Water Cerenkov muon detector near the Angra-II reactor core: the software.

A. F. Barbosa[*]
Centro Brasileiro de Pesquisas Fsicas - CBPF

Abstract

In a previous note [reference], we reported the installation of a cosmic ray detector based on the Cerenkov effect and described the hardware implemented on it for data acquisition. We here present the corresponding software, with emphasis on the framework used to address the issues imposed by the data acquisition requirements. The main tools adopted are the *Root* environment for *C++* object-oriented programming and *Linux* as the operational system. The essential ingredients used in the code development are presented so that other applications may be built from this one.

[*]e-mail: laudo@cbpf.br

1 The *Root* environment

The high energy physics community using large experimental facilities such as *CERN* (European Organization for Nuclear Research, *www.cern.ch*) has developed efficient programming tools to face the challenges imposed by the huge amount of data generated at the experiments around particle accelerators, especially after the two last decades. In parallel, computer sciences have also evolved rapidly, making available new computer programming techniques, such as the object-oriented concept. A combination of these developments led to the implementation of the *Root* (*root.cern.ch*) programming framework. This framework relies on a software package, *CINT*, which allows the interpretation of the *C++* programming language commands. In addition, *Root* is also based on (but no limited to) the use and sharing of open libraries in an open-source free operational system: *Linux* [reference]. As a consequence, the framework operation and development takes benefit on feedback provided by its users, and it turns out to be of widespread use in most scientific collaborations involving physics research, as well as many other communities.

In the Angra Neutrino Project we have adopted *Root* and *Linux* as the platform for data acquisition code development. This choice had to comply with two main requirements: the software has to be able to interact with the hardware, acquiring data related to events randomly occurring in the detector; it also has to be a self-contained package. The latter means that a compiled executable file has to be provided, and the application must not run as a script dependent on the framework version.

Although the whole detector is not presently assembled, we may start to look at the possible strategies to develop the main processes dealing with data acquisition. Some of these will require a graphical user interface. The present note is a description of one such case. We report the production of a complete software package featuring the required capabilities, with enough details for other groups to use it as a startup for building their own applications.

2 Algorithm and program structure

The goal is to provide the software to acquire data from the water Cerenkov detector installed in the container, as reported in [reference], where the Data Processing Module (DPM) was described. Analog signals are driven to the DPM, inside which a Field Programmable Gate Array (FPGA) manages to feed data to a computer via the Universal Serial Bus (USB) port. From the hardware side, the software task is therefore to communicate with the USB port.

Two modes of operation are expected for data acquisition: either it is triggered externally (by a coincidence digital signal, for example), or it is internally triggered whenever the amplitude of the analog signal present at one of the four DPM channels exceeds a preset threshold. Both modes have to be programmed and controlled at the software level, including the threshold adjustment.

Once an event is triggered, the algorithm has to read 541 bytes of information, corresponding to the digitalization of the analog signals in the DPM channels. The first 270 bytes refer to the photo multiplier tube (PMT) signal, sampled in channel 1. This actually corresponds to the waveform produced by the detector when Cerenkov light is generated by an incident particle. The next 134 bytes are used to store the temperature, provided by the DC level output of a sensor which is input to channel 2. The DC level is directly proportional to temperature. Another 134 bytes are read and refer to the counting rate, which is also a DC level provided by a dedicated circuit and input to channel 3. DC levels are rather understood as slowly varying signals, the average of which is assigned to the measured parameter. Two bytes are used to eventually read the counting rate as estimated by an

Library	**Header files**
Standard *C*, *C++*	stdlib.h, stdio.h, time.h, iostream.h, string.h
Root	TROOT.h, TObject.h, TQObject.h, RQ_OBJECT.h, TObjString.h, TApplication.h, TVirtualX.h, TGFrame.h, TGIcon.h, TGLabel.h TGButton.h, TGTextEntry.h, TGNumberEntry.h, TGMsgBox.h, TGMenu.h TGCanvas.h, TGComboBox.h, TGTab.h, TGFileDialog.h, TGTextEdit.h, TGColorSelect.h, TRootEmbeddedCanvas.h, TCanvas.h, TColor.h, TList.h, TGraph.h, TLine.h, TPolyLine.h, TRandom.h, TH1F.h, TSystem.h, TSystemDirectory.h, TFile.h, TTimer.h, TThread.h
USB Driver	ftd2xx.h, ftd2xx.cxx

Table 1: Libraries and header files used in the DAQ software.

algorithm running independently as an FPGA state machine. The first and the last two bytes carry fixed information which delimits the data block, so that it is possible to make sure that the transferred data is valid. Channel 4 of the DPM is not used.

From the user side, the software has to be able to display the raw data (in a graph and on tables) and to provide the communication with the hardware. Some pre-processing of data may be done, for example to compute the charge absorbed by the detector, the signal amplitude etc. The user may as well enable or disable features like data saving, graphics update and other facilities.

2.1 Libraries and header files

The *C* and *C++* programming languages allow the programmer to have access to lots of functions (functions are here understood as subroutines, methods, classes of objects etc.) which are organized in libraries. In order to do so, it is only required that a header file defining the functions is included in the source code, with the *#include* command. This command is a preprocessor directive, indicating that the functions listed in the header file will be compiled with the source code. The included functions, which have to be found by the compiler, are stored in standard directories (typically */usr/lib*, */usr/local/lib*).

The libraries and header files used in the application here described are listed in Table 1. For simplicity, all these files are included in a single header file called *DAQ_Tank.h*, which also contains the declarations of other functions and classes required in the code, as shown in Table 2 and described in the next section.

The final code is, therefore, composed by the included functions and the functions developed for the particular application. The latter have to be locally implemented and compiled. In a special function, called the main one and referred to as *main()*, the others are eventually called and used as necessary in the algorithm. All the implementation is concentrated in a source file (here called *DAQ_Tank.cxx*) in which the main header file is included as mentioned above. It must be emphasized that many powerful programming tools are developed and made available to programmers for inclusion in their own applications under a given programming framework. Programmers take great benefit on using these tools, and have to take a framework that matches the expected code performance.

/* List of used header files */
...
/* List of command identifiers */
enum *ComandIdentifiers* {
... };
/* List of classes */
class *TGMainFrame*;
class *TTHread*;
class *TestMainFrame* : public *TGMainFrame* {
... };
class *SetupMPDBox* : public *TGTransientFrame* {
... };
class *DAQThread* : *TObject* {
... }
/* List of local functions */
float *get_baseline*(float y[]);
float *get_amplitude*(int size, float base, float y[]);
float *get_charge*(int size, float base, float y[]);
float *get_average*(float y[]);
void *Draw_Pulse*();
void *Show_Data*();
void *Draw_Grid*();
void *Draw_Text*();

Table 2: *DAQ_Tank.h*: the main header file structure.

3 Implementation as a graphical user interface

Taking the point of view of the user who will need to interact with the hardware and plan data acquisition runs, what is required is preferably a graphical interface from which he may easily find the way through the program facilities. We describe the code implementation from this point of view.

Once the program is launched, the user is prompted to the application main window, as shown in Figure 1. From this, all the planned facilities are provided, making use of the objects available in the framework.

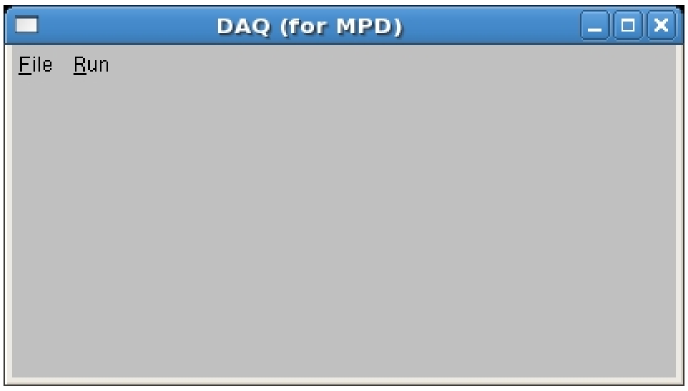


Figure 1: The main window prompted to the user.

Two kinds of objects are readily identified in the main window. The first one is the window itself. It is an object of the *Root* class *TGMainFrame*. As such, it allows the programmer to add several

WriteFtd(channel, register); (enables *register* in *channel*)
gSystem->Sleep(n); (waits *n* milliseconds)
WriteFtd(channel, data); (writes *data* to *register* in *channel*.

Table 3: Sequence of commands to write data to the USB port

other objects to it, as necessary in the application. The second kind already represents objects added to the main window: the two menus labeled as *File* and *Run*.

Before any data acquisition is executed, the DPM must be properly powered and connected to the USB port in the computer. All program facilities are therefore disabled before this connection is checked. This directive is followed along the whole implementation: the objects and their availability are arranged in such a way that the data acquisition steps and possibilities are correctly followed, and no particular expertise is expected from the user. As shown in Figure 2, submenus pop up in the main window but they are not enabled before the required DPM configuration is accomplished. The menus, submenus and cascaded menus shown in the picture are obtained from the class *TGPopUpMenu*.

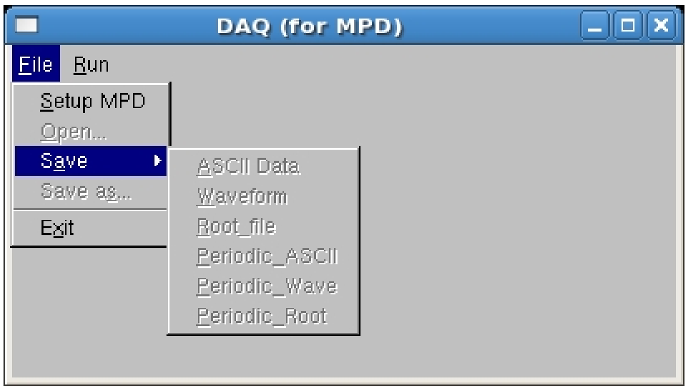


Figure 2: Submenus related to data saving and MPD configuration

The DPM configuration is done after the user clicks on the *Setup MPD* file menu item. This brings up another object, belonging to the *TGTransientFrame* class, which is just another window, or sub-window, where actions are taken and then it disappears from the screen. The *Setup MPD* sub-window is shown in Figure 3. Notice that all the data acquisition parameters related to the hardware are set in this transient window: the triggering mode (internal or external), the signal slope, the channel where the analog signal is sampled, the threshold level (used if internal trigger is chosen). Default parameters are provided, and may of course be modified by the user. Once he clicks on the *GO* button, the configuration is written to the DPM FPGA registers via the USB communication port. In case the communication is not available (either the DPM is not powered or not physically connected to the computer) a third sub-window appears and reports the failure. Notice as well that several objects are used in the *Setup MPD* sub-window: *TGCompositeFrame*, *TGTextButton*, *TGRadioButton*, *TG-GroupFrame*, *TGTextEntry*, *TGTextBuffer*, *TGLabel*, *TGComboBox*. The organization of the objects inside a frame is managed by an object of the *TGLayoutHints* class.

After the DPM is configured, the user may start data acquisition by clicking on the *Start* item of the *Run* menu, shown in Figure 4. In order to implement the configuration, the software makes use of the USB driver library functions, provided by the USB interface circuit manufacturer. Writing data to the USB port is done with a sequence of commands, as listed in Table 3.

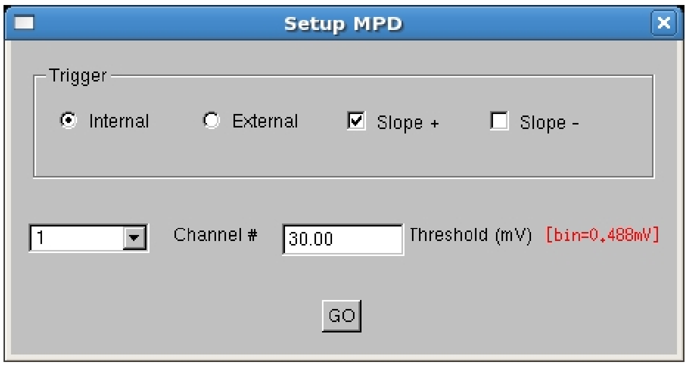


Figure 3: Sub-window for DPM configuration

Several USB ports may be available in the computer. The variable named *channel* refers to the one connected to the DPM. For each of them, *registers* are associated. If only one USB port is used in the computer, then we have *channel = 0*. The first command line in Table 3 activates one *register* in one *channel*. The next command should be writing *data* to the activated *register* in the available *channel*. However, it is recommended to introduce a time interval between these two commands, to make sure that the first one has been executed. This is done with the *gSystem->Sleep(n)* instruction, where *n* is given in milliseconds. For every register configuration, one sequence of the commands listed in Table 3 is executed.

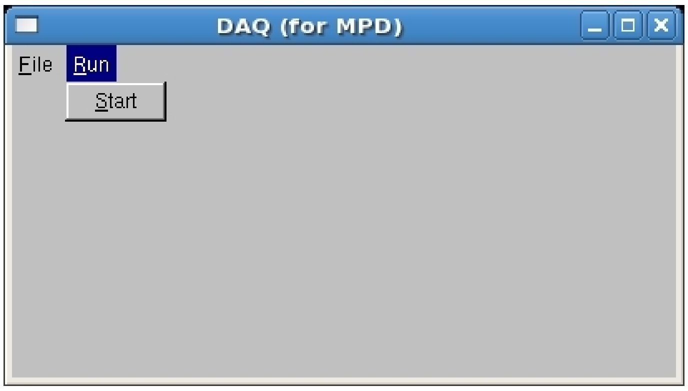


Figure 4: The *Run* menu, with the option to start data acquisition

When the user starts data acquisition, other objects composing the main window are made visible, as illustrated in Figure 5. The waveforms corresponding to the electric signals input to channel 1 in the DPM are shown in a graphic area. Other relevant information related to each acquired signal is also displayed in specific areas in the window. The objects related to data acquisition remain visible, until the user clicks on the *Quit* button. Then the main window is prompted again as it is when the program is launched (Figure 1), and the user may change acquisition parameters, re-start data acquisition, or exit the application.

Under the *Save* item of the *File* menu some options for data saving are accessible. Three kinds of files are prepared:

- The *.txt* file: An ASCII table with 8 columns, one line per detected event. The following information is written in each column: event number, time in seconds (since data acquisition start), the waveform baseline in millivolts, the amplitude in millivolts, the signal charge in pico-coulombs, the count rate in *Hz* (measured in a rate-meter circuit), the temperature in *C*, the count rate in *Hz* (measured inside the FPGA);

- The *.dat* file: An ASCII table with three columns, containing the waveform raw data. The first column indexes the sample number, the second and the third are respectively the time (in nanoseconds) and the amplitude (in millivolts). Each event fills 134 lines in the file, associated to a signal duration of 2.235μs in steps of 16.67 nanoseconds. Given the time when the signal is triggered, the transferred data contains one quarter of total number of samples before the trigger and the remaining three quarters after the trigger;

- The *.root* file. For each waveform a graphic object is created using the *Root* framework facility. These graphics are copied into a file that has the structure of a *Root* file [reference]. It may therefore be easily analyzed with the file management tools available in the *Root* framework.

Files may be continuously saved while data acquisition is going on, or they may be saved periodically. In the latter case, *.txt* files are saved every hour during a time interval of approximately five minutes; *.dat* and *.root* files are saved once a day, also for approximately 5 minutes. This is the current implementation, for taking data in the water Cerenkov detector at the container. It may of course be modified as necessary. The user only has to enable which kind of file has to be saved, by selecting it in a cascaded menu (see Figure 2). When this is done, a standard *Root* browser is prompted before data acquisition starts, so the file name and directory are specified.

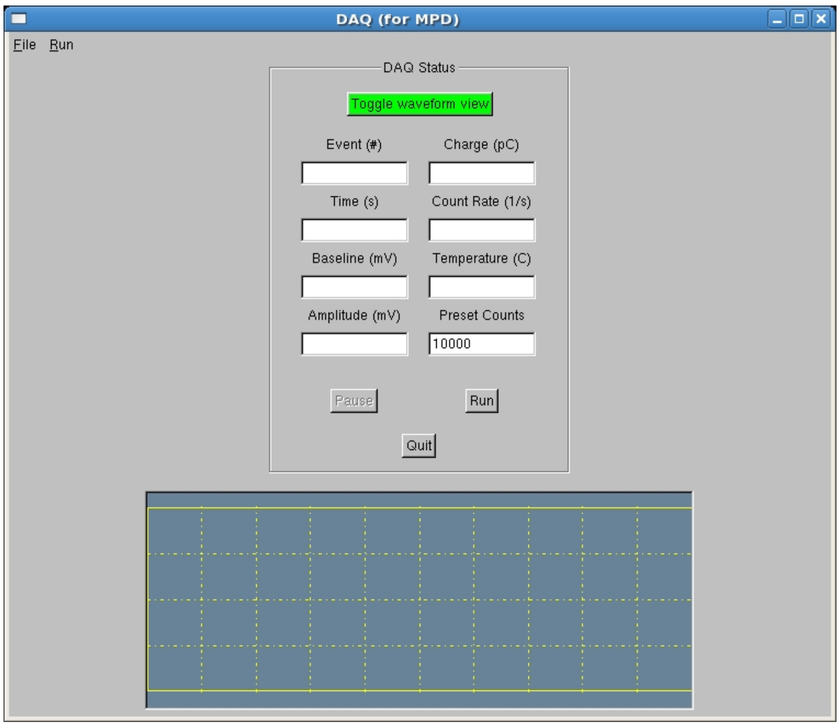


Figure 5: The data acquisition window

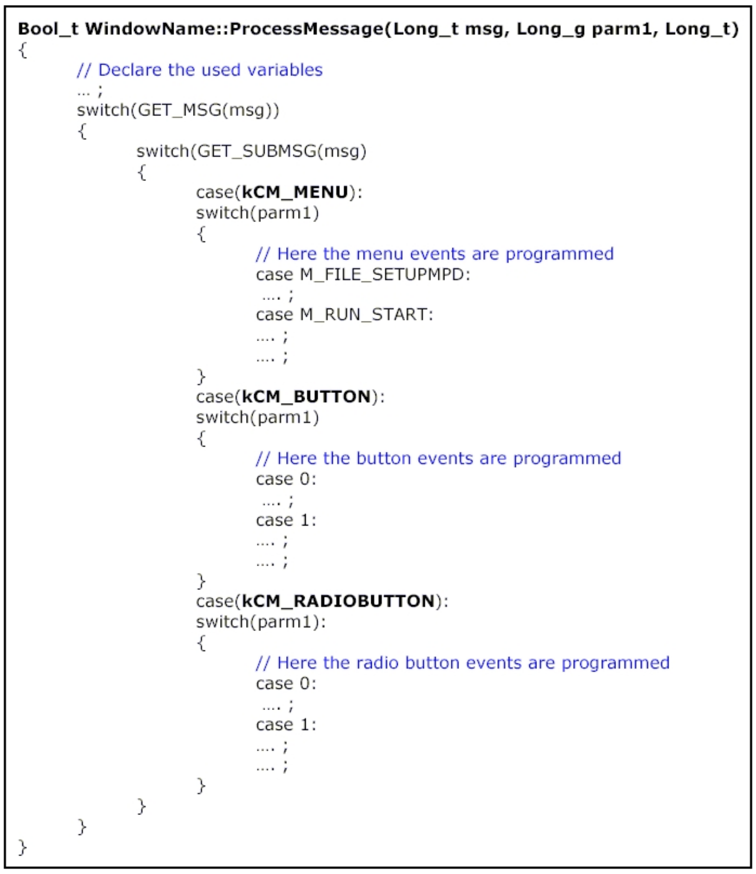
A button (seen in Figure 5) is provided in the data acquisition active window to toggle between three pulse display modes: pulse is shown with a grid and units; pulse is shown with a grid without units; pulse is not shown. The choice made by the user has impact on the dead time between acquired pulses. If the pulse is shown with the grid and units, some processing is required before the next pulse is accepted from the hardware, imposing a dead time between acquired events. The dead time is minimum when the pulse is not shown at all. Even though, in order to minimize the dead time, the pulse display is mostly done locally in the developed code (the *Root* graphic object is not used), still the display routine is time consuming. If maximum data transfer rate is required ($\approx 10^2$ pulses per second), the pulse should not be displayed. The local functions implemented for pulse display are listed in Table 2: *Draw_Pulse*(), *Draw_Grid*(), *Draw_Text*(). All the other listed local functions are executed for every acquired pulse. The data transfer rate capability may be improved by reducing the processing. For example, only the function to get the pulse amplitude could be executed. However, the data transfer rate would still be limited to $\approx 10^3$ pulses per second due to the maximum transfer rate allowed by the used USB version (1.0). This is why the detector counting rate information is stored independently by an analog rate-meter circuit and by an FPGA state machine. These two counting rate measurements are recorded in the *.txt* files.

3.1 Software events handling: the DAQ classes coding

Two different kinds of events have to be dealt with in the data acquisition software. One of these refers to actions taken by the user in the graphical interface level. We here call them software events. The other kind includes the events occurring in the detector, mainly originated by the detected particles. It could also be physical events such as an alarming power supply condition, an electronic failure, high temperature etc. We here call them hardware events, and describe their handling procedure in the next subsection.

Concerning the software events, their handling techniques are well developed in the programming framework. Starting from the main window shown in Figure 1, for example, the user may decide to click on different objects. Depending on his decision, the software has to provide the expected action, and each action is implemented in the code according to the object present in the clicked window area. In order to direct the program execution flow, command identifiers are established and listed in the main header file (*DAQ_Tank.h*). In the present application the command identifiers are defined as: *M_FILE_SETUP, M_FILE_OPEN, M_FILE_SAVE, M_FILE_SAVEAS, M_FILE_EXIT, ASCIIDATA, WAVEFORM, ROOTFILE, PERIODIC_ASCII, PERIODIC_WAVE, PERIODIC_ROOT, M_RUN_START*. These correspond to the possible items clicked in the menus, and they are identified under the *Root* variable *kCM_MENU*. In addition, other objects in the window, such as buttons, may also be clicked. To each of them is assigned a parameter (a number) in the declaration, so it is possible to identify which are clicked. The information provided by the user as text entries and others (e.g. *TGTextEntry, TGComboBox* item etc.) is, of course, used in the code implementation. In the present application main window, only menu and button objects trigger events in the software. To the latter is assigned the *Root* variable *kCM_BUTTON*. In the *Setup_MPD* transient window, another event is used, when the user clicks on the radio button, and this is associated to the *Root kCM_RADIOBUTTON* variable.

In the code implementation, the windows are declared and the objects contained in it are organized as desired. Then a special method (subroutine) called *ProcessMessage* has to be implemented for each window, in which the actions are programmed according to the expected events. The general structure of this special subroutine is shown in Figure 6.



```
Bool_t WindowName::ProcessMessage(Long_t msg, Long_g parm1, Long_t)
{
        // Declare the used variables
        ... ;
        switch(GET_MSG(msg))
        {
                switch(GET_SUBMSG(msg)
                {
                        case(kCM_MENU):
                        switch(parm1)
                        {
                                // Here the menu events are programmed
                                case M_FILE_SETUPMPD:
                                 .... ;
                                case M_RUN_START:
                                .... ;
                                .... ;
                        }
                        case(kCM_BUTTON):
                        switch(parm1)
                        {
                                // Here the button events are programmed
                                case 0:
                                 .... ;
                                case 1:
                                .... ;
                                .... ;
                        }
                        case(kCM_RADIOBUTTON):
                        switch(parm1):
                        {
                                // Here the radio button events are programmed
                                case 0:
                                 .... ;
                                case 1:
                                .... ;
                                .... ;
                        }
                }
        }
}
```

Figure 6: Scheme for software events handling using the *switch & case* tool

As seen in Figure 6, the software events are finally handled by use of the *C* language *switch & case* tool. It must be noticed, however, that this is not the only possible approach. Alternatively, the *C++ slot & socket* technique may as well be applied. In this case the code is distributed in several subroutines, instead of being concentrated as in the *ProcessMessage* method.

3.2 Hardware events handling: the DAQ thread

3.3 Compilation

4 Conclusion

The first experimental setup for taking data - in the context of the Angra Neutrino Project - from a detector placed near an Angra nuclear reactor has been completed. Although this will not be exactly the one used in the neutrino detector, it represents an important prototype from which much will be learnt and developed. The system will acquire relevant data concerning the local background and may as well be used in other applications - as, for example, cosmic rays measurements - not necessarily in the reactor area.

5 Acknowledgements

We are thankful to Thaynea Blanche and Leandro Amaral for their technical support, respectively on mounting the circuits and on the preliminary characterization data analysis. We also acknowledge invaluable support from Eletronuclear administration (in particular from Ant$\hat{o}$nio Carlos M$\acute{a}$zzaro, head of Angra-II) and technical staff (in particular Ilson Soares, Lenildo Domingues de Souza and Alu$\acute{i}$zio Mendes Santos Filho). This work has had financial support from FAPERJ and CAPES, brazilian agencies assisting scientific and technological research activities.

References

[1] I. Frank, I. Tamm. Doklady Akademii Nauk SSSR, Vol. 14, 109-14, Seriya A (1937).

[2] R. Machado. Centro Brasileiro de Pesquisas Fsicas, Msc. Thesis (2005).

[3] D. Alexander, K. M. Pathak, M. G. Thompson. J. Phys. A (Proc. Phys. Soc.), Vol. 1, Ser. 2, 578-583 (1968).

[4] I. Alekotte et al. Nucl. Instrum. and Meth. in Phys. Res., A, 586, 409-420 (2008).

[5] P. S. Allison, D. Barnhill. The Auger Project, GAP Note 2004-046 (2004).

[6] A. F. Barbosa. The Angra Neutrino Project, AngraNote 001-2007 (2007).