Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXIII, págs. 3169-3194 (artículo completo)
Graciela Bertolino, Mariano Cantero, Mario Storti y Federico Teruel (Eds.)
San Carlos de Bariloche, 23-26 Setiembre 2014

ON THE DESIGN BASIS OF A NEW CORE-LEVEL NEUTRONIC CODE WRITTEN FROM SCRATCH

Germán Theler

TECNA Estudios y Proyectos de Ingeniería S.A.
Encarnación Ezcurra 365, C1107CLA Buenos Aires, Argentina

Instituto Balseiro, CNEA-Universidad Nacional de Cuyo
Av. Bustillo Km 9.5, R8400 Bariloche, Argentina

Keywords: Nuclear Reactor, Neutron Diffusion, Core-level neutronic code

Abstract. In the study, analysis and design of nuclear reactors there exist a wide variety of mathematical models that describe the different phenomena that take place in a nuclear facility. As in many other engineering fields, the corresponding equations are rather complex and require a considerable amount of both user expertise and computational effort to be successfully solved. Traditionally, there appeared some computer codes that specialized in solving a certain aspect of fission nuclear reactors such as neutronic codes, thermal-hydraulic codes, control system codes, plant codes, etc. Moreover, each discipline may be taxonomically split into further particular categories . For example neutronic codes can be aimed at lattice-level or core-level calculations, can use transport or diffusion formulations, etc. Since the dawn of the nuclear industry, a variety of codes have populated the universe of available tools we nuclear engineers have available to study, analyze and design nuclear reactors. In this article, the lessons learned in both the academia and in the nuclear industry during some years of experience are taken into consideration when defining the design basis of a new core-level neutronic code written from scratch, namely the free nuclear reactor core analysis code milonga. Some of the paradigm shifts both the hardware and software industries have had during the last years are considered into the way a modern engineering computer code should behave. The discussion includes the kind of problems that should be solved and the way the inputs are read and outputs are written. Also, implementation-related design decisions such as formats, languages and architectures are discussed. Illustrative problems are solved using the proposed project to serve as examples of desired features in modern and useful nuclear engineering codes.

1 INTRODUCTION

More often than not, when an engineer stumbles upon a new computational code designed to aid her to solve a certain problem, she starts to wonder why the program behaves as it does and not in another way. If she can contact the actual developer, usually the questions are translated into requests of changes and new features. Most of these questions can be rationally answered, even though some may have sound technical justifications and some may just inherit past design flaws that cannot be easily solved or improved—especially when dealing with legacy computational codes designed many years ago. This article discusses and explains the design basis of a new core-level computational code written from scratch. According to the experience gained by the author in the nuclear industry, in which he has worked both as a user and as a developer, it is very important to discuss, justify and clearly state the design basis over which such code is to be written.

The core-level neutronic code under study is named milonga, which even though it is usable in its current condition, should be considered still under development and not completely mature. The set of features that either are already included or are expected to be included into the code, and the details of the actual implementation are thoroughly discussed in the present work. The code is freely available under the terms of the GNU General Public License (following a decision that belongs to the design basis itself, as explained in section 2.1), so even if any of the features hereby discussed does not fulfill the reader's expectation, the actual design basis—and thus the code—can be modified at will following the spirit of free software, especially taking into account that most users of nuclear engineering codes are also hackers which are capable of reading, understanding and modifying source code.

The decision whether or not to include a particular feature into a computational code, or even the details about its implementation is not about performing the same calculation in a different way. It is about defining how the user works. For example, when performance is radically boosted say because the main computations are performed in GPUs instead of CPUs, not only does the code solve a problem faster but also the user starts to employ a different work flow, which hopefully supersedes the previous one. The design basis is therefore a very important subject that has to be discussed early in the development of a nuclear engineering computer code as it impacts on the actual projects that employ it even after the code is considered finished.

2 THE DESIGN BASIS

In the nuclear industry the term *design basis* is used when referring to ideas as design-basis accidents. Nevertheless, the term as used in this work is an approach usually employed by process engineers when conceptually designing industrial facilities, and refers to the *set of conditions, needs, and requirements taken into account in designing a facility or product*.

It is both illustrative and amusing to picture the design basis of a certain product (a computational code in this case) in the mathematical sense of a set of vectors which span a vector space of a certain dimension. Each vector may be thought of as a feature of the product having a certain magnitude and pointing in a certain direction. Usually, these basis vectors will form clusters pointing at similar directions. In particular, for milonga, four clusters are identified and separately discussed, namely

- the types of problem the code ought to be able to cope with
- how the input data is to be prepared by the user
- how the output results are to be written by the code
- details of the code's actual computational implementation

They all are addressed in the following four sections. There are, still, some vectors that will point at intermediate directions between such clusters. That is to say, some features may correspond to more than one of the four categories and some subjects may reference another feature that may no be already reviewed. Hopefully, by the end of the article all cross references should be consistently closed.

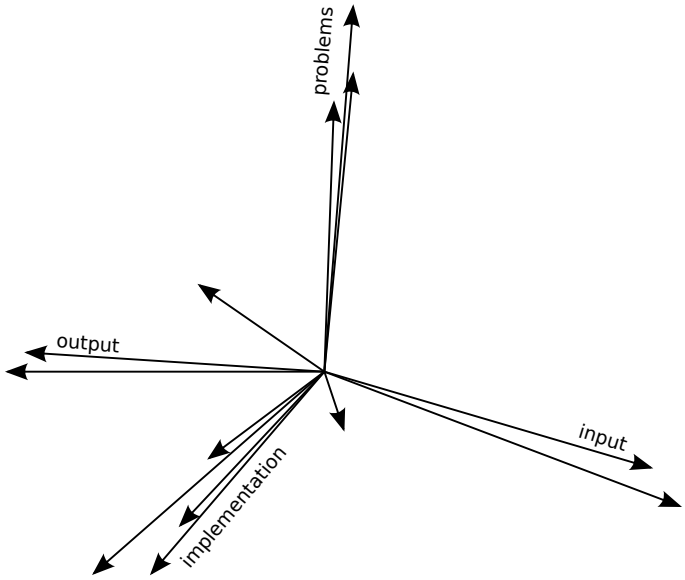


Figure 1: The design basis of a neutronic code amusingly depicted as a mathematical basis that spans a fictional n-dimensional vector space of features. Many vectors of the basis form clusters at some particular locations, while a few may point to a not-so-well-defined direction. We analyze the kind of problems the code solves, how the input is expected, how the output is written and details about its computational implementation.

2.1 Problems

First of all, it should be stated that the problems which should be tackled by milonga involve the core-level neutronics of a fission reactor (see Theler (2013c) for a brief description of the different kind of calculations involved in reactor analysis). That is to say, both power plants and research reactors should be taken into account. In principle, at least the steady-state multi-group neutron diffusion equation is to be solved. However, it is expected that a code such as milonga should be able to cope with either neutron diffusion or transport, under either the presence or absence of a neutron source within either multiplicative or non-multiplicative media. In addition, the ability to solve spatial kinetics may be desirable. Moreover, experience shows that more often than not, multi-point kinetic equations are enough to accurately model the spatial-temporal behavior of nuclear reactor cores. Therefore, the possibility to compute the coupling coefficients of multi-point models from steady-state results may be needed before implementing full three-dimensional kinetics equations.

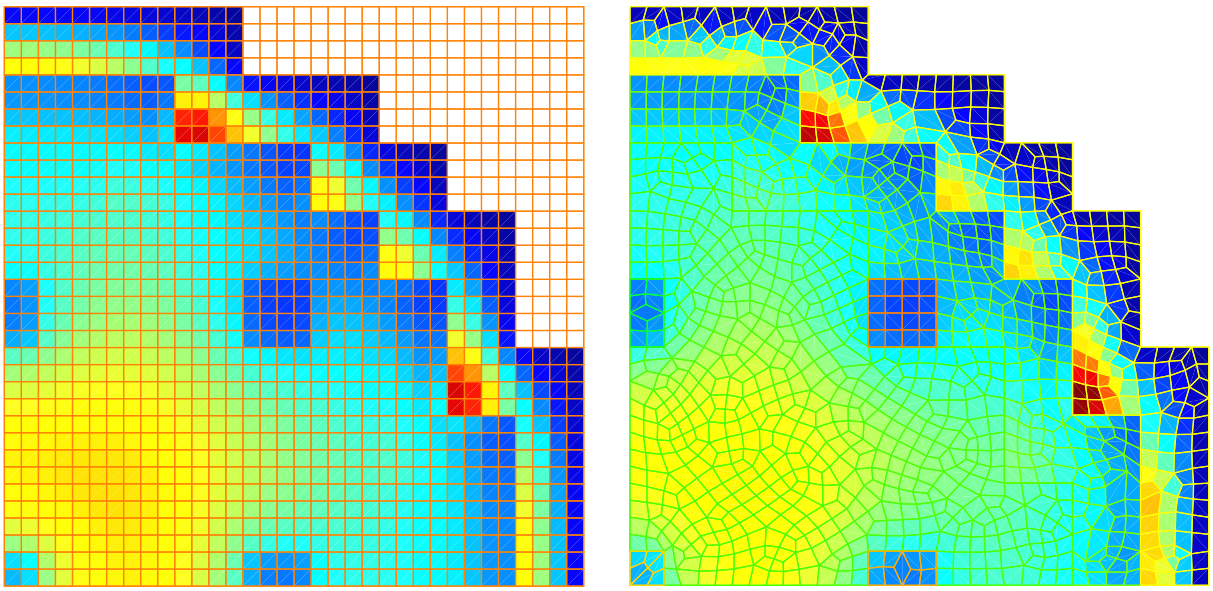
The enumeration and the description of the kind of problems that are the objective of a computational code is a subject of taxonomy, which almost always leads to incompleteness and or incorrectness. Nevertheless, it is useful to state the type of problems that ought to be able to be solved by milonga, in order of increasing complexity, as follows:

a) academic cases
   i. problems with analytical solution, i.e. bare homogeneous geometries with one neutron energy group
   ii. problems without analytical solution but simple enough to illustrate the physics, i.e one-dimensional reflected semi-homogeneous slabs with two energy groups

b) benchmark tests
   i. two and three-dimensional few-group problems with different materials (each one with uniform cross sections) and mixed boundary conditions
   ii. sensitivity studies using different meshes and numerical schemes

c) industrial problems
   i. full three-dimensional reflected geometry with an arbitrary number of energy groups using homogenized macroscopic cross sections that depend on the distribution of other properties (temperatures and densities, boron, xenon, control rods, etc.)
   ii. fuel management optimization
   iii. coupled transient operational and safety calculations

The ability to solve the problems listed in point b) is a suitable mechanism to verify and validate the code (ANSI, 2011), so it is a must. These problems require, on the one hand, the possibility to define several materials, each one with different cross sections; and on the other one, the flexibility to include one type of boundary condition over one part of the domain (such as null flux on external surfaces) and one over type over other part (null current on symmetry surfaces). See for example references ANS (1977); Mosteller (1997); Bernal et al. (2014). Anyway, these features may be better discussed in section 2.2 regarding the input of the code. As



(a) Structured grid as in Theler et al. (2011)

(b) Unstructured grid as in Theler (2013d)

Figure 2: Thermal flux of the 2D PWR IAEA Benchmark computed by milonga using finite volumes over (a) structured and (b) unstructured grids. In both cases the characteristic length of the mesh is $\ell_c = 5$ cm.

these benchmark problems are often designed to test different solution schemes, it is desirable for a code to be able to choose between different types of meshes (either structured or unstructured grids), basic geometries (triangles, quadrangles, etc), numerical schemes (finite volumes or finite elements) as illustrated in figure 2.

The third type of problems constitute the main objective of the code and, presumably, would span the majority of milonga's applications. In order for users to be able to build their own models and express them as input files for the code to understand and process (related to section 2.2 about input), a well-written and complete documentation set is mandatory. However, years of experience with industrial software packages show that good manuals—which, by the way, are hard to be found—are not enough, and thus access to the source code is needed. This requirement is purely based on a technical background, that comes from the necessity the final user—which is likely to be categorized as a hacker as defined by Richard Stallman (Vadén and Stallman, 2002)—has to fully understand the mathematical equations that the code is solving, to be able to debug, to control the computational implementation (section 2.4) and to even correct eventual programming bugs, which most computational codes have. Even if the user is not technically able to solve the problems she may encounter, a source-aware bug report is far more useful for the original developers than a simple statement saying "your code fails." These technical reasons are in the direction of Raymond's quotation "given enough eyeballs, all bugs are shallow," implying that if one wants to write a core-level neutronic code from scratch able to put up with current nuclear-industry scenarios, it should be *open source* as defined by the Open Source Initiative (1998). Besides, an open-source code may reach a great share of potential users due to the elimination of distribution fees and royalties.

Now, analyzing the type of problems in point a) it is evident that students need to interact with the code in a different way an expert with many years of experience in the nuclear industry does. A student that is learning not only neutron physics and reactor analysis but also advanced calculus and numerical methods, needs first to be able to run the code and then to understand it. Figure 3 shows an illustration of the effect known as "thermal shoulder" in a reflected slab. A student would completely grasp the physical background of such effect—and many others—by playing not only with the parameters in the input file (section 2.2, input) but also studying and even modifying the code's source to fully understand the mathematics behind it. Finally, based on an ethical background, she would also need to be able to share her findings with other classmates. It turns out, these four steps are exactly the four essential freedoms that *free software* provide. In words of the Free Software Foundation (2001),

> A program is free software if the program's users have the four essential freedoms:
>
> 0. The freedom to run the program, for any purpose
> 1. The freedom to study how the program works, and change it so it does your computing as you wish
> 2. The freedom to redistribute copies so you can help your neighbor
> 3. The freedom to distribute copies of your modified versions to others

Therefore, to successfully solve the type of problems listed in points a) and c) of the taxonomic list above, milonga should be distributed under a free-software compatible license. As a matter of fact, it was the lack of availability of free core-level neutronic codes back in the times when the author was an undergraduate student—that was later reinforced when working in the industry—that encouraged him to start a new code from scratch. Even freedom zero is not

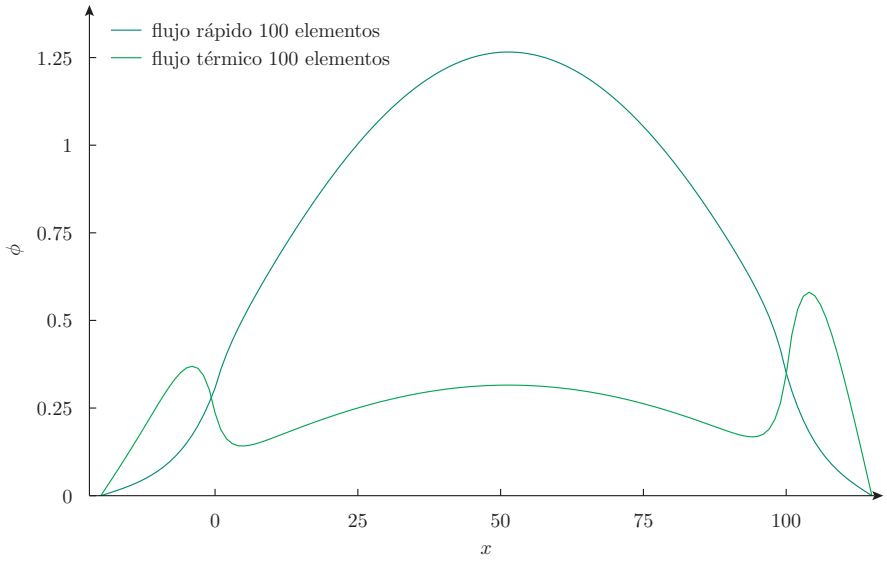


Figure 3: Illustration of the *thermal shoulder* effect on a reflected slab with two energy groups, as solved by milonga using finite elements. A student with access to such a code can play with the input to better understand the physics and to grasp the mathematical consequences of the non-separability of the flux near interfaces. Figure taken from Theler (2013a).

provided for many nuclear codes, still less the other three. Therefore, with the expectation of contributing with a grain of sand to the diffusion of scientific knowledge is that it was decided to release milonga under the terms of the GNU General Public License version 3.

Besides solving a problem and obtaining a single set of results, nuclear engineers usually need to study how these results change with respect to certain parameters. For instance, the computation of the reactivity worth of a control rod involves several calculations of almost identical cases which differ in a single parameter, i.e. the insertion of the control rod. The possibility of performing such parametric computations is certainly a desired feature. Indeed, milonga is able to sweep a certain portion of the parameter space and provide one or more results as a function of such parameters, efficiently performing sensitivity studies (figure 4) and building design maps (Theler and Bonetto, 2010).

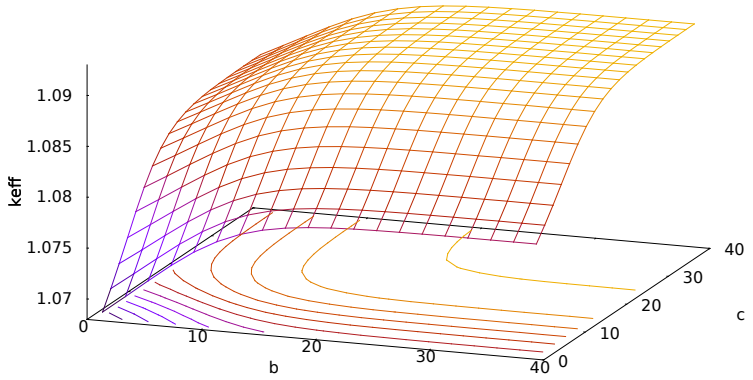


Figure 4: Effective multiplication factor k_{eff} of a slab of active width a with a left reflector of width b and a right reflector of width c, as a function of b and c (figure 5.40 of Theler (2013a)).

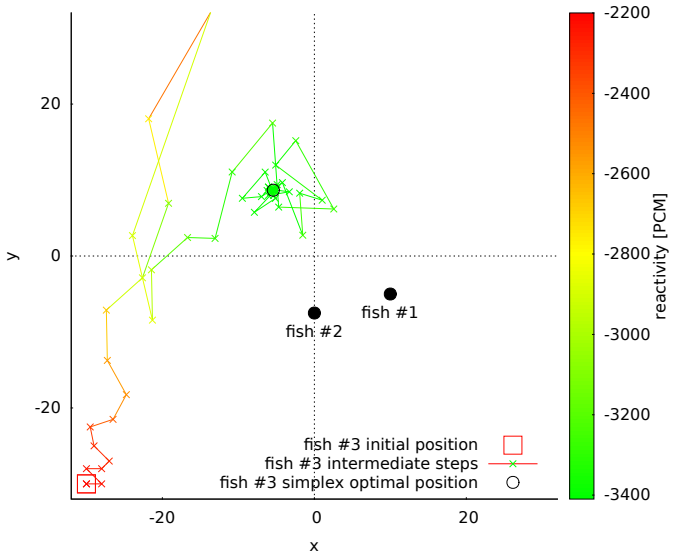


Figure 5: Milonga's solution of the "three-fish problem" in which two absorbing fish are fixed in a circular reactor and the location of the third one has to be found in order to maximize the negative reactivity insertion with the Nelder & Mead simplex algorithm. (Theler, 2013b).

Finally, it should be noted that optimization problems are a particular case of parametric studies in which, instead of varying the parameters in a certain pre-defined way—for example using quasi-random number sequences—a special algorithm tries to improve a certain energy function $E(\mathbf{p})$ by iteratively proposing a suitable sequence of vector of parameters $\mathbf{p}$. Figure 5 shows how milonga obtained the optimal location of a fixed-size absorber such that the overall reactor multiplication factor is minimum, i.e. where the absorber should be located in order to produce the maximum negative reactivity insertion (Theler, 2013b). By providing the ability to perform this type of calculations, a number of complicated issues related to nuclear reactor design and operation can be tackled. For example, non-linear optimization techniques such as simulated annealing or genetic algorithms can be applied to improve the efficiency of the fuel extraction burn-up in PHWRs reactors by studying and optimizing the refueling instructions given to the operators.

2.2 Input

In order for milonga to solve one of the problems discussed in section 2.1, all the needed data—including the reactor geometry, macroscopic cross sections, choice of numerical schemes, kind of computation the user wants and even the selected output (section 2.3)—should be completely defined in one or more input files, that are to be read by the executable binary program. That is to say, except for very technical cases of extensibility by means of dynamic shared objects (see section 2.4), it should not be necessary to re-compile the code to solve one problem or another, or to ask for a certain particular result with a given precision.

It was decided that such input file (or files) must be plain text containing keywords that take zero or more arguments, trying to maximize the self-description of the definitions and instructions contained in it. First, it is important to choose plain ASCII instead of binary data because of readability, scriptability, traceability and durability reasons (section 2.4 implementation). There is no technical background to prefer binary inputs for core-level calculations, especially

when working in an open and free operating system. Besides, control version systems such as Git or Bazaar may be used when preparing inputs for complex problems, simplifying at least the engineering coordination and management. Secondly, in the case of milonga, self-describing keywords were preferred to XML-based definitions because the following vector of the space basis should be kept in mind: *simple problems ought to need simple inputs*.

In effect, let us consider the canonical case of solving an homogeneous bare slab with one group of energy groups using a structured grid. To obtain both the effective multiplication factor k_{eff} and the flux distribution, the following input can be used:

```
NUMBER ncells_x = 10
NUMBER length_x = 100

MESH STRUCTURED DIMENSIONS 1 DEGREES 1
MATERIAL fuel D 1 nuSigmaF 2e-3 SigmaA 1e-3
PHYSICAL_ENTITY MATERIAL fuel X_MIN 0 X_MAX length_x

MILONGA_STEP

PRINT TEXT "\# keff = " keff
PRINT TEXT "\# x \t \t    phi"
PRINT_FUNCTION phi_1
```

When executing milonga with such input as an argument, the expected results are obtained in the standard output (see section 2.3)

```
 $ milonga slab-structured.was
# keff =        1.010678e+00
# x                phi
5.000000e+00    2.447174e-01
1.500000e+01    7.101976e-01
2.500000e+01    1.106159e+00
3.500000e+01    1.393841e+00
4.500000e+01    1.545085e+00
5.500000e+01    1.545085e+00
6.500000e+01    1.393841e+00
7.500000e+01    1.106159e+00
8.500000e+01    7.101976e-01
9.500000e+01    2.447174e-01
 $
```

which can be easily plotted using any free tool such as gnuplot or Pyxplot, that are tools that follow the rule of composition (section 2.4, implementation) themselves and as such, know that hashed lines are comments for human experts and are to be ignored by plotting programs.

Besides illustrating the "simple problem $\Longleftrightarrow$ simple input" feature, the example above shows that keyword-based text files are prone to syntax highlighting which improves the quality of the associated documentation (LaTeX) and eases file editing (milonga provides syntax highlighting for Kate and Geany). Also, plain-text output enhances scriptability and interaction with other tools, following the principles of UNIX philosophy (Raymond, 2003), which is further discussed in section 2.4 regarding milonga's implementation. In addition, plain-text files provide further flexibility for complicated problems by allowing complex inputs to be built using macro languages such as M4.

In general, a computational code provides many parameters and options that control the calculation—such as tolerances, numerical schemes, boundary conditions, etc.—which may be tweaked by the user in order to have full control of the results. To be able to solve simple problems with simple inputs, the code should have good default values for all the parameters and options, so the size of the actual input file is kept to a minimum. For example, the bare slab input above does not define any boundary conditions, so milonga assumes null flux at both ends

of the slab. Neither a convergence tolerance nor a spatial discretization scheme are specified, therefore default suitable values are used. Every parameter and option should default to an educated guess, yet the possibility of changing it has to be provided should the problem need it.

The keyword-argument approach to define the input file helps to follow the rules of clarity and least suprise (section 2.4). In effect, let us consider now the following two lines which may be found in an input file:

```
FUNCTION f(x) FILE_PATH f.dat INTERPOLATION akima
PHYSICAL_ENTITY NAME external BC robin -0.4692
```

From a programming point of view, it is far more simple to make a correspondence between the available function interpolation methods and the types of supported boundary conditions with integer values. However, the usage of explicit strings such as "akima" and "robin" should—in most cases—avoid forcing the user to recur to the documentation to find out what a certain integer flag means. In the same direction, position-dependent parameters should be avoided and self-describing keywords should be used instead. For example, in milonga, an auxiliary matrix named A may be defined by entering the line

```
MATRIX A ROWS 3 COLS 4
```

which may be compared to a fictitious (but not extraneous) code that uses numerical cards instead of keywords and position-dependent parameters:

```
20500100 3 4
```

In the latter case, the user has to refer to the manual to find out first what the numerical card 20500100 means and then, to see which of the two arguments is the number of rows and which one is the number of columns. The milonga approach is based on the concept of *syntactic sugar* (Raymond, 2003, chapter 8), in which the syntax of a definition may seem to contain redundant information from a computational point of view but eases the analysis from the human's perspective. Moreover, if the user does not need to keep going back and forth to the documentation in order to write a reasonable simple problem, then it is said that the design is *compact* (Raymond, 2003, chapter 4)—which is a desirable feature for a code like milonga.

Another feature that is really helpful for a computational code as a neutronic tool to have is the ability to parse and evaluate algebraic expressions whenever a numerical data is needed. For example, the following is a valid input for milonga:

```
deltax = 1/40
VECTOR psi SIZE 3*40
```

Even better, named constants may be used to give more insight about what 3 and 40 refer to:

```
NUMBER ngroups 3
NUMBER ncells 40
deltax = 1/ncells
VECTOR psi SIZE ngroups*ndim
```

Adding support for useful functions (log, exp, cos, steps, ramps, etc.) and functionals (integration, derivation, root and minima finding, etc.) is straightforward, and heavily enhances not only input flexibility but also in the manipulation of computed results (section 2.3, output).

It was stated above that the problem may be formulated in one or more input files. The reason is that some data—typically the definition of the materials and cross sections—may be

shared amongst several cases and thus it is a good idea to allow inclusion of other files from the main input. Even more, the mechanism should allow conditional and recursive inclusion. For example, either one or another set of cross sections may be used, depending of a certain flag:

```
bol = $1

IF bol=1
 INCLUDE xs-fresh.was
ELSE
 INCLUDE xs-equilibrium.was
ENDIF
```

This particular example also illustrates another key feature that is highly desirable. Besides reading the problem definition from a plain-text input file, some particular parameters or options may be given at run-time in the command line. In the above example, the flag `bol`—which stands for *beginning of life*—shall be provided as an argument when executing milonga:

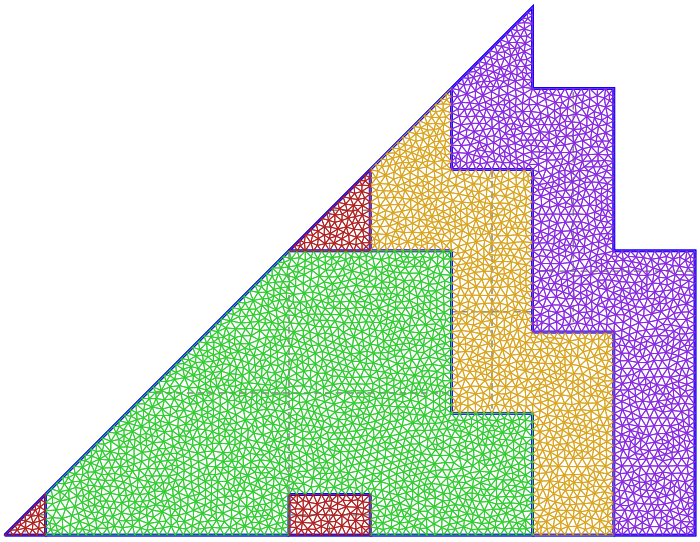
```
$ milonga problem.was 1
```

In the same way that solving a problem by reading an input file instead of hard-coding data into the executable avoids re-compilation, the ability to read some data from the command line allows to perform many similar computations without needing to prepare one input file for each case. For example, in reference Theler (2013d), the 2D IAEA PWR Benchmark problem is solved for eighty combinations of symmetry conditions, meshing algorithms, basic grid shapes, numerical schemes, and characteristic mesh sizes with a single input, by calling milonga from a Bash script with different arguments. Figure 6 illustrates case #048.

In general, it is better to avoid referring to generic numerical data as "tables" because this meaning changes with the context. The code should encourage to use well-defined mathematical entities such as functions of one or more variables, vectors, or matrices, according to the proper usage. For instance, the following input

```
# fission XS table as a function of burnup for temperature T0
# burnup in MW-day/tonU and XS in 1/cm
FUNCTION nusigmaf0(q) INTERPOLATION akima DATA {
0       0.205
1200    0.214
5000    0.183
8000    0.155
10000   0.110
}

# base fuel temperature [K]
T0 = 700

# coefficient of variation with temperature
c = 3.5e-3

# absorption XS as a function of burnup and temperature
# given in a data file
FUNCTION sigmaa(q,T) FILE sigmaa.dat INTERPOLATION rectangle

# definition of material "fuel" and its XS [1/cm]
VECTOR vec_x  SIZE ncells
VECTOR burnup SIZE ncells
VECTOR temp   SIZE ncells
FUNCTION burnup(x) VECTORS vec_x vec_bu
FUNCTION temp(x)   VECTORS vec_x vec_temp

MATERIAL fuel {
  D_1          1.5
  D_2          0.4
```

milonga's 2D LWR IAEA Benchmark Problem case #048
eighth-symmetry core meshed using delaunay (triangs, ℓ_c =2) solved with finite elements

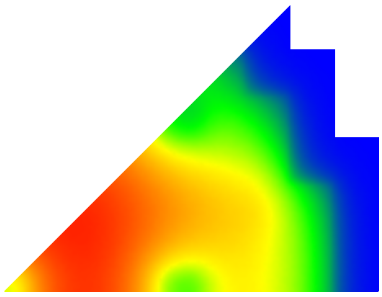


(a) Geometry and mesh: triangs with delaunay and $\ell_c = 2$

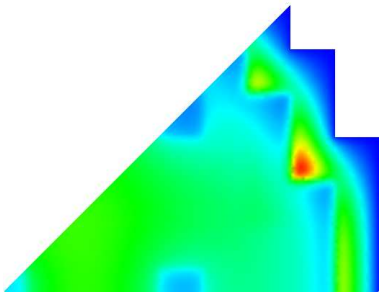
largest eigenvalue k_{eff}	1.029617 (2876.48 pcm)
$\max \phi_2(x,y)$@core	11.12 @ (30.93, 30.93)
$\max \phi_2(x,y)$@reflector	12.71 @ (131.15, 51.41)
number of unknowns	8156
outer iterations	3
linear iterations	32
inner iterations	1019
residual norm	1.89×10^{-12}
relative error	9.313×10^{-13}
error estimate	7.639×10^{-13}
memory used	158440 kB
soft page faults	40771
hard page faults	0
total CPU time	3.132 seconds

k	P_k	ϕ_{1k}	ϕ_{2k}	k	P_k	ϕ_{1k}	ϕ_{2k}
1	0.74	32.21	5.49	20	1.17	37.03	8.69
2	1.30	41.53	9.61	21	1.07	33.60	7.91
3	1.44	45.51	10.68	22	0.97	29.09	7.22
4	1.20	38.41	8.90	23	0.62	16.73	5.16
5	0.61	26.43	4.51	24	—	2.48	5.90
6	0.93	29.83	6.90	25	1.19	37.46	8.78
7	0.93	29.38	6.92	26	0.96	30.76	7.13
8	0.72	20.37	5.61	27	0.91	28.45	6.70
9	—	3.42	8.02	28	0.79	22.63	6.29
10	1.42	44.96	10.54	29	—	6.03	12.48
11	1.47	46.34	10.88	30	—	0.72	2.84
12	1.31	41.24	9.67	31	0.47	20.34	3.48
13	1.06	34.02	7.87	32	0.68	20.76	5.07
14	1.03	32.62	7.65	33	0.53	14.50	4.45
15	0.95	29.82	7.04	34	—	2.46	6.08
16	0.70	19.91	5.47	35	0.52	14.07	4.36
17	—	3.26	7.64	36	—	4.05	8.37
18	1.46	46.04	10.81	37	—	0.58	2.26
19	1.34	42.18	9.90	38	—	0.65	2.59

(b) Results with the finite elements method

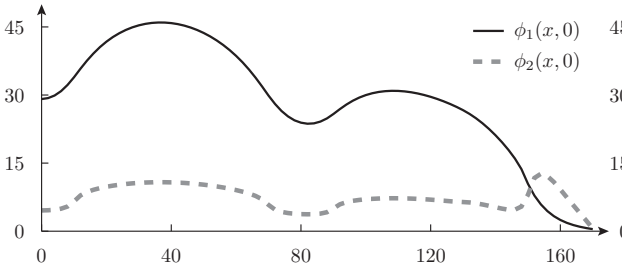


(c) Fast flux distribution

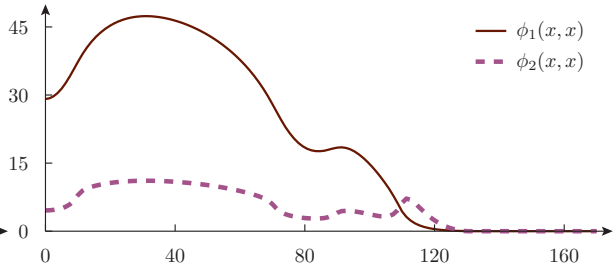


(d) Thermal flux distribution

(e) Power distribution



(f) Flux distribution $\phi(x,0)$ along the x axis



(g) Flux distribution $\phi(x,x)$ along the diagonal

Figure 6: One of the eighty solutions found with milonga of the 2D IAEA PWR Benchmark over structured meshes Theler (2013d,a)

```
    SigmaS_1->2  0.02
    SigmaA_1     0.01
    SigmaA_2     sigmaa(burnup(x),temp(x))
    nuSigmaF_2   nusigmaf0(burnup(x)-c*(sqrt(temp(x))-sqrt(T0))
}
```

illustrates how $\nu\Sigma_{f2}(x)$ is defined as an algebraic expression of interpolated data, as a function of a a certain temperature and burn-up distributions, which are functions of x. In turn, these distributions are defined by vectors which may be read at run-time from other files, shared memory segments or even from network sockets, allowing coupled calculations, that are being more and more necessary nowadays (Mazzantini et al., 2011; Theler, 2013c; Pinem et al., 2014). We will come back to this example later in section 2.3.

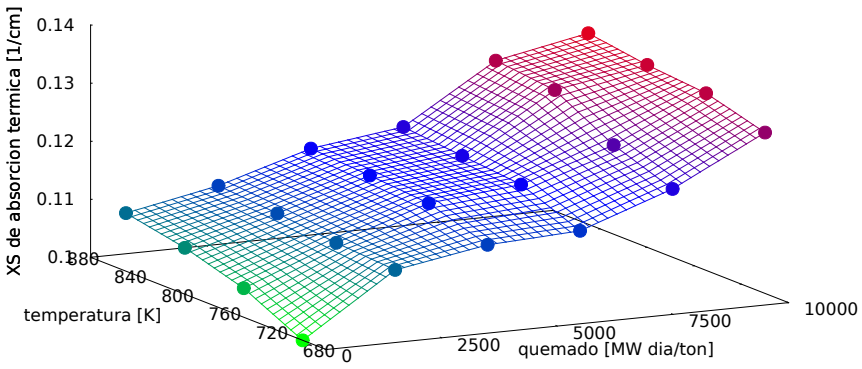


Figure 7: Fictitious macroscopic absorption cross section as a function of fuel burn-up and temperature given as a two-dimensional interpolation of scatted data (figure 5.48 of reference Theler (2013a)).

The last example also shows a great advantage of incorporating an algebraic parser into core-level neutronic codes: the dependence of macroscopic cross sections with parameter distributions (fuel burn-up, thermal-hydraulic properties, neutronic poisons distributions, etc) can be given in a rather flexible way. There is no need to pre-define that a certain cross sections depends linearly o quadratically with the temperature of certain component. Besides, on which temperatures and on which materials the cross sections depends can be completely different for each set. Even more, the proposed dependence can be written independently of the spatial discretization, allowing for even more flexibility. Multidimensional interpolation of scattered data (figure 7) provides even more power and greatly simplifies the introduction of cross section dependence through the spatial distribution of intermediate parameters.

It is important to note that by allowing to write cross sections as human-friendly algebraic expressions as in the example above, not only can the actual dependence form be arbitrarily chosen but also which parameters a certain cross sections actually depends on. Therefore, it is not needed to fix the computational code to a certain reactor technology as is the case where the fact that the moderator and the coolant is the same entity is hard-coded in software aimed at PWRs, preventing the application of such code to PHWRs. The flexibility of how the

macroscopic cross sections depend on the spatial coordinates x, y and z is a key feature of the code and affects many aspects of its usage and applications.

2.3 Output

Table 1 shows the cost and speed of some computers used in the 1960's for reactor analysis (Worlton and Voorhees, 1965). It is clear that since then, the costs of scientific and engineering projects have shifted from CPU time to human time, fact that is also known as the UNIX rule of economy (section 2.4, implementation). It is thus important to reduce the time human experts spend in performing simple and repetitive tasks, which by the way, are candidates to introduce errors which may delay the project and further increase costs and time (Knuth, 2003).

Back in the days of table 1, having to re-run a calculation because a result was missing in the code's output was catastrophic from an economical point of view. However, nowadays the aforementioned shift works the other way round: instead of writing as much results as possible in the output and generating huge amounts of data that need to be filtered and parsed by a cognizant human, it is better to just write what the user explicitly asks for—and nothing more. Not only does this feature simplify the post-processing phase but also enhances scriptability and follows the rule of composition (section 2.4, implementation).

The following example is taken from (Theler, 2013a, section 5.1.3.3) and compares the thermal flux peak factor, defined as

$$f_p = \frac{\max_x \phi_2(x)}{\frac{1}{a}\int_0^a \phi_2(x)\,dx} \tag{1}$$

of a two-group non-symmetrically reflected slab of active length a with two absorbers inside, for different spatial discretization schemes and mesh refinements:

```
# read the mesh file from the command line
MESH FILE_PATH $1 DIMENSIONS 1 DEGREES 2
SCHEME $2  # read also the numerical scheme (volumes/elements)

INCLUDE materials.was
MILONGA_STEP
```

Computer	Monthly Rental	Relative Speed	First Delivery
CDC 3800	$ 50,000	1	Jan 66
CDC 6600	$ 80,000	6	Sep 64
CDC 6800	$ 85,000	20	Jul 67
GE 635	$ 55,000	1	Nov 64
IBM 360/62	$ 58,000	1	Nov 65
IBM 360/70	$ 80,000	2	Nov 65
IBM 360/92	$ 142,000	20	Nov 66
PHILCO 213	$ 78,000	2	Sep 65
UNIVAC 1108	$ 45,000	2	Aug 65

Table 1: The new high speed computers (table 3 of reference Worlton and Voorhees (1965)). Costs are expressed in 1965 USD and may vary up to a factor of two. Relative speed is expressed with reference to IBM 7030. Data for computers expected to appear after 1965 was estimated.

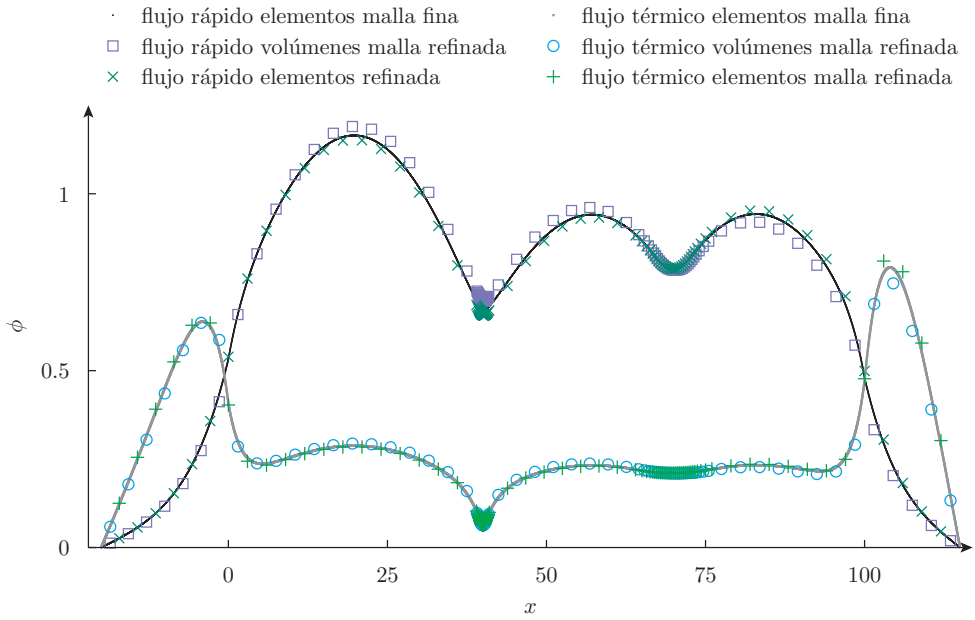


Figure 8: Flux distribution in a reflected slab with two absorbers inside. Comparison between a reference solution computed with a fine mesh and two numerical schemes (finite volumes and finite elements) using a coarse mesh refined at the absorbers. Figure taken from (Theler, 2013a, section 5.1.3.3).

```
a = 100

# first we look for the location of the maximum thermal flux
# (actually the minimum of -phi_2(x)) in the range [10,30]
xmax = func_min(-phi_2(x), x, 10, 30)

# the peak factor
f_p = phi_2(xmax)/(1/a * integral(phi_2(x), x, 0, a))

# show information in the standard output
PRINT "$1   " "$2 " %.1f 1e5*(keff-1)/keff %.2f xmax %.3f f_p

# and write the flux distribution in a text file ready to be plotted
OUTPUT_FILE flux $1-$2.dat
PRINT_FUNCTION phi_1 phi_2 FILE flux
```

A terminal mimic that illustrates how the problem can be solved follows:

```
$ echo mesh____________________method__________rho_____xmax____peak_factor > slab.txt
$ milonga slab.was slab-fino.msh volumes >> slab.txt
$ milonga slab.was slab-fino.msh elements >> slab.txt
$ milonga slab.was slab-sin-ref.msh volumes >> slab.txt
$ milonga slab.was slab-sin-ref.msh elements >> slab.txt
$ milonga slab.was slab-refinado.msh volumes >> slab.txt
$ milonga slab.was slab-refinado.msh elements >> slab.txt
$ cat slab.txt
mesh____________________method__________rho_____xmax____peak_factor
slab-fino.msh           volumes         -1163.3 19.72   1.239
slab-fino.msh           elements        -1163.5 19.74   1.239
slab-sin-ref.msh        volumes         -1032.0 18.00   1.270
slab-sin-ref.msh        elements        -1249.5 18.00   1.219
slab-refinado.msh       volumes         -1272.2 18.00   1.267
slab-refinado.msh       elements        -1115.7 21.00   1.224
$
```

This example illustrates many of the design basis features discussed so far, and some that are yet to be introduced (see reference Theler (2013a, section 5.1.3.3) for details about the example). Not only is the mesh computed by an specific tool (section 2.4, implementation), but the actual name of the mesh file is defined at run-time in the command line. The cross sections are included from another input file. The peak factor is explicitly computed by first finding the location $x_{\max}$ where the maximum thermal flux occurs and then evaluating equation (1) using explicit algebraic facilities provided by milonga. For each case, the actual output consists of a single line that states which mesh file was used, which numerical scheme was employed, the computed static reactivity in PCM, the location $x_{\max}$ of the maximum flux and the peak factor f_p, plus a separate text file containing both fluxes as a function of x written as a column-based ASCII representation of the numerical data. As can be seen in the terminal mimic, by calling the same input with different arguments, a chart that contains just the needed information—and nothing more—is obtained. Figure 8 illustrates the flux distribution of the slab as plotted by reading the separate text file with a proper tool (Pyxplot in this case). This economy of output also translates into economy of computation, because if by design a code writes as much information as it can, then it also has to compute as much results as it can. If a result is not asked for, there is no need to compute it, saving further time and effort.

In section 2.1 (problems) we stated that a wide variety of calculations such as parametric or optimization runs where a key feature of the code. This requirement can only be fulfilled by allowing arbitrary output instructions to be completely defined by the user in the input, as in parametric computations usually one or more results as a function of one or more arbitrary parameters are needed. Therefore, the only way to comply with the design basis of section 2.1 is to also implement the output of the code as discussed in this section.

Besides choosing which data is written into which file, the output of a computation program also involves how it interacts with other codes in coupled calculations (Mazzantini et al., 2011). This exchange of information may be done by accessing files (ASCII or binary), by interacting with network sockets or by means of the available IPC mechanisms. Amongst the last group, shared-memory objects synchronized using shared semaphores are the most convenient from an efficiency point of view (Theler, 2013c). Another approach that may be even more efficient in some cases is to implement particular calculation codes as shared libraries—i.e. as plugins—and then dynamically load as many codes as needed. This way, the user-space memory segment is effectively shared between the plugins and data exchange can be easily performed. Actually, milonga can be either compiled as a standalone binary or as a runtime-loadable plugin for the general engineering code wasora. Even more, the standalone executable of milonga is a particular case where both the basic wasora code and the milonga plugin are statically linked into a single binary, which may in turn load further plugins. By writing other tools such as thermal-hydraulic or control codes as wasora plugins, information exchange can be performed almost transparently. In effect, let us consider again the example where absorption cross section of a one-dimensional slab is given as a function of burn-up and temperature. In the input file shown, the fuel's Σ_{a2} is entered as

```
SigmaA_2      sigmaa(burnup(x),temp(x))
```

As can be seen, even without looking at the manual, that both `burnup` and `temp` are continuous functions of x. For academic or benchmark problems (section 3, input) these functions may be given as algebraic expressions. However, for industrial applications, these functions are not trivial and ought to be given as discrete $(x_i, f(x_i))$ pairs—or in general as $n+1$-tuples

for n dimensions. One way of doing this is by entering the numerical data into the input file, as shown for the function `nusigmaf0(q)` of the same example. But to be able to perform transient computations, i.e. either by burning the fuel with the fission power the neutronic code computes or by reading the temperature distribution from a thermal-hydraulic code, these $n+1$-tuples have to be read dynamically. And, in turn, some data about the power distribution has to be written somewhere somehow.

There are many ways to resolve the described situation. Without entering into details, here is one solution where the burn-up is computed by milonga itself whilst the temperate distribution is read from an external code after exporting the power distribution, exchanging information using POSIX shared memory objects and synchronizing using shared semaphores:

```
VECTOR power_dist SIZE ncells

vec_x(i) = (i-0.5)*width/ncells
vec_bu(i) = 0 + integral_dt(pow(vec_x(i)))/0.2

SEM thermal_ready    WAIT
READ SHM_OBJECT temperatures vec_temp
MILONGA_STEP
power_dist(i) = pow(vec_x(i))
SEM neutronics_ready POST
```

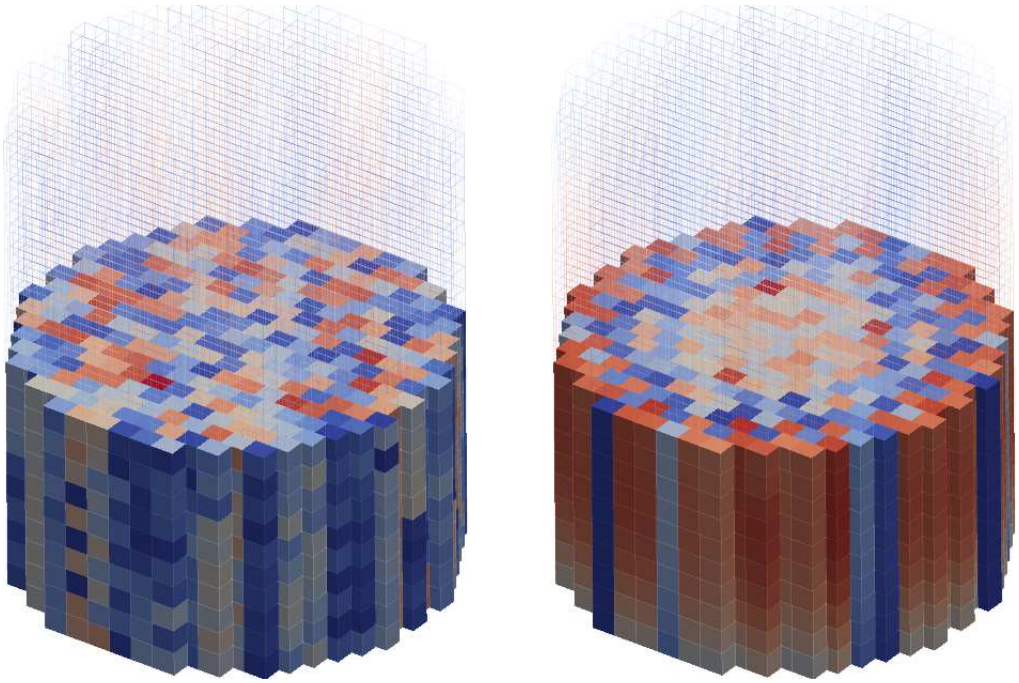
If the hypothetical thermal-hydraulic code had been coded as a wasora plugin (as milonga) that read the input data (power distribution) as one wasora vector (say `power_dist`) and wrote the result (fuel temperature distribution) as another wasora vector (say `temperature_dist`), then the last four lines would have had to be replaced by something like:

```
power_dist(i) = pow(vec_x(i))
vec_temp(i) = temperature_dist(i)
MILONGA_STEP
THERMAL_STEP
```

As discussed in reference Theler (2013c), this coupling scheme can be categorized as semi-implicit and the order at which the instructions are executed—both in the shared-memory and in the plugin cases—determines the convergence behavior. To sum up, as in section 2.2 (input), flexibility when writing data into different computational resources is a highly desirable feature.

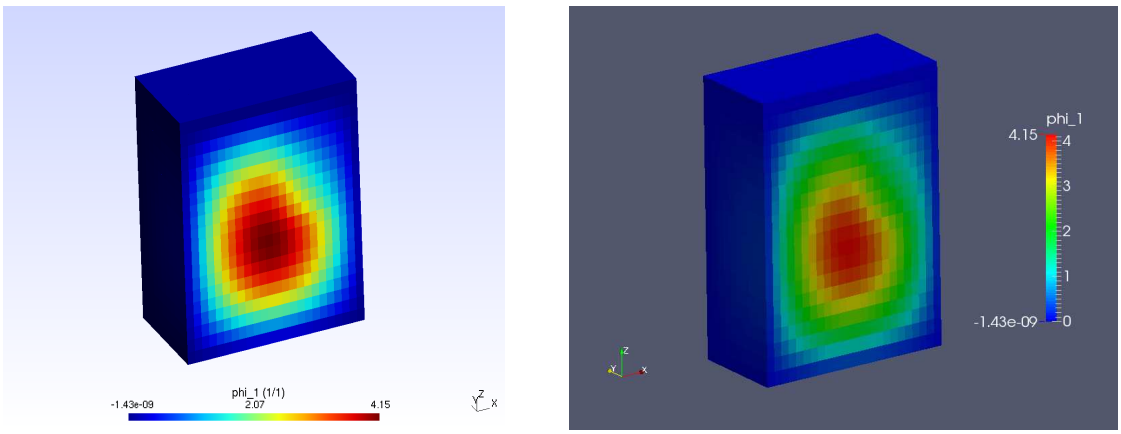
In general, the output of a core-level neutronic code consists of several scalar fields (group fluxes, fission power, delayed power, etc.) over up to four dimensions (three spatial and one temporal). It is important for the engineer to be able to correctly interpret, understand and post-process this huge amount of information to analyze the obtained results, especially in the early phases of the construction of the models where mistakes are more common. In this regard, experience shows that the usage of three-dimensional graphical representations of results can provide a great aid to detect errors in the discretized geometry, symmetries, input distributions, etc. that may be otherwise difficult to find (and to even realize there was actually an error in the model to begin with) by just looking at the numerical data.

For instance, when interfacing with other codes it is important to be consistent with the way collections of numbers are interpreted. A common example is to give properties in a matrix-like fashion with two-dimensional indexes such as (channel number, axial cell) dumped into files or shared-memory objects. However, the matrix data may be interpreted either as row-major (as in C) or as column-major (as in Fortran) orders. Asking the neutronic code to give a graphical representation of the data it reads, quickly allows the user to detect inconsistencies, as illustrated in figure 9 where the burn-up distribution is read from a file.



(a) Wrong (b) Right

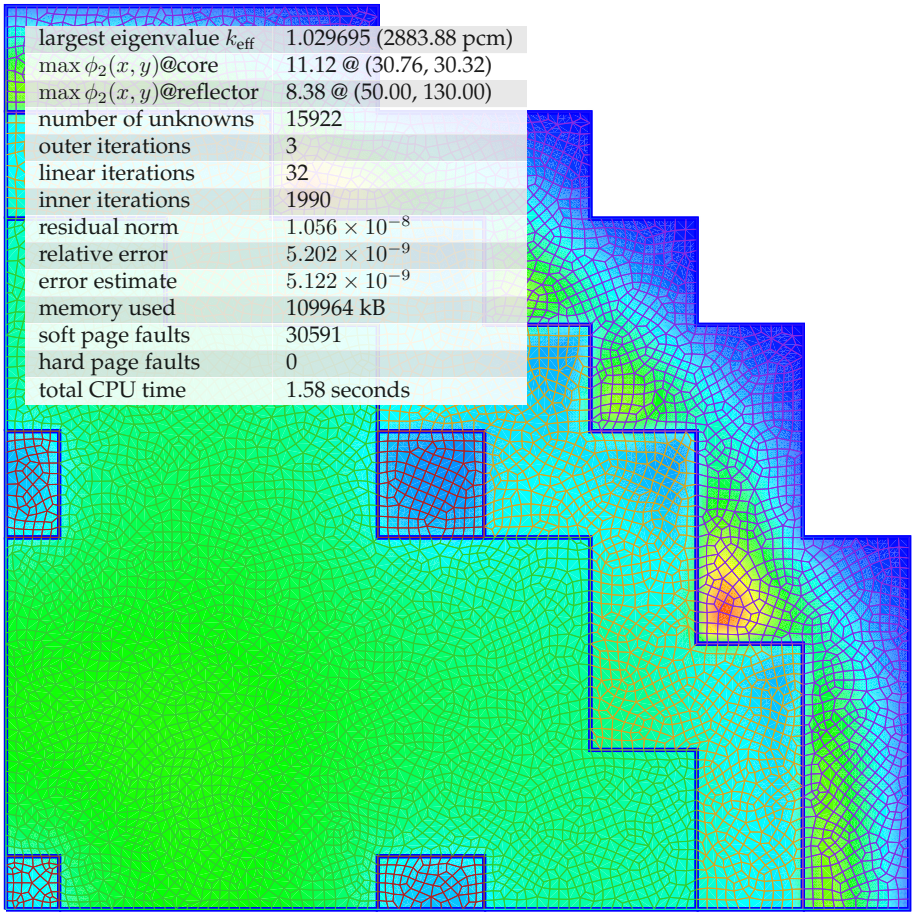
Figure 9: Axial cut view of the time-average fuel burn-up distribution as read from a file containing matrix data interpreted as (a) column-major order (b) row-major order. The graphical representation helps the cognizant engineer to quickly determine that the consistent interpretation for this case is figure (b).



(a) Gmsh (b) ParaView

Figure 10: Azimuthal cut view of the thermal flux distribution of a fictitious three-dimensional reactor with a non-symmetrical vertical control rod as computed by milonga and interactively represented by two three-dimensional free post-processing programs: (a) Gmsh (b) ParaView

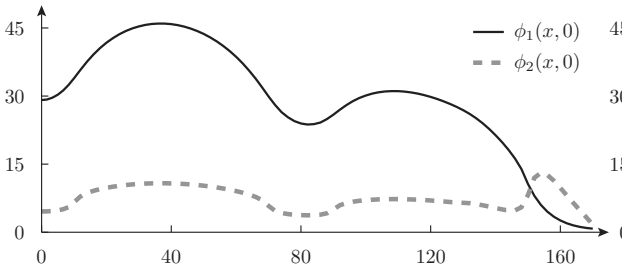
milonga's 2D LWR IAEA Benchmark Problem case #018
quarter-symmetry core meshed using delaunay (quads, ℓ_c =2) solved with finite elements



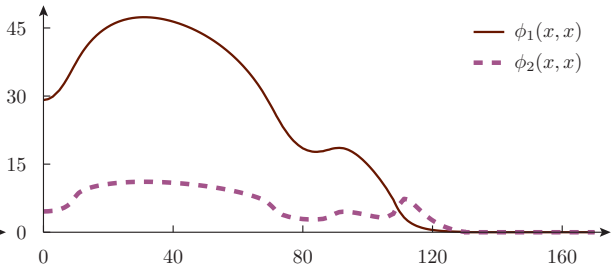
(a) Mesh and thermal flux distribution

k	P_k	ϕ_{1k}	ϕ_{2k}
1	0.74	32.20	5.49
2	1.30	41.52	9.61
3	1.44	45.51	10.68
4	1.20	38.43	8.90
5	0.61	26.48	4.52
6	0.94	29.96	6.93
7	0.94	29.59	6.96
8	0.72	20.61	5.69
9	—	3.58	8.62
10	1.42	44.95	10.54
11	1.47	46.34	10.88
12	1.31	41.27	9.68
13	1.07	34.09	7.89
14	1.04	32.75	7.68
15	0.96	30.03	7.08
16	0.71	20.14	5.54
17	—	3.42	8.20
18	1.46	46.06	10.81
19	1.34	42.22	9.91
20	1.18	37.11	8.71
21	1.07	33.75	7.94
22	0.98	29.29	7.27
23	0.62	16.92	5.22
24	—	2.58	6.32
25	1.19	37.52	8.80
26	0.96	30.84	7.15
27	0.91	28.58	6.73
28	0.80	22.80	6.33
29	—	6.11	12.77
30	—	0.80	3.18
31	0.47	20.43	3.50
32	0.69	20.88	5.10
33	0.54	14.63	4.50
34	—	2.55	6.48
35	0.51	14.18	4.39
36	—	4.10	8.54
37	—	0.64	2.52
38	—	0.71	2.85

(b) Power and fluxes



(c) Flux distribution $\phi_g(x,0)$ along the x axis



(d) Flux distribution $\phi_g(x,x)$ along the diagonal

Figure 11: Processing milonga's output with a Bash script that calls sed, awk, Gmsh, Inkscape and LaTeX can produce professional, accurate and aesthetically-pleasant results (figure 12 of reference Theler (2013d,a)

It should be noted that, as discussed in section 2.4 (implementation), the calculation code should merely interact with graphical tools by being able to write information in the corresponding format. It is a design flaw to try to incorporate graphical primitives into the code, because besides breaking the rules of composition, separation, parsimony and extensibility (section 2.4, implementation), chances are that any implementation of a graphical engine that a nuclear engineer may be able to write would be far less powerful than those designed by professional programmers. Figure 10 shows the same result computed by milonga—corresponding to a bare three-dimensional reactor with a single non-symmetric vertical control rod—being interactively post-processed by two different freely-available professionally-coded programs.

Other desired features that are better achieved by the fact that output is *completely* defined by explicit instructions in the input file are ease of scriptability, interaction with other programs and reducing the need of manual processing of the results (section 2.4). If, for example, a normalized power distribution with respect to the instantaneous power is needed, it is best to compute it within the neutronic code with something like

```
# write both the vec_power vector and each element vec_pow(i)
# divided by the total power (output results in two columns)
PRINT_VECTOR vec_pow vec_pow(i)/power
```

instead of importing the `vec_pow` vector into a spreadsheet and then applying some manual operations to normalize it.[1] The usage of such point-and-click intermediate steps is extremely prone to introduce errors and is highly discouraged (rule of generation, section 2.4). Should the output of a calculation code need further processing, automated and repeatable tools such as sed or awk ought to be preferred (section 2.4, implementation). As an example, figure 6 is one of eighty similar figures that were created automatically out of milonga's output by a Bash script using tools such as sed, awk, Gmsh, Inkscape and LaTeX. Figure 11 shows another variation, which includes a transparent alpha channel and allows arbitrary zooming of plots, as published in Theler (2013d). These two figures also illustrate that the output obtained using already-existing tools with years of development on their back may attain publication-grade quality, mainly because these programs were designed to achieve such goal, which may be very difficult to obtain by trying to hard-code graphical output routines into the neutronic code.

2.4 Implementation

The cluster that points to the implementation direction contains the most numerous and complicated basis vectors. These vector are related to almost any other feature discussed so far belonging to the other three sections. The rationale of this section is based on the seventeen rules of the UNIX philosophy compiled by Raymond (2003). Although a detailed description of the relationship of each rule with the design of milonga is out of the scope of this article, it is illustrative to quote them as described in chapter four of the original reference. Nevertheless, the reader is encourage to mentally apply and link all of them to at least one nuclear engineering computational code she had used in the past:

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.

[1] Incidentally, I dare anyone to try to open such spreadsheet file twenty years from now.

6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so code logic can be stupid and robust.
10. Rule of Least Surprise: In interface design, always do the least surprising thing.
11. Rule of Silence: When a program has nothing surprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to CPU time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs if possible.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for "one true way".
17. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

The basic idea of this cluster is to avoid programming features that other people have already developed, usually better than us. The canonical case is that of numerical methods: using available free and open general mathematical libraries which implement algorithms designed by mathematicians, coded by professional programmers reviewed by the academic community is by far a better decision than embedding tailor-made numerical recipes into our specific engineering codes. In particular, in steady-state multi-group reactor analysis, the main objective of calculation code is to build the fission and removal matrices from the data contained in the input file (as discussed in section 2.2) and to write the appropriate output as requested by the user (section 2.3). The actual solution of the generalized eigenvalue problem is best obtained by using available libraries which are already, revised, verified, tested and optimized. Specifically, milonga solves the eigenvalue problem using the free library SLEPc (Hernandez et al., 2005) that works on top of the framework for handling big and sparse matrices provided by the library PETSc (Balay et al., 2013). Not only are the rules of composition, separation and optimization satisfied this way, but also—and more important—the extensibility rule is. On the one hand, in the future—which may be here sooner than you think—mathematicians may come up with new algorithms that need fewer iterations to solve a certain problem. On the other hand, programmers may come up with new implementations of existing algorithms that run faster than today—for example by using GPUs instead of CPUs. These two features are easily attained by using libraries instead of hard-coding numerical routines into the neutronic code.

Another example of composition, separation, simplicity and parsimony is that of the discretization of the problem geometry. The generation of suitable meshes is an ubiquitous problem itself. It is not necessary to tackle this problem from within our calculation code. As with the already-discussed post-processing phase, the task of defining the geometry may result complex and error-prone, even when dealing just with structured meshes. Therefore, using an existing graphical geometry editor with mesh generator instead of programming an *ad-hoc* implementation is a very convenient design decision. At the same, time and as already discussed in section 2.3 (output), publication-grade quality figures such as 6 and 11 are almost impossible to obtain without interfacing with existing high-quality post-processing, vector graphics and documentation software such as Gmsh, Inkscape, gnuplot and LaTeX. Not only do these programs incorporate many years of development which allow their developers to pay attention to details that would escape to most of us (see for example Knuth (1984)), but they also follow the rules

of UNIX philosophy themselves, rely on further libraries and interface with other programs as well.

The rule of optimization is one of the most difficult to follow, because we engineers are entities that are constantly optimizing our environment and cannot stand inefficiencies. However, experience shows that more often than not, optimized code does not work out of the box as originally expected. Besides, the resulting source is usually so obfuscated that even the original programmer cannot understand it as shortly as only a couple of months after the first coding. Taking into account the rule of economy (cost shift from CPU time to expert time) and the related rules of simplicity, transparency and extensibility, simple understandable slow code is preferred to obscure complex fast code.

Also important for the case of milonga is the rule of representation, that recommends adding complexity into the data structures (we humans perform better at abstraction) and removing complexity from the logical algorithms (we humans perform worse at low-level computation). On the one hand, this last rule states that the programming language of a calculation code such as milonga should be one able to represent and handle complex data structures. On the other hand, in the preceding paragraph a core-level neutronic code was defined as a "glue" layer between the physical problem defined in the input file and the low-level numerical library that solves the generalized eigenvalue problem. These two conditions intersect at being C the right language for milonga (Raymond, 2003). Indeed, the standard hardware platform where core-level neutronic codes are expected to run in the next few years has converged to what it is called the *classical architecture* (Blaauw and Brooks, 1997). Namely: binary representation, flat address space, a distinction between memory and working store (registers), general-purpose registers, address resolution to fixed-length bytes, two-address instructions, big-endianness and data types set consistent with sizes that are multiples of 4 bits. This is exactly the hypothetical computer for which the C programming language was originally designed for by Kernighan and Ritchie back in 1973, which in part explains its success.

Even though in the mid 1950's the appearance of the FORTRAN-I language was a breakthrough that allowed technical staff—i.e. scientists and engineers—that were not expert programmers to be able solve very complex problems with comparatively little effort (Knuth, 2003), it has nowadays become obsolete and is only used when dealing with legacy snippets of code. In words of the PETSc developers (Balay et al., 2013)

> C enables us to build data structures for storing sparse matrices, solver information, etc. in ways that Fortran simply does not allow

Another choice would be C++, but its intrinsic complexity is not aimed at implementing the already discussed "glue layer" (Raymond, 2003). Again, PETSc developers state that

> Using C function pointers to provide data encapsulation and polymorphism allows us to get many of the advantages of C++ without using such a large and more complicated language."

Indeed, milonga (and wasora) make extensive use of function pointers. Besides, in Raymond's words,

> C++ is anti-compact—the language's designer has admitted that he doesn't expect any one programmer to ever understand it all.

Finally, sticking to standards is a good recommendation, especially when they are well designed and thoroughly thought. ANSI C is a complete standard that all modern C compilers support, and the language is identical on all machines. In particular, milonga makes extensive use of dynamic memory allocation, pointers to structures, linked lists and hashed tables—all features that either are embedded into the intrinsic design of the C Programming Language (Kernighan and Ritchie, 1988) or that can be easily implemented. Moreover, any modern operating system provides access to low-level system calls by means of a C-compatible API, which indicates that the language will be at least under consideration at least for many years from now.

The issue of scriptability is related to several of the rules, and is the principal way of embedding a computational code into a calculation environment that may include other calculation codes (lattice-level neutronics, thermalhydraulics, control systems, etc.) or further general tools (geometry editors, post-processors, plotting utilities). The two already-introduced figures 6 and 11 were produced by a script that called successively mesh generators, calculation codes, plotting utilities and document preparation tools. But even more, the whole chapter four of reference Theler (2013a) that spans one hundred and sixty one pages worth of inputs, results, figures and discussions of eight types of problems was generated by a single script. That is to say, should the chapter need to be re-generated (for example because a bug was found either in the code or in one of the inputs), all it would take is a single command. This feature, that amongst other benefits enhances repeatably, can be obtained by reading the input as proposed, by understanding command-line arguments (section 2.2, input) and by being flexible in the way results are written (section 2.3, output).

Milonga obeys the the extensibility rule by with a few number of characteristics. First, the introduction of hard-coded numerical schemes is minimized and the code relies on freely available libraries as already discussed. If the flexibility the code provides (algebraic expressions, function interpolation, conditional evaluation, etc.) is not enough to solve a certain problem, arbitrary user-provided code can be executed provided it is compiled into a dynamically loadable shared object. An open API is provided by the host code to be able to interface with wasora's objects (variables, vectors, matrices, functions, etc.). For example, user-provided eigenvalue solver can be loaded from a shared object to replace the one provided in SLEPc and compare its performance. Actually a Master's Thesis in Engineering employed milonga with a user-provided solver to study parallelization schemes of the solution of the steady-state neutron diffusion equation (Rivero, 2011). Another way of extending the code's capabilities is to load another wasora plugin (just like milonga itself is a wasora plugin) that interfaces with the Python libraries and provides the ability not only to execute Python instructions but also to map wasora variables and vectors back and forth to the scripts. And if this type of scalability is not enough, a new *ad-hoc* plugin can be written starting from the freely available template.

Experience has also shown how important the rule of repair is. From time to time, one or another calculation gets some kind of ill-conditioning and causes numerical problems such as overflows or divisions by zero. Even computational problems such as arrays out of bounds or invalid pointer arithmetic may occur. The code should try to warn the user and give her as much information as possible as where the problem occurred. Nevertheless, numerical issues such as domain errors are to be expected in parametric or optimization computations if the set of parameters are not physically consistent. Therefore, in some cases the appearance of a *not-a-number* is not necessarily an error. Following the "simple problem simple input" rule, milonga by defaults detects numerical errors and gives as much information as possible before politely quitting flushing caches and closing file descriptors. But there are options to ignore

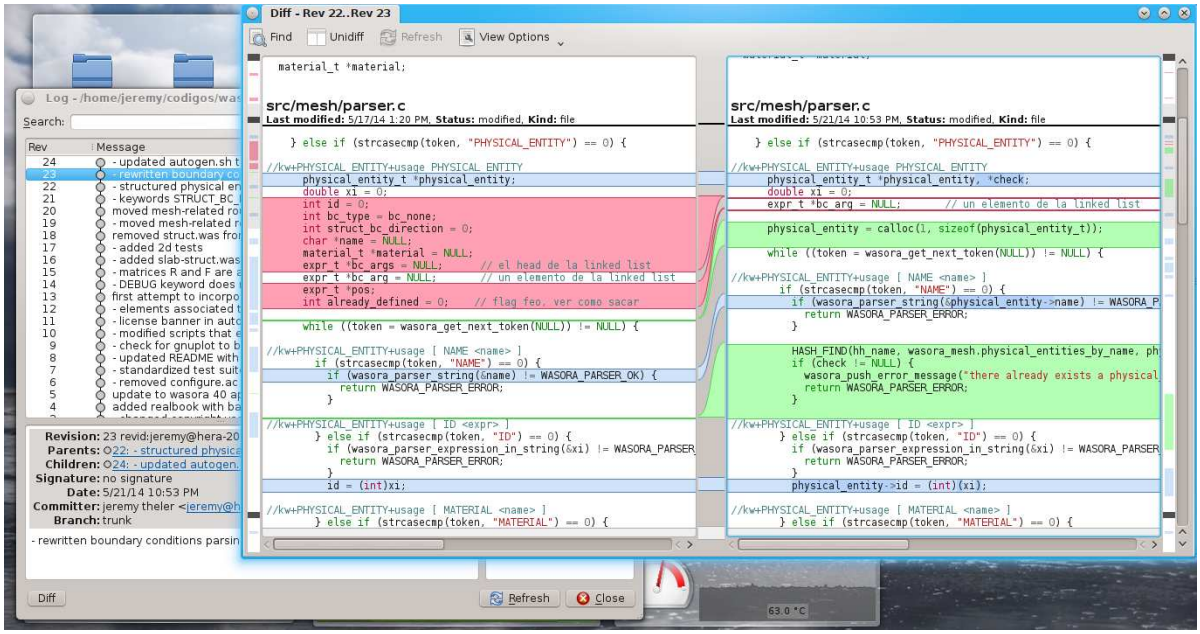


Figure 12: Graphical interface of the Bazaar control version system showing the revision history in the background and a side-by-side differential view of the changes introduced by a particular commit.

NaNs and to even not issue warnings.

Nowadays, using any kind of distributed control version system is a must for any serious software development project, especially for a free and open one as milonga where users are potential contributors. First, this way each user has the ability to have her own branch containing the full project history and can thus search and analyze for particular changes. And second, the process of incorporating and tracking user-provided fixes can be better managed by the original author by merging the corresponding branches. Distributed control version systems also encourage the "bazaar" development model as opposed to the "cathedral" mode, as discussed by Raymond (2001). Figure 12 shows a screenshot of the graphical user interface of the distributed version control system used by milonga—that coincidentally is called Bazaar—that visualizes the differences introduced into the code by a particular commit.

Not every bug fix or new feature needs to imply a new release, especially when the code has not been frozen yet—as is the case of milonga. However, control version systems—especially distributed ones—provide many mechanisms that allow the user to track an executable back to the actual source tree that was used to generate it, provided the hash signature of the tree is reported by the binary. For example, if the milonga executable is called with the -v option, it reports some useful information into the standard output:

```
$ milonga -v
milonga 0.2.21 trunk (2014-05-16 17:33:51 -0300 clean)
free nuclear reactor core analysis code

 branch gtheler@tecna.com-20140516203351-kn8p518jmxcv1evi
 last commit on 2014-05-16 17:33:51 -0300 (rev 21 clean)
 last build on  2014-06-14 12:57:49 -0300

 compiled on 2014-06-14 12:58:44 by jeremy@tom (linux-gnu x86_64)
 with gcc (Debian 4.8.3-3) 4.8.3 using -O2 linked against
  SLEPc Release Version 3.4.3, oct 14, 2013
```

```
  Petsc Release Version 3.4.3, Oct, 15, 2013  arch-linux2-c-debug
 running on Linux 3.2.0-4-amd64 #1 SMP Debian 3.2.57-3+deb7u2 x86_64
 8  Intel(R) Core(TM) i7 CPU         920  @ 2.67GHz

 milonga is copyright (c) 2010-2014 jeremy theler
 licensed under GNU GPL version 3 or later.
 milonga is free software: you are free to change and redistribute it.
 There is NO WARRANTY, to the extent permitted by law.

-----------          ---------       -------       ------      -----
wasora 0.2.130 trunk (2014-06-11 16:51:23 -0300 dirty)
wasora's an advanced suite for optimization & reactor analysis

 branch gtheler@tecna.com-20140611195123-xc1cw5yzca7w0e01
 last commit on 2014-06-11 16:51:23 -0300 (rev 130 dirty)
 last build on  2014-06-12 19:02:23 -0300

 compiled on 2014-06-12 19:02:51 by jeremy@tom (linux-gnu x86_64)
 with gcc (Debian 4.8.3-2) 4.8.3 using -O2 and linked against
  GNU Scientific Library version 1.16
  GNU Readline version 6.3
  SUNDIALs Library version 2.5.0

 wasora is copyright (C) 2009-2014 jeremy theler
 licensed under GNU GPL version 3 or later.
 wasora is free software: you are free to change and redistribute it.
 There is NO WARRANTY, to the extent permitted by law.
```

This way, given a particular binary executable or shared object plugin—that may be distributed as such to avoid having the end-user to compile and link the required libraries—the actual source tree can be found by branching from the central repository the revisions of milonga and wasora that correspond to the reported hash checksums. Also, one can then refer to the documentation of that actual revision—that in milonga is generated from particularly-formatted comments in the source code, as can be seen in figure 12—and know which syntax that particular version understands. This behavior is as much as the rule of clarity can be applied to binary files.

3 CONCLUSIONS

Throughout the present work many features that are desirable in a core-level neutronic code were discussed in detail, and particular examples of how they were included into the milonga code were illustrated. These features were grouped into four clusters, namely problems, input, output and implementation. After analyzing the problems milonga should be able to tackle, it was concluded that the code should be both open and free—terms that are not equivalent. The main conclusion is that for the industry it is important for users to access to source code so more eyes can make bugs shallower, whilst students and researchers benefit by studying, modifying and sharing source code. From the second group related to preparation of input, a basic rule that is expected to be followed by a code such as milonga is "simple problems ought to need simple inputs." This rule can only be obeyed by providing a compact syntax for the input file and good defaults for the calculation parameters. Also, the main feature expected from a core-level neutronic code is flexibility to enter how the macroscopic cross sections depend finally on the spatial coordinates x, y and z through intermediate distributions of an arbitrary number of properties (burn-up, temperatures, poisons, etc.). Regarding output, the basic conclusion is that the code should write only the results that the user explicitly asks for—and nothing more. Flexibility in the way data is written is also desired in order to allow interaction with other programs, whether they are other calculation codes or post-processing tools. As for the implementation details, the basic idea is to avoid programming features that

other people have already done, usually better than us. By following the principles of UNIX philosophy many years of experience in good programming practices can be easily included into scientific calculations, and particularly into neutronic codes to be used in nuclear engineering analysis.

REFERENCES

ANS. Argonne Code Center: Benchmark problem book. Technical Report ANL-7416 Supplement 2, Argonne National Laboratory, 1977.

ANSI. Steady-state neutronics methods for power reactor analysis. Technical Report ANS-19.3-2011, American Nuclear Society, 2011.

Balay S., Brown J., Buschelman K., Eijkhout V., Gropp W.D., Kaushik D., Knepley M.G., Curfman McInnes L., Smith B.F., and Zhang H. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

Bernal A., Miró R., Ginestar D., and Verdú G. Resolution of the generalized eigenvalue problem in the neutron diffusion equation discretized by the finite volume method. *Abstract and Applied Analysis*, 913043, 2014. Doi:10.1155/2014/913043.

Blaauw G.A. and Brooks F.P. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997.

Free Software Foundation. What is free software. 2001. https://www.gnu.org/philosophy/free-sw.html.

Hernandez V., Roman J.E., and Vidal V. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, 2005.

Kernighan B.W. and Ritchie D.M. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

Knuth D.E. *The TeXbook*. Addison-Wesley, 1984.

Knuth D.E. *Selected Papers on Computer Languages*. Center for the Study of Language and Information, 2003.

Mazzantini O., Schivo M., Di Cesare J., Garbero R., Rivero M., and Theler G. A coupled calculation suite for Atucha II operational transients analysis. *Science and Technology of Nuclear Installations*, 2011:785304, 2011.

Mosteller R. Static benchmarking of the NESTLE advanced nodal code. *Proceedings of the Joint International Conference on Mathematical Methods and Supercomputing for Nuclear Applications*, 2:1596–1605, 1997.

Open Source Initiative. The open source definition. 1998. http://opensource.org/docs/osd.

Pinem S., Sembiring T.M., and Liem P.H. The verification of coupled neutronics thermal-hydraulics code NODAL3 in the PWR rod ejection benchmark. *Science and Technology of Nuclear Installations*, 2014:845832, 2014.

Raymond E.S. *The Cathedral and the Bazaar*. O'Reilly, second edition, 2001.

Raymond E.S. *The Art of UNIX Programming*. Addison-Wesley, 2003.

Rivero M. *Optimización computacional de la solución numérica de la ecuación de difusión de neutrones*. Tesis de la Maestría en Simulación Numérica y Control, Univesidad de Buenos Aires, 2011.

Theler G. Difusión de neutrones en mallas no estructuradas: comparación entre volúmenes y elementos finitos. Academic Monograph, Universidad de Buenos Aires, 2013a. In Spanish.

Theler G. Geometric optimization of nuclear reactor cores. *Mecanica Computacional*, XXXII(32), 2013b.

Theler G. A shared-memory-based coupling scheme for modeling the behavior of a nuclear

power plant core. *Mecanica Computacional*, XXXII(18), 2013c.

Theler G. Unstructured grids and the multigroup neutron diffusion equation. *Science and Technology of Nuclear Installations*, 2014:641863, 2013d. Doi:10.1155/2013/641863.

Theler G. and Bonetto F.J. On the stability of the point reactor kinetics equations. *Nuclear Engineering and Design*, 240(6):1443–1449, 2010.

Theler G., Bonetto F.J., and Clausse A. Solution of the 2D IAEA PWR Benchmark with the neutronic code milonga. *Actas de la Reunión Anual de la Asociación Argentina de Tecnología Nuclear*, XXXVIII, 2011.

Vadén T. and Stallman R.M. *The Hacker Community and Ethics: An Interview with Richard M. Stallman*. Tampere University Press, 2002.

Worlton W.J. and Voorhees E.A. Recent developments in computers and their implication for reactor calculations. In *Proceedings of the Conference on the Applications to Reactor Problems*. American Nuclear Society, 1965.