

Nuclear Model Codes Meeting Report

Coronado, CA - May 12, 2003

Abstract

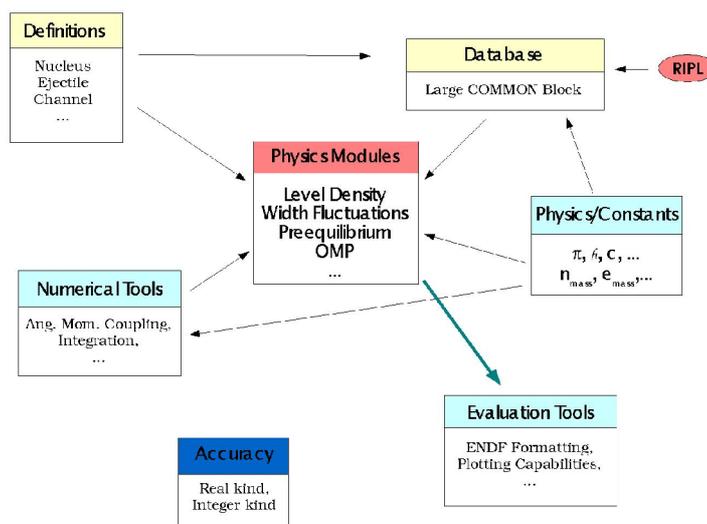
A general presentation of the modules library effort was presented by P.Talou at the beginning of the meeting, describing its goals, structure, individual module structure, mail/web communication tools, and a list of items to be discussed during the meeting. This presentation was followed by various more specific presentations on RIPL-2/3 (M. Herman), a new numerical tools module (T. Kawano), the width fluctuation module delivered earlier to the developers mailing list (P. Talou), and a brief presentation of the Livermore code MOARC (F. Dietrich). The rest of the meeting was devoted to the discussion on more technical topics. Various decisions have been made regarding the structure of the library, the structure of the individual modules, version numbering, test cases, etc (see detailed list in the report). Finally, a panel discussion allowed to provide an updated list of modules that are considered important to the developers community (this may change according to suggestions of developers not present at the meeting).

Detailed Report

Instead of going through each and every presentation held during the meeting, this report will try to describe each technical item for which a decision has been taken, or where further discussion among participants is needed.

Structure of the Library:

The overall structure of the library had already been discussed in previous meetings, but it was important to make sure that we all understand/embrace the same concepts and that we agree on a common structure. This structure is represented schematically on the following figure:



There are some "elementary" modules, such as **Accuracy** and **Physics/Constants**, which may be used by most other modules. More "complex" modules are the ones which deal with the physics

problems at hand, such as **Level Density** (Koning) and **Width Fluctuations** (Talou et al.). These modules should be as independent as possible from each other, ensuring that the modules dependencies remain quite simple.

The module **Definitions** contains "abstract" declaration of new **global** objects, to be used in various other modules. For example (as taken from Arjan's level density module),

```

type particle_type
  character(len=1)      :: symbol      ! symbol of particle
  character(len=8)     :: name        ! name of particle
  integer              :: Z           ! charge number of particle
  integer              :: N           ! neutron number of particle
  integer              :: A           ! mass number of particle
  real(single)         :: spin        ! spin of particle
  real(double)         :: mass        ! mass of particle
end type particle_type

```

The term "global" is very important; only global objects should be declared here. Other, more specialized objects, should be declared within the modules which use them. In fact, we may realize that the **Definitions** module is not necessary after all, and could be included within the **Database** module. This would remove an extra layer of dependency.

The module **Database** acts as a large F77 COMMON block where all **global** data can be stored and easily shared among other modules. In particular, this module will contain some concrete representations of the abstract objects declared in **Definitions**.

The **Numerical Tools** module (to be provided by Kawano) will include the most important numerical routines that are commonly used in our nuclear reaction codes. This module will certainly constantly evolve as the needs for new tools are recognized. A first set of routines has been established according to the needs already revealed within the scope of the first delivered physics modules.

We also decided that some tools to retrieve and calculate simple information from the **RIPL** database should become part of ModLib, i.e., they should be either reformatted (if they already exist) or created following the programming rules of ModLib. In fact, the RIPL-3 project is already dedicated to providing some of these tools, and it should therefore just be a "translating" problem to make them compatible with ModLib.

Finally, some so-called evaluation tools such as **ENDF** formatting and plotting capabilities could become part of ModLib.

Structure of an Individual Module:

We agreed that each individual module should be written in a standardized form. Not only submitted modules should follow some specific programming rules, but they should also have a standard header in order to render their use somehow straightforward.

As an example, here is the header for the Physics module (downloadable from the web):

<pre> ===== ! ! MODULE PHYSICS ! ! ! GOAL: Provides the values of physical constants as ! recommended in the ENDF-102 manual (Appendix H, April ! 2001 revision, Ed. V.McLane). ! !----- ! MODULE(S) REQUIRED: ACCURACY ! !----- ! USE: ! ! To call the module: ! ! > use PHYSICS ! !----- ! COMMENTS: ! ! Checked against 1998 CODATA recommended values. ! Ref.: J. Phys. and Chem. Ref. Data, Vol.28,#6, pp. 1713-1852, ! (1999). </pre>	<p>Name of the module</p> <p>Defines the purpose(s) of the module</p> <p>List all prerequisites modules</p> <p>Details on how to use the module</p> <p>More details on the structure, coding, limitations, use, ... of the module</p>
---	--

```

!           Rev. Mod. Phys., 72,#2, pp.351-495 (2000).
!
!-----
! RELEASE(S):
!
! Date          Release #   Author(s)      Comments
!-----
! 11/18/2002    0.1         P.Talou       Original version
!-----
!
! AUTHOR(S) INFORMATION(S):
!
! Release 0.1:   P.Talou
!               T-16, Nuclear Physics Group
!               Los Alamos National Laboratory,
!               Los Alamos, NM 87545, USA.
!               talou@lanl.gov
!=====

```

Release information

Author(s) information

Obviously, this module is elementary and only limited information is needed for its description, in particular the section on structure, coding, limitations, etc. However, the main structure can be regarded as a template.

As already mentioned, the coding itself should follow some standard programming rules, already partly described on our web site. Eventually, we could come up with a specific document expliciting in detail the different rules we agree upon.

Regarding such programming rules, the idea of using SPAG configuration files has been raised.

Standard Documentation:

Each module should be delivered with an adequate documentation in the form of a separate file in a commonly readable format (simple test, pdf, ...). This documentation should describe the physics, coding details, input/output, etc.

Test Cases:

At least one sample should be provided with each module. This sample will contain a driver code, and its input and output files, so that potential users can quickly test the module and compare their results with the ones provided.

Interfaces:

Interfaces between the modules themselves and between a module and existing reaction codes is a very important topic, since it is required to ensure that there is no conflict in variables and routines naming, and that the modules from the library can be easily and safely incorporated in our already well developed reaction codes.

Here are some suggestions made mainly by T. Kawano (thanks!):

- Variables naming: here, we refer to variables which have to be passed as arguments to/from the module from/to the driver code. As an example, the π constant defined in the Physics module could be named:

PH π ,

the two first letters referring to the module itself (PHysics), and the rest of the letters corresponding to the variable itself. Similarly, we will define PHhbar, PHclight, ...

- Routines naming: functions and subroutines used at the interface of the module should also follow a similar naming idea. For example,

NTGauss**L**aguerre

corresponding to the Gauss-Laguerre integration subroutine from the Numerical Tools (NT) module. Again, the two first letters design the module itself, while the rest corresponds to the routine. A

noticeable difference from the variables naming is the capital letter for the first character of the routine.

Makefiles:

Makefiles are small scripts which can render the installation/compilation/testing of the modules very easy and straightforward. While working only under UNIX like systems, they are certainly worthwhile (since most developers use UNIX or Linux anyway). A template will be provided shortly, and should be available from the web.

C/C++ vs. F90 interface:

During the Geel meeting in 2002, it was decided that the Fortran 90 is the language of choice for this new library of modules. However, it was also stated that we would accept modules written in C/C++, as long as they are compatible with the rest of the modules. According to T. Kawano, compatibility between C and Fortran (at least Fortran 77, not sure for F90) is quite feasible as long as we use only standard IEEE Single and Double precisions. I personally believe that mixing languages is not a very good idea, though we may not have much choice about that. Nothing was really decided about that, except than just waiting to see what comes up. That might just be the best solution...

Single/Double/Real Kind:

This question does not seem to have been settled after all. Should we go with only one Real Kind definition, or two definitions, basically corresponding to the IEEE standard Single and Double? A solution is to have both Single and Double defined, and using by default the Double precision for any variable/function in the modules. In some cases (e.g., very large memory array), it might still be useful to define variables in Single instead.

Licensing:

This issue is certainly the most controversial and difficult to solve that we had to deal with. In fact, no definite answer has been achieved. However, all participants agreed that the main objective should be to freely and fully collaborate and exchange ideas/codes/... among themselves. Problems arise because of copyright issues inherited from each of our home institutes. The final word on this topic was that model developers should inquire independently in their own labs about various solutions for distributing codes (or modules- it might make a difference) outside the institutes.

Version Numbering:

Basically, each module will be attributed a triple digit number x.y.z as

Library . Module-major . Module-minor

The first number ('x') corresponds to a version of the library of modules. The second ('y') and third ('z') numbers correspond to the module itself. The 'y' number is used when major changes occur since the last revision. This means that physics and coding contents may have been changed significantly, and may not be compatible with earlier module versions anymore. On the contrary, the 'z' number corresponds to minor modifications of the code or/and the physics, and should definitely not affect the way the module is used from the external driver codes.

For instance, let's consider the **Physics** module:

Preliminary (beta) version:	1.0.1	
First official version:	1.1.0	
Minor revisions:	1.1.1, 1.1.2, etc.	
Major revision:	1.2.0	
New library/new module:	2.1.0	(or 2.2.0 ?)

The question mark on the last point corresponds to the following question: what do we do to the major module number when the library number is updated? Should it stay at x.2.0, or eventually x.2.14, or should it be reinitialized at x.1.0? At least, looking at the two modules numbers only, when the major number is upgraded, then the minor number is always reinitialized to 0. Hence, it might be natural to do the same thing for the major number of the module when the library number is upgraded.

Delivering of modules:

This will strongly depend on the licensing issue that is still to be resolved. Hopefully, the delivery will be done freely through the web.

Review process of the modules:

Obviously, all these rules that were agreed upon need to be applied to have any significant impact on the overall development of the Library. Therefore, a system of peer-review needs to be set up. In short, we could establish a review system similar to the ones in place for any technical journal. Modules are sent to one or more editors who then redirect the module to one or two persons for review. This review process would allow to check that the programming rules have been correctly followed (in this regard, the SPAG software may be of great value), that all required information (header of the module) is present, and that the module package is complete, i.e., that the module comes with an attached documentation, a test code with at least one input and output. The reviewer should also run the test case on his/her own machine, and compare the results with the output provided by the author(s). The goal of the review process **is not** to test the coding or/and the physics contained in the module. This would require too much time, and would never be complete anyway. If an error/bug is detected by a potential user later on, then this user should send a "bug report" to the author(s) or/and the developers list so that this bug can be fixed quickly in the next release of the module.

C. Dunford provided some valuable suggestions regarding this subject. In particular, he compared the successful management model of CSWEG to the one which could be applied to ModLib. Basically, the process review could take place in two stages:

PHASE	ModLib	CSEWG
1	SPAG Compile (several platforms) Assemble documentation Coding and Methods Review	STANEF CHECKR, FIZCON Review kit Microscopic data review
2	Testing of results	Integral benchmark testing

Since SPAG is already available at BNL, Charlie also volunteered to take care of the two first items of the Phase I (Thanks!). Comments ? Suggestions ?

High priority list of modules:

At the end of the meeting, we came up with a fairly long list of modules which could be provided or/and which are of interest to some developers. However, since a shorter list is definitely more useful, here is the final short list that was established from the discussions:

- DDHMS preequilibrium code: this code by M.B.Chadwick was already delivered to the community in Fortran 77. Its translation in Fortran 90 with standards adopted in ModLib will be a priority.
- Fission: a fission module was thought to be important by many developers. Such a module could encompass many reasonable formulations already present in legacy codes, and fueled by some special expertise from people like Lynn, Maslov, etc.
- ENDF: tools to perform tasks related to the production of evaluated nuclear data files are also quite important. Such tools could be slowly (but surely) incorporated in a F90 module.

- RIPL-3 contribution: some routines already developed or being developed within the RIPL-2/3 project could be incorporated in a F90 module following ModLib rules. In particular, the interface between RIPL database and ModLib is considered very important.

Milestones / Deadlines / Meetings / ...

Due to lack of time, and because of the absence of important developers, these items could not be really addressed during the meeting. Instead, I propose a list of items for every developer to think about and to return some ideas/suggestions/answers, so we can quickly come to a common ground.

Here is the list of questions/items to be debated:

1. Is the proposed structure for the header of a module ok to everyone? If not, suggestions welcome.
2. Is the overall structure of the library accepted? If not, suggestions please.
3. What do we do about C/C++ vs. F90 interfaces? Wait and See?
4. I propose that a template for Makefiles be defined, and posted on the web for easy download.
5. Single/Double precisions: should we come up with these two definitions (instead of only one real kind parameter), and by default choose them as the standard IEEE values?
6. Is the naming of variables/routines at the interface of the modules ok?
7. Version numbering ok? If not, suggestions.
8. Licensing question. Every developer should inquire in their own institutes what the offered possibilities are. Then, we should have a final discussion, either through email, phone or video-conference.
9. Review process: does that sound right to everyone? If not, suggestions welcome. If yes, then we should decide about one or two editors (who can then define the rules more precisely), and about several potential reviewers (I imagine all active developers ?!).
10. High priority list of modules. I believe the first modules already delivered in their preliminary version (Level density, Width Fluctuation) should be the first to be adapted to follow the accepted rules, and be submitted through the review process mentioned earlier. Then, is the new list of modules to be delivered alright? And of course, who should be the developers involved?

Finally, we should decide about some precise milestones to be achieved within some specific deadlines. Such a deadline could correspond to our next meeting (suggestions?).

P. Talou
T-16, Los Alamos National Laboratory
May 29, 2003