J-PET 2022

J-PET Data Analysis with Framework software version 10.0

Krzysztof Kacprzak

Physics Institute, Jagiellonian University, Poland

e-mail: `k.kacprzak@alumni.uj.edu.pl`

This guide is an introduction to Framework software, prepared for data analysis in the J-PET experiment. It contains description of: installation, software design and steps of creating a new analysis module. The main purpose of this paper is to guide fresh Framework users through the first steps and serve as a reference source. This manual was originally prepared for the 9th version of Framework, June 2020. It was updated for version 10 in May 2022.

Previous version of this Guide by Magdalena Skurzok, Michał Silarski and Krzysztof Kacprzak can be found on PetWiki:

- Framework version 5.0
- Framework version 6.1
- Framework version 7.0
- Framework version 9.0

Contents

1. Introduction

J-PET experiment in its design combines several technologies, that together require a set of dedicated solutions, in order to prove itself as a functioning prototype. The purpose of analyzing the acquired data is fulfilled by software project - J-PET Framework. As the name suggests, it is a library of methods created specifically for this experiment, that can be used for creation of all sort of procedures, that would be useful in advancing the research. In this document, the user of mentioned programs can find the technical references and descriptions of nearly all pieces of this Framework.
The manual starts with an outlay of the structure and design, continues with documentation about installation and software maintenance, description of available tools and finishes with an example of some custom analysis. This version of the manual was originally prepared for the release of the 9th version of the Framework, published in June 2020. It was updated for version 10.0 in May 20202.

2. Structure of Framework

Before getting to use the Framework software, it is a good idea to review (or learn from the basics) about the J-PET experiment, since there are some `JPet*` Classes in the Framework, that represent detector elements and physical phenomena, that take place during measurements.

2.1 Detector objects

2.1.1 JPetFrame

The J-PET Detector consists of slots (two photomultipliers, one scintilator) arranged in cylindrical layers (3 at the moment), as shown in Fig. 1. It is represented by `JPetFrame` class.

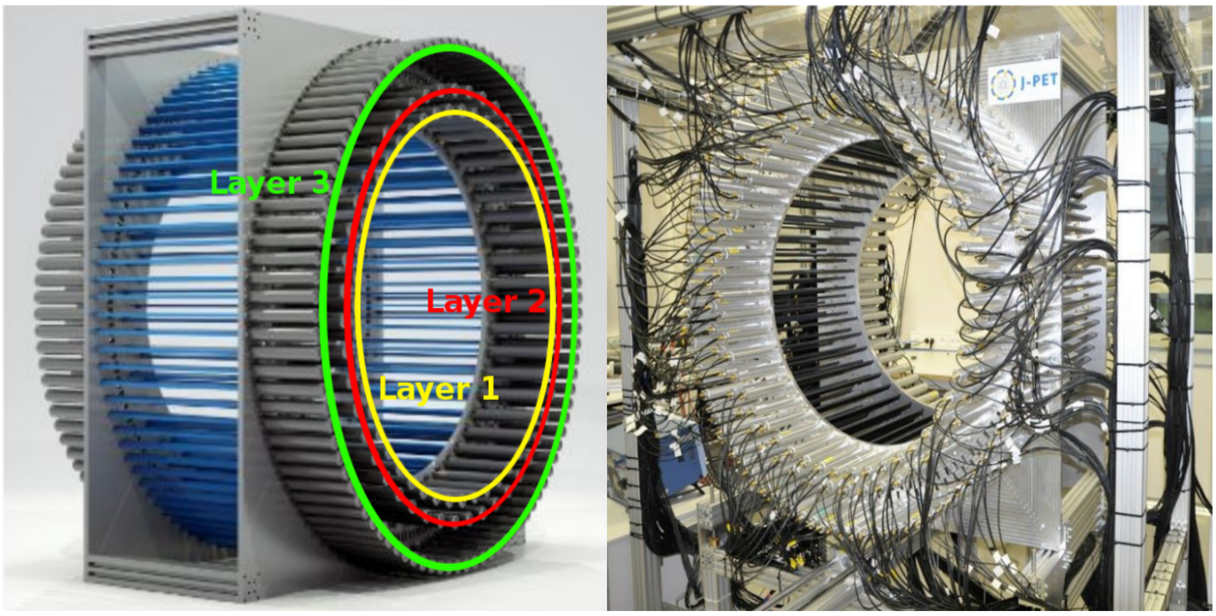


Figure 1: J-PET detector scheme with marked layers (left panel) and photo of J-PET detector setup in laboratory (right panel).

2.1.2 JPetLayer

The representation of a Layer consisting of Barrel Slots. The numbering of the Layers starts from 1 and is ordered from the one with the shortest radius to the one with the longest (see left panel of Fig. 1). The properties of each `JPetLayer` Object are: `ID, Name, Radius, Active/inactive` status. In the current setup `Layer 1` and `Layer 2` consist of 48 slots and `Layer 3` consists of 96 slots.

2.1.3 JPetBarrelSlot

The representation of Slot, that consists of Scintillator with Photomultipliers attached to the ends. `JPetBarrelSlot` objects contain information about: `ID` (in Layer), `ID in Frame` (general numbering), `Name`, `Active/inactive` status, `Angle (Theta)` that describes position in Layer. Along with the information about Layer radius and hit position, one can calculate i.e. position of hit in respect to detector center `(0,0,0)`.

2.1.4 JPetScin

The representation of Scintillators, that capture incoming photons from the source (whatever source it is). Refer to a page of PetWiki with details about Scintilators. Each `JPetScin` object holds information about: `ID`, `Sizes (dimensions)`, `Barrel Slot` it belongs to.

2.1.5 JPetPM

The representation of Photomultipliers, that measure Signals arriving from Scintillators. . Each `JPetPM` object has properties such as: `ID`, `Side (A or B)` - info that can be easily used to pair signals on the opposite PMs, `High Voltage` gains, settings and options, `FEB` (Front End Board) that it is connected to, `Barrel Slot` it belongs to.

2.1.6 Remarks so far

While using Framework there is possibility to easily obtain information about connections between objects, by using `getter` functions. So for example, if in the code we have available an object of `JPetPM` class, we are able to:

- get radius of Layer that this PM belongs to
  `float radius = pm.getBarrelSlot().getLayer().getRadius();`
- get the ID of Scintillator, that PM is connected to
  `int scinID = pm.getScin().getID();`
- check whether PM is connected to active `BarrelSlot`
  `bool isBSActive = pm.getBarrelSlot().isActive();`

Next we are describing the Classes, that represent physical phenomenons, that take place during the measurement.

2.2 Data objects

2.2.1 JPetEvent

J-PET Detector registers interactions of photons with scintillating material. Those photons originate from some physical phenomenon, lets call it an Event. The source of this Event can be different - whether it is decaying ortho-positronium or something less exciting. One can illustrate an example Event as such (Fig. 2): In first case, our Event is marked in that picture with number 2. We assume that whatever it is, it emits photons - in illustrated example three photons marked with solid line arrows originating from point number 2. This phenomenon is represented as `JPetEvent` object, that holds information about: `Hits` that construct this Event, `Type` - so was it an Event with 3 photons? Maybe with 2? Or one, that can be described as photon from deexcitation? The types of events can be various and will probably be updated in the future to describe accurately the gathered experimental data.

2.2.2 JPetLOR

A `LOR`, meaning Line of Response, is a simple container with 2 hits, designed to describe a straight line between them, which can be used for medical imaging purposes. Each `LOR` holds information about: `time [ps]`, `time difference` between hits and access to contained `JPetHit` objects. Idea of `LOR` is presented in Fig. 3.

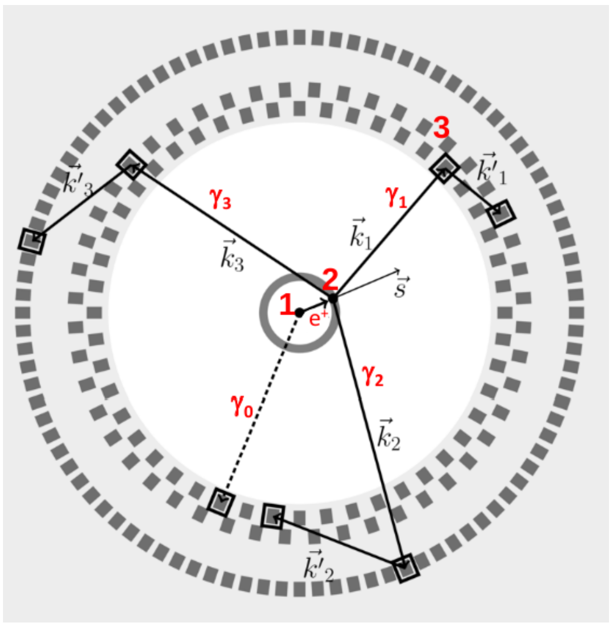


Figure 2: Schematic view of the cross section of the J-PET detector with marked ortho-Positronium decay. Ortho-Positronium decay point is marked with number 2, while outgoing photons with γ_1, γ_2 and γ_3.

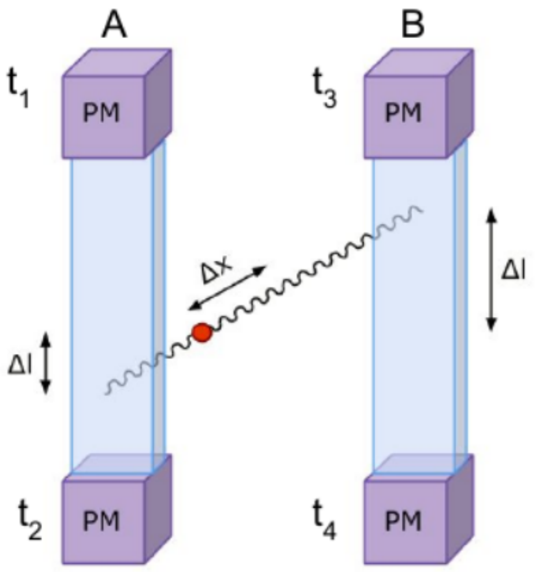


Figure 3: Idea of `LOR` construction with two hits in the pair of scintillators.

2.2.3 JPetHit

When photon (i.e. from the previous example) deposits energy in a Scintillator, light produced in the interaction, propagates in every direction - also towards the Photomultipliers at the ends. The deposition position is marked with yellow circle in the scheme presented in Fig. 4. Every `JPetHit` object contains information about: `Arrival Time [ps]`, `Energy`, `Position` in the Scintillator `(X,Y,Z)`, the two `Signals` that construct the hit, `Time difference` between arrival times of two Physical Signals.

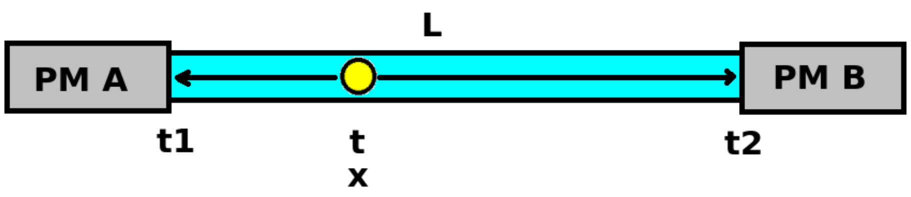


Figure 4: Schematic view of one scintillator strip with two photomultipliers on both sides. Yellow circle denotes hit of photon.

2.2.4 JPetPhysSig

The representation of reconstructed Signal, that arrives to the Photomultiplier. It contains information about: `Arrival Time [ps]`, `Number of Photoelectrons`. Currently arrival time value is inherited from `JPetRawSignal` and method of estimating photoelectrons is not yet established. `JPetPhysSig` is marked with dense-pointed curve in Fig. 5.

2.2.5 JPetRawSig

The collection of 1-8 points, that are representation of Signal Channels. In the example in the Fig. 5 there is a group of 8 points - 4 red and 4 green shown for `Leading Edge` and `Trailing Edge`. Each `JPetRawSig` object hold info about: `Number of points` that construct it, `Signal Channel Points` with division to `Leading Edge` and `Trailing Edge` points.

2.2.6 JPetSigCh

The representation of a part a Signal, that was registered on a channel in a certain Photomultiplier. It can be on one of 4 thresholds and be type of `Leading Edge` or `Trailing Edge` (that information is provided by electronics boards). Such points are represented in Fig. 5 with red and green points, thresholds with dashed lines, and the reconstructed Physical Signal is a dense-pointed curve.
Each `JPetSigCh` object has information about: `arrival time`, `Threshold number` and `Threshold value`, `Photomultiplier` it belongs to, `FEB` and `TRB` it belongs to and type of `Edge` (`Leading` or `Trailing`) .

2.2.7 JPetTimeWindow

The measurement is conducted in real time and consists of sequential periods, that are called Time Windows. For example in `Run 1` the Time Window was equal to 666,7 microseconds. There is very little sense in analyzing sets of `JPetSigCh` form different Time Windows, that are represented by `JPetTimeWindow` Class. Each `JPetTimeWindow` is a collection of `JPetSigCh` Objects, that come from all `PM` on all `Barrel Slots` on all `Layers`.

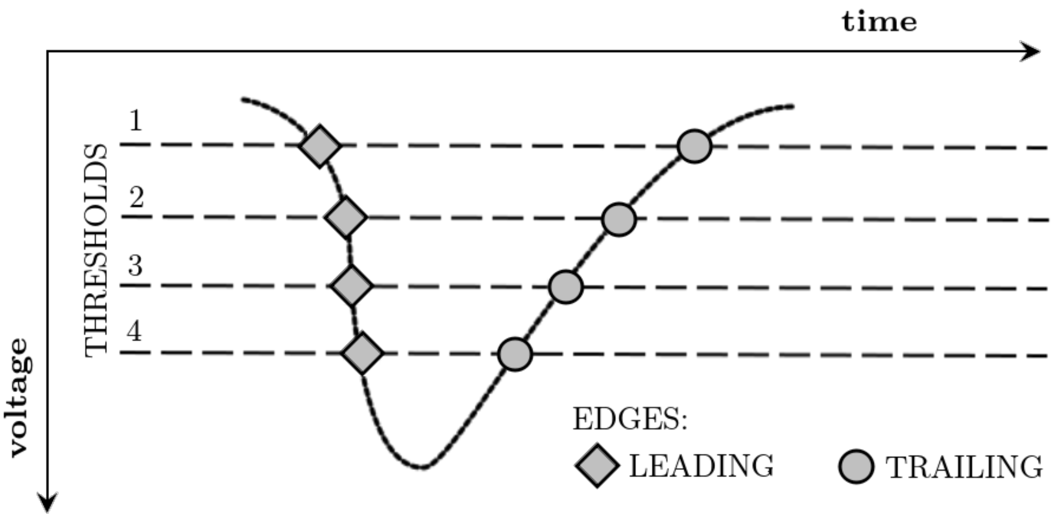


Figure 5: Schematic view of reconstructed Physical Signal (dense-pointed curve). 8 points denote representation of Signal Channel - 4 thresholds and type of `Leading Edge` and `Trailing` (4 square and 4 circle points, respectively).

2.2.8 Final remarks

When performing data reconstruction, one has to perform tasks, that construct objects from the simplest (`JPetSigCh`) to the most complex (`JPetEvent` or `JPetLOR`). The aim of the work of the whole J-PET group is to work out examples and methods, that will allow this construction ina common way, with the conservation of the order of creating objects - which is presented in Fig. 6. It is the reverse order, when compared to this presented in this Report.

2.3 Units used in reconstruction procedures

Everywhere in J-PET Framework software projects it is strongly advised to use the agreed set of units:

- voltage: `mV`
- time: `ps`
- distance: `cm`
- energy: `keV`

2.4 TOT calculation

TOT of the JPetHit can be calculated using three different methods, two of which use the information carried by the thresholds from the data acquisition system. Scheme of the calculations can be seen in Fig. 7. User can set different methods for calculation by setting an option in the JSON file `"HitFinder_TOTCalculationType_std::string"` with appropriate name of the method: `"standard"`, `"rectangular"` or `"trapeze"`. If the file with the thresholds is not given in the JSON file, Framework will take the nominal values for a given run.

The **`standard`** method uses the trailing and leading times from the signal alone. Then for each side of the strip the TOT is calculated as the sum for each threshold value, i, of the difference between trailing and leading times. If we define $TOT_i = trailingTime_i - leadingTime_i$

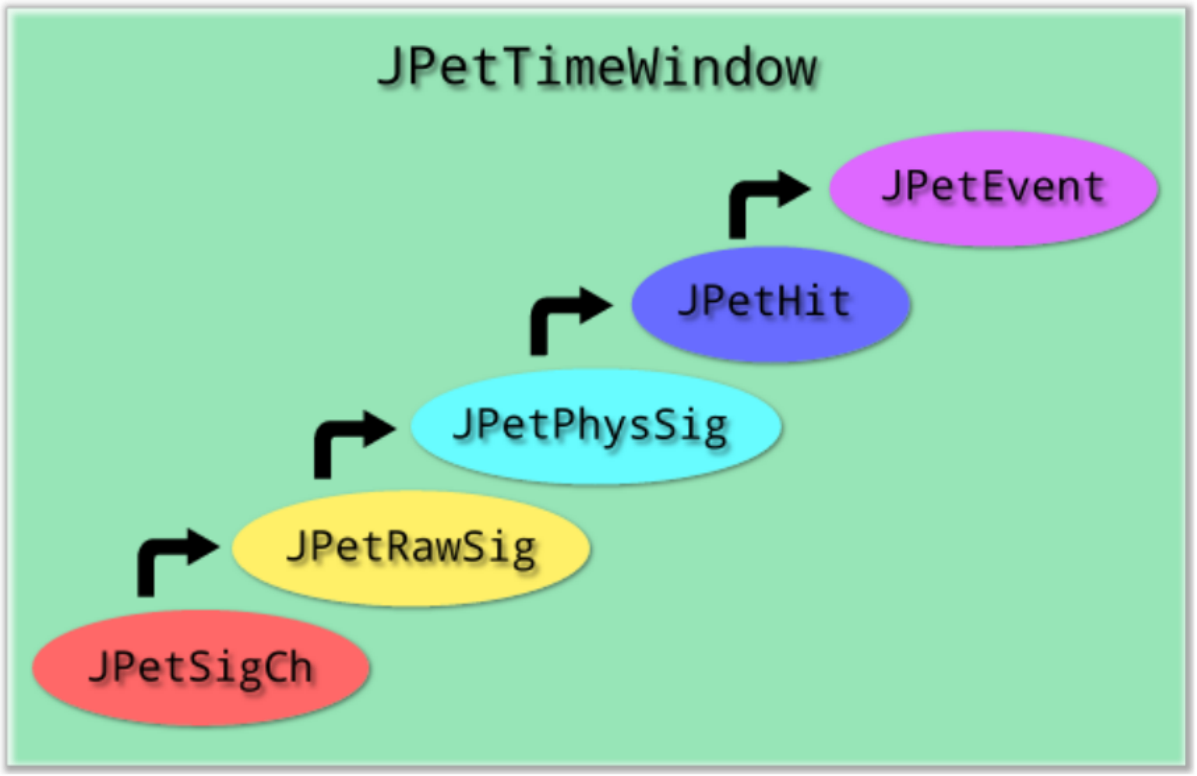


Figure 6: J-PET Objects order of reconstruction within one Time Window.

then the TOT of the signal will be equal to:

$$TOT = \Sigma_{i=1}^{4} TOT_i \tag{1}$$

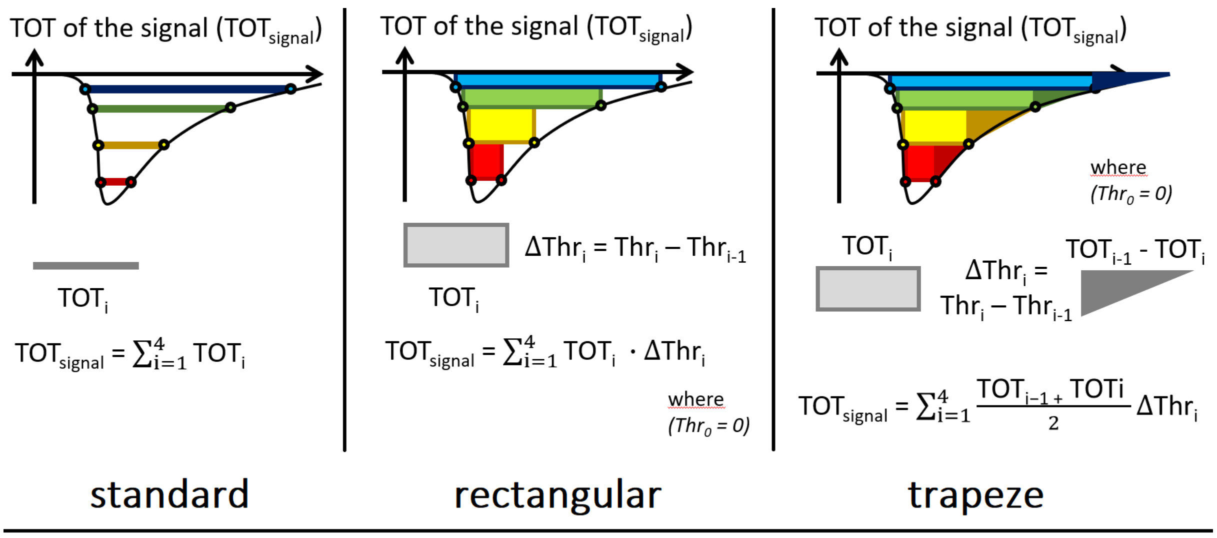
The **`rectangular`** method calculates the TOT of the signal approximating to the area of the rectangle that can be constructed as seen in Fig. 7. In this case we construct a rectangle at each threshold with an area $A = base \cdot height = TOT_i \cdot \Delta Thr_i$, with $i = threshold$ [1] and $\Delta Thr_i = Thr_i - Thr_{i-1}$, assuming that $Thr_0 = 0$. This is implemented in the code by taking an extra assumption, making the first height of value unity, $\Delta Thr_1 = 1$, and normalizing the rest of the heights to the first threshold:

$$TOT = TOT_1 + \Sigma_{i=2}^{4} \left(TOT_i \cdot \frac{\Delta Thr_i}{Thr_1} \right) \tag{2}$$

The **`trapeze`** method approximates the TOT by using the area of the trapeze defined by the area, $A = \frac{B+b}{2} \cdot h$, being B and b the respective bases (larger and shorter), and h the height of the trapeze. The full signal TOT would then be the sum of the areas of the four trapezes formed as seen in Fig. 7. In the code, this is implemented by the following formula:

$$TOT = TOT_1 + \Sigma_{i=2}^{4} \frac{\Delta Thr_i}{Thr_1} \cdot \left(\frac{|TOT_i - TOT_{i-1}|}{2} + TOT_i \right) \tag{3}$$

[1] For run 11, e.g., $Thr_1 = 30$.


TOT of the signal (TOT_{signal})
TOT_i
$TOT_{signal} = \sum_{i=1}^{4} TOT_i$
standard
TOT of the signal (TOT_{signal})
$\Delta Thr_i = Thr_i - Thr_{i-1}$
TOT_i
$TOT_{signal} = \sum_{i=1}^{4} TOT_i \cdot \Delta Thr_i$
where ($Thr_0 = 0$)
rectangular
TOT of the signal (TOT_{signal})
where ($Thr_0 = 0$)
TOT_i
$\Delta Thr_i = Thr_i - Thr_{i-1}$
$TOT_{i-1} - TOT_i$
$TOT_{signal} = \sum_{i=1}^{4} \frac{TOT_{i-1} + TOTi}{2} \Delta Thr_i$
trapeze

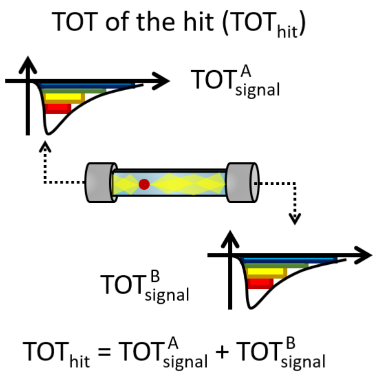

TOT of the hit (TOT_{hit})
TOT^A_{signal}
TOT^B_{signal}
$TOT_{hit} = TOT^A_{signal} + TOT^B_{signal}$

Figure 7: Different methods for the calculations of the TOT of the JPetSignal/JPetHit.

3. Framework installation

Installation process of Framework is proven to be challenging for first-time users, but do not give up! All needed information is posted on PetWiki:
`http://koza.if.uj.edu.pl/petwiki/index.php/Installing_the_J-PET_Framework_on_Ubuntu`
The best way is to follow the instructions step by step and stay optimistic. In case of problems, you can always ask for help fellow Framework ~~victim~~ user or post a support request on Redmine forum
`http://sphinx.if.uj.edu.pl/redmine/projects/j-pet-framework`

3.1 Requirements

J-PET Framework can be installed at your own computer or at J-PET server. Basically, for installation one need:

- `Linux Ubuntu` operating system - tested and used with versions from 18.04 to 22.04.
- `cmake` cmake version greater or equal 3.8
- `g++ compiler` - version supporting C++14 standard

- `BOOST` libraries `https://www.boost.org/` - working from version 1.58
- `ROOT Data Analysis Framework` - version 6 (`https://root.cern.ch/`)

Always refer to `INSTALL` file, to check required libraries. There are three components that should be installed

1. Unpacker2 - a low level tool, to decode binary information from FEBs
2. Framework library - all data and core objects as well as the architecture are defined and described in this library
3. Examples (with MLEM module) - a set of examples on how to use framework library to achieve few different goals in analysis

3.2 Installing `Unpacker2`

Installation of Unpacker project can be performed as described below.

```
git clone https://github.com/JPETTomography/Unpacker2.git
mkdir unpacker-build
mkdir unpacker-install
cd unpacker-build/
cmake -DCMAKE_INSTALL_PREFIX=../unpacker-install ../Unpacker2
make
make install
```

Note that, if you choose to provide install path in a `root` directory (for example `/usr/local/`), you need to run install with admin access: `sudo make install`.

After successful installation, there should be a shared-object library `libUnpacker2.so` in set folder: `unpacker-install/lib`. In order to run framework programs, you may need to add this path to the `LD_LIBRARY_PATH` environment variable. You can do it using the script `thisunpacker.sh` located in the *bin* subdirectory of the Unpacker2 installation location, e.g. `unpacker-install/bin/thisunpacker.sh` in this example:

```
source unpacker-install/bin/thisunpacker.sh
```

3.3 Installing `Framework`

This procedure is similar to installation of `Unpacker2`:

```
git clone https://github.com/JPETTomography/j-pet-framework.git
mkdir framework-build
mkdir framework-install
cd framework-build/
cmake -DCMAKE_INSTALL_PREFIX=../framework-install ../j-pet-framework/
```

During `cmake` the test files will be downloaded from the server.

```
make
make install
```

After successful installation, there should be a shared-object library `libJPetFramework.so` in set folder: `framework-install/lib` Note that, if you choose to provide install path in a `root` directory (for example `/usr/local/`), you need to run install with admin access: `sudo make install`.

3.4 Building example procedures

The repository `j-pet-framework-examples` contains several procedures, that are described in section 5. Before installation with `cmake` is it required to set proper paths with `thisframework.sh` script:

```
source <path/to/framework>/bin/thisframework.sh
```

If you followed the described above example, then it is:

```
source framework-install/bin/thisframework.sh
```

Then follow the recipe:

```
git clone --recursive
   https://github.com/JPETTomography/j-pet-framework-examples.git
mkdir examples-build
cd examples-build
cmake ../j-pet-framework-examples/
make
```

How to use the executable is described in section 5.5.

3.5 Using j-pet-servers

Constantly growing computing infrastructure for J-PET experiment is administrated by Eryk Czerwiński, contact him in case of a need for access to the experimental data or server account creation. To login to servers from outside of the Institute, you need a personal VPN access, ask your supervisor how to obtain it from Departments IT staff. Using a VPN access:

- Install `openvpn` on Ubuntu (version 2.4.3):
  `sudo apt-get install openvpn`
- With your private key, use software to connect to VPN:
  `sudo openvpn <yourName>.conf`
  `<yourName>.conf` is the name of your private configuration file
- provide you `sudo` password and you private VPN `key`
- after successful connection, in another terminal you can connect with j-pet-server:
  `ssh yourUserName@serverIP`

3.6 GitHub repositories

J-PET Framework software is maintained within public repositories on GitHub:

- main library: `https://github.com/JPETTomography/j-pet-framework`
- examples: `https://github.com/JPETTomography/j-pet-framework-examples`

The most useful way to obtain Framework and Examples code, is with `Git` version control system. If you are not acquainted with `Git`, there are many useful tutorials for beginner users of version control systems, i.e. `http://rogerdudler.github.io/git-guide/`. To obtain Examples and Framework at the same time, just type in terminal the command:

```
git clone --recursive
https://github.com/JPETTomography/j-pet-framework-examples.git
```

4. What and where and maybe... how?

4.1 Installation result

After installation, J-PET Framework is located in folder `j-pet-framework-examples`. This folder contains:

- `LargeBarrelAnalysis`, which is the current best attempt and a working version for universal reconstruction of experimental data gathered with the Large Barrel setup. Some of the modules are used in the other examples, those tasks are described in details in Sec. 5.4.
- 3 extensions of reconstruction procedures for a approach to data streaming: `CosmicAnalysis`, `Imaging` and `PhysicAnalysis`.
- calibration procedures:
  - `TimeCalibration` - a procedure analyses data from measurement with reference detector and produces files with synchronization constants of all scintillators in the detector.
  - `TimeCalibration_iter` - a iterative version of time calibration
  - `InterThresholdCalibration` - an analysis of time walk effect
  - `VelocityCalibration` - produces values of effective speed of light in scintillators
- `MCGeantAnalysis` - a task, that translates output data from J-PET Monte Carlo with Geant4 project into structures in Framework
- `ImageReconstruction` - an example using `j-pet-mlem` for imaging and sinogram creation
- `UserDataClassExample` - an example of a custom data class, that can be an extension of default `JPetEvent` or other structure
- `NewAnalysisTemplate` a place to start your own analysis. This example links the modules from `LargeBarrelAnalysis`, so a goal for a user is to add new `Tasks`, that analyze the data further, after completion of all previous procedures. It is advised to use for event categorization the tools and methods, that are available in i.e. `EventCategorizerTools` class from `LargeBarellAnalysis` example.
- scripts - here you can find useful ROOT scripts, that load Framework shared-object library at starting ROOT software (i.e. `rootlogon.C`).
- `DOXYGEN` documentation, if generated, can be found in build folder in html or latex directories. Method of generating the documentation is described in Sec. 4.4,
- `CMakeList.txt` file in which all analysis sub-directories should be added like this `add_subdirectory(LargeBarrelAnalysis)`.
- after successful `cmake` command (refer to installation guide) there is the directory containing additional files, to be used during execution of examples: `CalibrationFiles`. The are downloaded from the `sphinx.if.uj.edu.pl` server.
- your `build` directory should contain folders for each sub-directory listed in `CMakeLists.txt` with executable programs; directory with Framework shared-object library `libJPetFramework.so`.

4.2 Note about submodules

If after executing `git clone` for obtaining project files with example procedures, the `j-pet-mlem` directory is empty, you have to obtain this submodule searately:

```
git submodule init
git submodule update
```

And the same for linking `Unpacker` to `j-pet-framework`:

```
cd j-pet-framework/
git submodule init
git submodule update
```

4.3 Note about Monte Carlo

The simulations with Mote Carlo methods are being developed as a separate project J-PET-Geant4. This is a piece of software using simulation package Geant4. The output of this program needs to be translated into J-PET Framework structures with the use of `MCGeantAnalysis`, that creates `JPetHits` and analyses them later in the same way as main reconstruction procedures. The manual for working with this Monte Carlo simulator is available in .

This is a piece of software using simulation package `Geant4`. Output of this program needs to be translated to J-PET Framework structures with the use of `MCGeantAnalysis`, that creates `JPetHits` and analyses them later the same way as main reconstruction procedures.
It is done by the example `MCGeantAnalysis`, that utilizes class `JPetGeantParser`. During execution of this program (and as well MC generation with `J-PET-Geant4`) the default `ROOT` random generator is used both for lifetime value and for applying the experimental smearing resolutions. The user can provide an option in the `json` file with the seed value that will be used by the random generator to start the pseudo-random sequence. If no option is provided the default 0 seed value is used and the random generator will generate seed based on the `ROOT TUUID` identifier. More information about available options for this procedure can be found in `PARAMETERS.md` file in the `MCGeantAnalysis` folder.

4.3.1 Experimental smearing

There are currently three hit properties that are modified in the `JPetGeantParser` by applying the experimental smearing. Those are: time, deposited energy and the hit position along the scintillator (z coordinate), and the corresponding three smearing functions. The functions have four default parameters, and can have any number of additional parameters. The default parameters are scintillator ID, hit position along the scintillator (z), energy, time. The user can change both the definition and the values of the smearing parameters by providing the appropriate options in the configuration file. More information about available options for this procedure can be found in `PARAMETERS.md` file in the `MCGeantAnalysis` folder [2].

As a artificial small example, let's assume that we would like to keep the standard function definitions and parameter values for time and energy smearing, but substitute the z position smearing by our function based on the Landau distribution. We will use ROOT:

```
TMath::Landau(Double_t z, Double_t mpv = 0, Double_t sigma = 1, Bool_t norm = kFALSE)
```

[2] If no definitions or values are given in the configuration file, the default mathematical definitions and the values of the parameters of the smearing functions can be found in the `j-pet-framework/src/GeantParser/JPetSmearingFunctions/JPetSmearingFunctions.cpp`

implementation. So any z hit position from the Geant, will be replaced by the random value according to the Landau distribution, where mpv corresponds to the input z value. We have arbitrary chosen following set of parameters: $norm = 0(kFALSE)$ and the sigma parameter will be read from the configuration file. Earlier, we mentioned that any smearing function has 4 default parameter namely: scintillator ID, hit position along the scintillator (z), energy, time. In ROOT nomenclature, they correspond to elements $p[0], p[1], p[2], p[3]$ of the parameter array. Therefore sigma will correspond to the 5th parameter ($p[4]$). In addition, we can also set a range of usage to our function.

In the configuration file we put the following lines:

```
{
  "GeantParser_ZPositionSmearingFunction_std::string":"[&](double*␣x,␣
     double*␣p)->double{␣return␣TMath::Landau(x[0],p[1],p[4],␣0);};",
  "GeantParser_ZPositionSmearingParameters_std::vector<double>":[2.0],
  "GeantParser_ZPositionSmearingFunctionLimits_std::vector<double>":"[-4.0,
     ␣4.0]"
}
```

The definition of the function uses quite obfuscated syntax of lambda functions (see some examples of TF1 definitions with lambdas here: `https://root.cern.ch/doc/master/classTF1.html#F3`), however the main definition is rather straightforward to understand, it is `TMath::Landau(x[0],p[1],p[4], 0)`. The function takes p[1] as mean, and p[1] corresponds to input z position, p[4] as sigma which is the parameter provided in the second line. The third line corresponds to the function limit e.g. for the input z value equal 2, the randomization will be done in the range $[-4 + 2, 4 + 2] = [-2, 6]$.

4.4 Documentation

The code documentation can be generated in `build` directory with `Doxygen` package with command:

```
make documentation
```

or inside `j-pet-framework` directory (where `Doxyfile` is located)

```
doxygen
```

Look for `index.html` file, that is available in the `j-pet-framework/html/` directory and open it with your favourite web browser.

5. Analysis step by step

This section includes useful information concerning the J-PET data analysis. The steps are presented based on `LargeBarrelAnalysis` example.

5.1 Obtaining data files

The data from measurements with J-PET detector is recorded on servers and eventually transported to tapes for storage. Information about types of files and their location can be found on `PetWiki` in Documents/Reports section and in Archived Data section, in case if the files

were moved to tapes. The right person to ask for data access is Eryk Czerwiński, and other J-PET collaborators for sure store single files in personal workspaces. Raw data files are in `HLD` format (uncompressed files) and have usually size of 2 GB.
The file name only will not provide information about type and purpose of measurement, so it is up to the user to get the knowledge what is she/he about to analyze with Framework software. However the original filenames contain the time of the acquisition, following a rule

```
dabc_<year (two digits)><day of the year><hour><minute><second>.hld
```

There are many types of measurements done during the Runs, i.e. calibration runs - with collimator or reference detector, bare source, annihilation chamber of different sizes, imaging tests, phantoms etc.

5.2 Calibration and detector setup files

In order to perform a meaningful analysis of J-PET data, the user needs to provide proper calibration and setup description files corresponding to the same run as the processed data files. As these files are specific to a particular run or measurement, they are intentionally not included in the example program directories. The example programs represent general analysis use cases generally independent of the type of data, therefore it is the responsibility of the user to identify and provide the right calibration files.

Note: The standard analysis example *"LargeBarrelAnalysis"* by default refers to a dummy calibration file `dummyCalibration.txt`. The sole purpose of this file is to demonstrate the structure of such a calibration file and the values contained therein are not meaningful for any reasonable analysis! Please make sure you replace it with the relevant calibration files.

Calibration and detector setup files can be found in two locations:

- At the following PetWiki page: `http://koza.if.uj.edu.pl/petwiki/index.php/Default_settings_and_parameters_used_in_the_analyses`,
- In the `CalibrationFiles` directory located inside the `j-pet-framework-examples` source directory to where the files are automatically downloaded when you build the examples for the first time.

In the latter location, the calibration files are grouped in subdirectories by the number of run.

Paths to all of the required calibration files should be defined by the user in the JSON file with user options (see Section 5.5).

The following calibrations are needed in order to perform a reasonable analysis of data:

- **Calibration of:** TOT stretcher offsets in the HLD file
  User option in the JSON file: `"Unpacker_TOToffsetCalib_std::string"`
  Example file for Run 5: `CalibrationFiles/5_RUN5/TOTConfigRun5After.root`
  Applied in module: Unpacker
  File format: ROOT
  Note: Stable in time and thus only redone once in every few runs; If not present for your run, find and choose the most recent of previous runs which has it.
- **Calibration of:** Nonlinearity of TDC response in the HLD file
  User option in the JSON file: `"Unpacker_TDCnonlinearityCalib_std::string"`
  Example file for Run 5: `CalibrationFiles/5_RUN5/TDC_CORR_RUN4.root`

Applied in module: Unpacker
File format: ROOT
Note: Stable in time and thus only redone every few runs; If not present for your run, find and choose the most recent of previous runs which has it.

- **Calibration of:** Time offsets between photomultiplier signals
  User option in the JSON file: `"TimeCalibLoader_ConfigFile_std::string"`
  Example file for Run 5: `CalibrationFiles/5_RUN5/TimeCalib_Run5_RefDetTDCCorr.txt`
  Applied in module: TimeWindowCreator
  File format: text file (txt)
  Note: Event though the default unit for time-related values in the Framework are picoseconds, **the time offsets in this calibration file are stored in nanoseconds**.

- **Calibration of:** Effective light velocity in scintillator strips
  User option in the JSON file: `"HitFinder_VelocityFile_std::string"`
  Example file for Run 5: `CalibrationFiles/5_RUN5/EffVelocitiesRun5.txt`
  Applied in module: HitFinder
  File format: text file (txt)
  Note: Event though the default unit for time-related values in the Framework are picoseconds, **the constants in this calibration file are stored in centimeters per nanosecond**.

Moreover, two types of files describing the hardware setup in a given measurement are required. In contrast to the calibrations, paths to these files are not provided as user parameters in the JSON file but specified on the command line when starting the analysis program (see the example command line in Section 5.5). These files are:

- **Description of:** Detector geometry, element connections and threshold values
  Command line parameter: `"-l"`
  Example file for Run 5: `CalibrationFiles/5_RUN5/detectorSetupRun5.json`
  File format: JSON
  Note: This file also contains the threshold values. Therefore, runs which featured measurements with different threshold configurations (usually labeled A, B, C, ...) usually have a separate detector setup file for each of ruch measurements.

- **Description of:** DAQ setup
  Command line parameter: `"-p"`
  Example file for Run 5: `LargeBarrelAnalysis/conf_trb3.xml`
  File format: XML
  Note: The same file, usually named `conf_trb3.xml` is used for all measurements with J-PET large barrel. Therefore, this file is not kept among the run-specific ones but directly downloaded to the `LargeBarrelAnalysis` example directory.

5.3 Run-specific user options

Besides paths to calibration files, the JSON file with user options (passed with the `-u` command line parameter, see Section 5.5) also defines certain options which vary depending on the type of data being analyzed:

- `"TimeWindowCreator_MinTime_float`: Minimum reasonable TDC time in nanoseconds. As the times from J-PET TDC-s are negative, this corresponds to -1*(size of the time window) where the time window size is:
  - 650 μs = 650000 ns for runs up to 5,

  - 20 μs = 20000 ns for runs starting from 6.

- `"TimeWindowCreator_MaxTime_float"`: Maximum reasonable TDC time in nanoseconds. As the times from J-PET TDC-s are negative, this should be 0 for all runs.
- `"ModuleName_UseCorruptedHits_bool"`: Whether a given module (there will be one such option for every standard analysis module) should pass data marked as corrupted on for further analysis. Use `false` for all modules unless you intend to study the impact of corrupted data filtering.
- `"EventFinder_EventTime_float"`: Size of the coincidence time window (in picoseconds) used to identify events, i.e. groups of hits clustered in time. A value of 4000 ps is commonly used to identify event candidates. However, if you intend to e.g. observe ortho-positronium lifetimes you may want to set it much larger, e.g. 300000 ps.
- `"EventFinder_MinEventMultiplicity_int"`: Controls how many hits must be found in coincidence in order to identify them and store as an event for further analysis. The higher multiplicity is required, the stronger is the data reduction by the EventFinder module.

Other user options control the behaviour of analysis modules which have become practically standard and do not require modification of the default values in most typical analysis cases.

5.4 Analysis modules

Each directory with Framework analysis contains modules responsible for proper tasks and `main.cpp` file with included modules. Every Framework module consists of `init()`, `exec()` and `terminate()` methods. The `init()` and `terminate()` methods are executed only once in each module. One can e.g. make there histograms, which next are filled in `exec()` method, that runs separately for each Time Slot within the data file. Basically the analysis modules correspond to proper procedures, that are transforming simpler data structures to a more complex ones, as it was show in Figure 6. The program starts with `HLD` file (it may be compressed with i.e. `xz`) and transforms it to sequence of `ROOT` files with different extensions. The output from one module is an input for the next.

5.4.1 LargeBarrelAnalysis modules

The modules of considered `LargeBarrelAnalysis` example (located in directory: `j-pet-framework-examples/LargeBarrelAnalysis`) are following:

- `Unpacker` - reading `HLD` file and making it usable for `ROOT` format, uses configuration `XML` and setup `JSON`
- `TimeWindowCreator` - process unpacked `HLD` file into a tree of `JPetTimeWindow` objects with `JPetSigCh`. Uses file with time calibration offsets values and performs check for data corruption and proper flagging of signa channels.
- `SignalFinder` - creates Raw Signals with the matching procedure based on parameters of time windows.
- `SignalTransformer` - creates Reco & Phys Signals

- `HitFinder` - creates Hits from Physical signals, calculates Hit position with the use of values of velocities of light from proper calibration file.
- `EventFinder` - creates Events as group of Hits within a time window of a given value.
- `EventCategorizer` - categorizes Events with simple procedures, assigning one or several types of a event.

5.5 Analysis Run

In case of `LargeBarrelAnalysis` example described in previous Section, one can run analysis in directory `/build/LargeBarrelAnalysis` in a manner illustrated by an example:

```
 ./LargeBarrelAnalysis.x -t hld -f dabc_16218140613.hld
-l detectorSetup1.json -i 1 -p conf_trb3.xml -o outputDir/
-r 0 100000 -u userParams.json -b
```

General way of executing the program with options is:

```
./<program_name>.x -t <opt_t> -f <opt_f> -l <opt_l> -i <opt_i> -p <opt_p>
-o <opt_o> -r <opt_r> -u <opt_u> -b
```

where: `t, f, i, l, p, o, r, u, b` are options corresponding to:

- `opt_t` - input data file format (usually `HLD`, `ROOT` or compressed `XZ, GZ, BZ2`)
- `opt_f` - path to data file `path/filename` with the same extension as declared with `-t` option
- `opt_l` - `JSON` file with detector setup
- `opt_i` - run number, you can find it as a first key in the used `JSON` file
- `opt_p` - `XML` file with configuration for `Unpacker` task
- `opt_o` - path to folder, where you want the resulting files to be written. If not specified, the output files will be created in the same directory as input file
- `opt_r` - range of analyzed events, two numbers denoting entry index - begin and end (i.e. 1230 2340). Option useful for quick tests, when it is enough to limit run to analysis of several Time Slots.
- displaying progress bar - use this option (without any argument) to show a percentage of task progress. Turned off by default.
- `opt_u` - `JSON` file with user options. This is the best way to declare parameters used by different modules, without the need of recompiling whole code. Please see the `PARAMETERS.md` file for description of values/files to be set in `JSON` file. Contents of the file for `LargeBarrelAnalysis`
- `b` - optional, without any parameters, if used it displays information about current task and percentage of the progress.

5.6 Place to start - NewAnalysisTemplate

Let's start!
First enter directory `j-pet-framework-examples/NewAnalysisTemplate`. There you can find basic files: `CMakeList.txt`, `README.md` and `main.cpp`. Modify `main.cpp` to fit your needs: include or exclude modules linked from `LargeBarrelAnalysis` i.e. by commenting them out or removing; add your custom task.

```
#include <JPetManager/JPetManager.h>
#include "../LargeBarrelAnalysis/TimeWindowCreator.h"
// #include "../LargeBarrelAnalysis/SignalFinder.h"
#include "MyCustomTask.h"
using namespace std;

int main (int argc, const char * argv []) {
    JPetManager& manager = JPetManager::getManager();
    manager.registerTask<TimeWindowCreator>("TimeWindowCreator");
    // manager.registerTask<SignalFinder>("SignalFinder");
    manager.registerTask<MyCustomTask>("MyCustomTask");
    manager.useTask("TimeWindowCreator", "hld", "tslot.calib");
    // manager.useTask("SignalFinder", "tslot.calib" , "raw.sig");
    manager.useTask("MyCustomTask", "tslot.calib" , "my.sig");

    manager.run(argc, argv);
}
```

Class `MyCustomTask` should extend `JPetUserTask`, in this example it takes as an input the result of task `TimeWindowCreator`. If needed, another folder with separate analysis can be added. If your directory structure looks like this:

```
j-pet-framework-examples
  build/
  LargeBarrelAnalysis/
  NewAnalysisTemplate/
  j-pet-framework/
  CMakeLists.txt
  ...
```

then add new folder (i.e. `MyFirstAnalysis`) inside `j-pet-framework-examples` with proper `main.cpp` file and append the `CMakeLists.txt` in the top directory with

```
add_subdirectory(MyFirstAnalysis)
```

To compile, go to `build/` directory and run

```
cmake ..

make
```

All the examples, including yours, shall be built. The executable file is in
`j-pet-framework-examples/build/MyFirstAnalysis`
directory, and it is possible to run it as it was demonstrated above. As a result of the analysis we obtain output files with names corresponding to original file name, with extensions appropriate to module definition. So each task - one output file. The result of a `LargeBarrelAnalysis`

program performed on a file named i.e. `dabc_16218140613.hld` shall be:

```
dabc_16218140613.tslot.calib.root
dabc_16218140613.raw.sig.root
dabc_16218140613.phys.sig.root
dabc_16218140613.hits.root
dabc_16218140613.unk.evt.root
dabc_16218140613.cat.evt.root
```

5.7 Merging files produced with the framework using the `hadd` tool

Multiple ROOT files produced by the same framework analysis module can be merged using the `hadd` program provided by ROOT. However, in order for `hadd` to work correctly with files produced by the framework, the user need to set the appropriate environment prior to calling `hadd`. For this, it is enough to call the following scripts:

- `<Unpacker2_install_location>/bin/thisunpacker.sh`,
- `<Framework_install_location>/bin/thisframework.sh`.

For the installation paths used in the examples in Section 3, the appropriate environment for hadd-ing of files can be set with:

```
source unpacker-install/bin/thisunpacker.sh
source framework-install/bin/thisframework.sh
```

Please note that these steps need to be repeated in every new terminal session before calling `hadd`.

6. Contacts and useful links

6.1 Reporting problems and searching for assistance

In case of problems when using the J-PET Analysis Framework, please first check the *Frequently Asked Questions (FAQ)* section of PetWiki:

`http://koza.if.uj.edu.pl/petwiki/index.php/Framework_FAQ`

If the problem you encountered is not reported there, please report it through the Redmine platform [3]:

`http://sphinx.if.uj.edu.pl/redmine/projects/j-pet-framework`

- questions related to usage should be asked in the Redmine Forum (`http://sphinx.if.uj.edu.pl/redmine/projects/j-pet-framework/boards`)
- bugs and requests of new features should be reported as *issues* in Redmine (`http://sphinx.if.uj.edu.pl/redmine/projects/j-pet-framework/issues/new`)

6.2 Contacts to experts on particular components

- **Aleksander Gajos** (aleksander.gajos@uj.edu.pl) - J-PET Framework development and PetWiki
- **Kamil Dulski** (kamil.dulski@gmail.com) - development of J-PET Monte Carlo simulations project

[3] if you do not have an account, contact Aleksander Gajos (aleksander.gajos@uj.edu.pl) to have it created

- **Szymon Niedźwiecki** (szymonniedzwiecki@googlemail.com) - both general and detailed knowledge of J-PET experiment
- **Krzysztof Kacprzak** (k.kacprzak@alumni.uj.edu.pl) - development of J-PET Framework examples
- **Wojciech Krzemień** (wojciech.krzemien@ncbj.gov.pl) - Framework development and design
- **Eryk Czerwiński** (eryk.czerwinski@uj.edu.pl) - administration of servers and data access/management

Links:

- **J-PET Experiment** homepage
  `http://koza.if.uj.edu.pl/pet/`
- **PetWiki** - all essential information, repository of publications, seminar presentations, current experiment and lab status
  `http://koza.if.uj.edu.pl/petwiki/`
- **Redmine** - bug tracking, task management and discussion forum for Framework developers and users
  `http://sphinx.if.uj.edu.pl/redmine/`
- **Online documentation** generated with Doxygen
  `http://sphinx.if.uj.edu.pl/framework/doc/`