by calling TRestHitsEventViewer
restViewReadout	REST_Viewer_Readout()	TString rootFile TString name Int_t plane = 0	Draw a figure of readout definition according to the save TRestReadout object in the file
restViewReadoutEvent	REST_Viewer_ReadoutEvent()	TString fName TString cfgFilename = "template/config.rml"	Shows a TRestBrowser which visualizes TRestReadoutEvent by calling TRestReadoutEventViewer

printer macros

Command	Function name	Input arguments	Description
restPrintEvents	REST_Printer_GenericEvents()	TString fName TString EventType="" Int_t firstEvent = 0	Print the event in file with given type. It calls the method TRestEvent::Print()
restPrintFileContents	REST_Printer_FileContents()	TString fName	Print name and title of Metadata/Application/Trees saved in the file.(Anything inherted from TNamed)
restPrintMetadata	REST_Printer_Metadata()	TString fName	Print the metadata content of metadata objects in the file.
restPrintTrees	REST_Printer_Trees()	TString fName	Print EventTree, AnalysisTree and observables in the file.

geant4 macros

Command	Function name	Input arguments	Description
restFindGammasEmitted	REST_Geant4_FindGammasEmitted()	TString fName	
restFindIsotopes	REST_Geant4_FindIsotopes()	TString fName TString fIsotope	
restGetBiasingError	REST_Geant4_GetBiasingError()	TString fName Int_t finalEvents = 0	
restGetROIEvents	REST_Geant4_GetROIEvents()	TString fName Double_t mean=2457.83 Double_t fwhm=0.03	
restGetROIEvents_Fiducial	REST_Geant4_GetROIEvents_Fiducial()	TString fName Double_t zMin Double_t zMax Double_t radius Double_t mean=2457.83 Double_t fwhm=25	
restListIsotopes	REST_Geant4_ListIsotopes()	TString fName TString fOutName	
restMeanTrackLeng	REST_Geant4_MeanTrackLength()	TString fName	
restQuickLookAnalysis	REST_Geant4_QuickLookAnalysis()	TString fName	
restReadNEvents	REST_Geant4_ReadNEvents()	TString fName int n1 int n2	

tool macros

Command	Function name	Input arguments	Description
restCheckReadout	REST_Tools_CheckReadout()	TRestReadoutPlane *p Int_t mask[4] Double_t region[4] Int_t N	
restCheckRunFileList	REST_Tools_CheckRunFileList()	TString namePattern Int_t N = 100000	
restDrawCombinedGasCurves	REST_Tools_DrawCombinedGasCurves()	TString fName	
restDrawResponseSignal	REST_Tools_DrawResponseSignal()		
restGenerateGasFiles	REST_Tools_GenerateGasFile()	TString cfgFile	
restMergeFiles	REST_Tools_MergeFiles()	TString pathAndPattern TString outputFilename	Merge ROOT files with given pattern
restProduceResponseSignal	REST_Tools_ProduceResponseSignal()	TString inputFileName TString outputFileName Int_t nPoints = 512 Double_t threshold = 1	
restValidateGeometry	REST_Tools_ValidateGeometry()	TString gdmlName	

REST pre-definition data

needs to be filled

> Author: Ni Kaixiang 1160274182@qq.com