Description of the computational tool wasora

Number: CIT-WSWA-TD-9E3D
Rev.: A
Date: 16-Mar-2016
Document type: Techical Document
Pages: 33

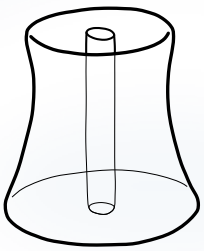
Author
G. Theler gtheler@cites-gss.com
Reviewed by
J. P. Gómez Omil jugomez@tecna.com
R. Vignolo rvignolo@tecna.com

Abstract

Wasora is a free computational tool designed to aid cognizant experts to analyze complex systems by solving mathematical problems by means of a high-level plain-text input file containing a syntactically-sweetened description of definitions and instructions. Some of its main features are:

- evaluation of algebraic expressions
- one and multi-dimensional function interpolation
- scalar, vector and matrix operations
- numerical integration, differentiation and root finding of functions
- possibility to solve iterative and/or time-dependent problems
- adaptive integration of systems of differential-algebraic equations
- I/O from files and shared-memory objects (with optional synchronization using semaphores)
- execution of arbitrary code provided as shared object files
- parametric runs using quasi-random sequence numbers to efficiently sweep a sub-space of parameter space
- solution of systems of non-linear algebraic equations
- non-linear fit of scattered data to one or multidimensional functions
- non-linear multidimensional optimization
- management of unstructured grids
- complex extensions by means of plugins

Besides solving general math problems usually associated with engineering analysis, the code is designed in such a way that particular (and potentialy complex) computations may be implemented as plugins (such as computations based on the finite element method or dedicated neutronic codes) that run over the framework, taking advantage of all the common background wasora provides. This technical document introduces the code and describes its main features by walking through the rationale behind its design and the types of problems that are suitable to be tackled with wasora.



Revision history

Rev.	Date	Author	
A	16-Mar-2016	G. Theler	First issue

Contents

1 Introduction

Wasora is a free computational tool that essentially solves the mathematical equations that are usually encountered in the models that arise when studying and analyzing engineering systems. In particular, the code history and the development team (see appendix B) is closely related to nuclear engineering and reactor analysis. Nevertheless, the code provides a number of basic mathematical algorithms and methods that make it suitable for solving problems in a wide variety of engineering and scientific applications, especially when dealing with dynamical systems. The main focus are parametric runs and multidimensional optimization of parameters that are themselves the results of the afore-mentioned models.

Even though wasora is a general mathematical framework, particular computations (such as specific finite-element formulations of problems or models of digital control systems) or features (such as real-time graphical outputs or the possibility to read ad-hoc binary data formats) may be implemented as dynamically-loadable plugins. The set of codes that comprise the wasora code plus its plugin is also known as the *wasora suite* (section 1.3).

The code is free software released under the terms of the GNU Public License version 3 or, at your option, any later version. Section 3 contains further details about the license of wasora.

1.1 What wasora is

Wasora should be seen as a syntactically-sweetened[1] way to ask a computer to perform a certain mathematical calculation. For example, the famous Lorenz system [1]

$$\begin{aligned}\dot{x} &= \sigma\,(y - x)\\ \dot{y} &= x\,(r - z) - y\\ \dot{z} &= xy - bz\end{aligned}$$

may be solved by writing these three differential equations into a human-friendly plain-text input file that wasora reads and solves when executed:

```
# lorenz' seminal dynamical system solved with wasora
PHASE_SPACE x y z
end_time = 40

# parameters that lead to chaos
sigma = 10
r = 28
b = 8/3

# initial conditions
x_0 = -11
y_0 = -16
z_0 = 22.5

# the dynamical system (note the dots before the '=' sign)
x_dot .= sigma*(y - x)
y_dot .= x*(r - z) - y
z_dot .= x*y - b*z

# write the solution to the standard output
PRINT t x y z
```

Listing 1: lorenz.was

```
$ wasora lorenz.was
0.000000e+00    -1.100000e+01   -1.600000e+01   2.250000e+01
2.384186e-07    -1.100001e+01   -1.600001e+01   2.250003e+01
```

[1] Quote from Wikipedia, *"In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language* sweeter *for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer".*

```
4.768372e-07    -1.100002e+01   -1.600002e+01   2.250006e+01
9.536743e-07    -1.100005e+01   -1.600004e+01   2.250013e+01
1.907349e-06    -1.100010e+01   -1.600008e+01   2.250024e+01
[...]
3.998879e+01    7.407148e+00    9.791065e-02    3.348664e+01
3.999306e+01    7.098288e+00    -6.613236e-02   3.310819e+01
3.999732e+01    6.795877e+00    -2.113808e-01   3.272946e+01
4.000159e+01    6.500405e+00    -3.390346e-01   3.235134e+01
$
```

Appendix D illustrate how wasora should be invoked in order to solve each of the example input files shown in this document. It also shows the output and associated figures and graphics built out of wasora's results.

As detailed in section 2, on the one hand wasora uses the UNIX idea of relying on existing libraries instead of re-implementing what other people have already done better. On the other hand, one of wasora's golden rule is "simple problems ought to need simple inputs." Therefore, it essentially consists of a high-level interface to low-level mathematical libraries so the final user can state the problem to be solved as simple as possible without wasting time and effort on unnecessary details. The example above should be compared with other ways of solving the Lorenz system, which may range from preparing a snippet of code to solve the equations (i.e. in C or Python) or using another computational tool (i.e. Octave or some non-free programs) that are not designed with syntactic sugar in mind as wasora is from the very beginning of its conception. For instance, in the example above, the parameters, initial conditions and actual differential equations are written in a natural way into a text file which is the read and solved by wasora. Moreover, the user does not need to get involved with tolerances or how to choose the time step in order to obtain convergence or other low-level details, although she may if she really needs to. This way, attention is paid to the part of the problem that is really important.

Wasora heavily relies on the GNU Scientific Library [2] to perform many low-level mathematical operations, including one-dimensional function interpolation, numerical differentiation and integration, one and multi-dimensional root-finding, random and quasi-random number generation, non-linear fitting and minimization, amongst others. A convenient high-level access to many of the features the library provides is given by wasora, as illustrated in the following two-line example that finds and prints the location of the minimum of the function $f(x) = \cos(x) + 1$ in the interval $0 < x < 6$:

```
VAR x
PRINT %.7f func_min(cos(x)+1,x,0,6)
```

Listing 2: min.was

This example should be compared with section 34.8 of the GNU GSL manual [2], that shows how to solve the same problem using a 65-lines-long source file written in C, which consists in preparation and calls to the library. In the same spirit, wasora solves systems of differential-algebraic equations (DAE) using the SUNDIALS IDA library [3]. The Lorenz example above should also be compared to the examples of usage of the low-level API provided in the library documentation.

The main focus of wasora is the numerical solution of non-linear equations,[2] which may represent either static or transient (i.e. time-dependent) models. Even more, an outer iterative scheme may be applied in order to perform parametric or optimization runs. In the case that the numerical methods provided by wasora through the GNU GSL and/or SUNDIALS IDA are not enough to model a certain problem, arbitrary user-provided code can be executed by loading dynamically-loadable shared objects. For even more complex or specific tasks (for example numerical routines coded in legacy Fortran code decades ago), a plugin may be implemented in such a way that new functionality is added to the code by interfacing with functions and administrative structures provided by wasora as an API. Many particular plug-ins may be loaded at the same time and can share data structures in order to perform coupled calculations.

[2] Actually, wasora's main focus is to help engineers to cope with the non-linear equations that appear in their chores. These include analysis and interpolation of data generated by other computational codes.

Wasora is thus, on the one hand, a computational tool that can be used to solve complex mathematical problems in such a way that the details are kept in a background plane as long as they are not needed. On the other hand it provides a flexible and extensible computational framework in which to develop specific calculation codes in the same spirit.

1.2 What wasora is not

Wasora should not be seen as a *programming language*, because it is definitely not. If a certain problem can be better solved by coding a computer program, then it should not be solved using wasora. A set of definitions and instructions (which is what wasora inputs are) does not necessarily configure a computer program. Wasora is neither a high-performance computing (HPC) tool. As usual, high-level interfaces come at the cost of speed.

1.3 The wasora suite

The set of computational codes comprised of wasora and the plugins freely distributed under a GPL-compatible license plus other related tools (the script qdp and the documentation system techgdoc) is known as the *wasora suite*, namely

- wasora: the main code that solves general mathematical problems and loads one or more plugins
- skel: template to write a wasora plugin from scratch
- besssugo: a graphical visualization plugin for wasora
- milonga: core-level neutronic code that solves neutron diffusion or transport on unstructured grids
- fino: plugin to solve general partial differential equations using the finite element method
- waspy: plugin to execute python code within wasora
- qdp: a shell script to generate scientific plots from the commandline
- techgdoc: a set of scripts and macros that help to create, modify and track technical documents

These codes share a common framework (the wasora framework) and are written in the same spirit. Many of them make extensive use of other free libraries (e.g. PETSc, SLEPc, SDL). They are hosted on Bitbucket using a distributed version control system (either Git or Mercurial). The list of codes can be accessed at

`https://bitbucket.org/wasora`

Besides the repository with the code itself, each project contains a wiki and an issue tracker. A public mailing list is available at `wasora@talador.com.ar`. Contributions are welcome by first forking the tree and then sending back a pull request.

There exist other plugins that involve private know-how and which are meant to be used within a certain company (i.e. not to be distributed). These plugins (for example pcex and dynetz) are considered private (not privative) and are not part of the wasora suite.

2 Design basis overview

The code was designed according to how a computational code that should serve as an aid to a cognizant engineer such as wasora was supposed to behave, from the original author's humble point of view. The original development began before he actually read Eric Raymond's 17 rules of UNIX Philosophy (appendix C), but it turned out that they were more or less implicitly followed. This section briefly reviews some design decisions that affect how wasora works.

2.1 Types of problems

Wasora performs a series of mathematical and logical algorithms in order to solve the equations that model real physical systems of interest. These equations include both static and transient problems. Static problems may involve many steps, for example, to solve a non-linear problem by performing several iterations. Transient problems may involve one or more static computations at $t = 0$, such as in the case suitable initial conditions are the result of non-linear equations. The number of static steps is given by the special variable `static_steps`. After the static computation, the time t advances either by explicitly setting a time step dt (which may change with t) or by allowing the DAE solver to choose an appropriate value for dt. The computation (either static or transient) ends when t exceeds a special variable called `end_time` or when the special variable `done` is set to a value different from zero.

Single-step static problems can be used to compute a simple mathematical expression such as

```
f(x) := (x+1)*x-5
PRINT %.7f root(f(x),x,0,5)
```

Listing 3: roots.was

or to solve a more complex but still one-step problem

```
# solves the system of equations
#  y = exp(-f(z)*x)
#  x = integral(f(z'), z', 0, z)
#  2 = x+y+z
# where f(z) is a point-wise defined function

FUNCTION f(z) INTERPOLATION akima DATA {
0    0
0.2  0.2
0.5  0.1
0.7  0.8
1    0.5
}

VAR z'
SOLVE 3 UNKNOWNS x y z METHOD hybrids RESIDUALS {
 y-exp(-f(z)*x)
 integral(f(z'),z',0,z)-x
 x+y+z-2
}

PRINT "␣x␣=␣" %f x
PRINT "␣y␣=␣" %f y
PRINT "␣z␣=␣" %f z
```

Listing 4: solve.was

or to evaluate a function at several points. For example, the following function $f(x)$ gives only prime numbers when x is an integer:

```
f(x) := x^2 - x + 41
PRINT_FUNCTION f MIN 1 MAX 40 STEP 1 FORMAT %g
```

Listing 5: allprimes.was

Multi-step static problems are employed to solve iterative problems. For example, the Fibonacci sequence may be generated iteratively with the following input:

```
static_steps = 25

IF step_static=1|step_static=2
 f_n = 1
 f_nminus1 = 1
 f_nminus2 = 1
ELSE
 f_n = f_nminus1 + f_nminus2
 f_nminus2 = f_nminus1
 f_nminus1 = f_n
```

```
ENDIF

PRINT %g step_static f_n
```

Listing 6: fibo-iterative.was

Transient problems may advance time either by explicitly setting the special variable `dt`

```
end_time = 2*pi
dt = 1/10

y = lag(heaviside(t-1), 1)
z = random_gauss(0, sqrt(2)/10)

PRINT t sin(t) cos(t) y z HEADER
```

Listing 7: tran.was

or by writing a DAE equation and letting wasora (actually IDA) take care of handling the time steps:

```
PHASE_SPACE x       # DAE problem with one variable
end_time = 1        # running time
x_0 = 1             # initial condition
x_dot .= -x         # differential equation
PRINT t x HEADER
```

Listing 8: exp.was

Engineers usually need to analyze how systems respond to changes in the input parameters. Therefore, wasora provides a convenient way to perform parametric computations by solving the same problem several times with different input parameters. Wasora can sweep a multidimensional parameter space in a certain pre-defined way (for example by sampling parameters using a quasi-random number sequence to perform a parametric computation) or by employing a certain recipe in order to find extrema of a scalar function of the parameters (for example using conjugate gradients to minimize a cost function). This sweep is obtained by performing an outer iterative loop, which ends either when the parameter space is exhausted by reaching the specified number of outer steps or by convergence of the minimization algorithm.

For instance, the logistic map $x_n = r \cdot x_{n-1}(1 - x_{n-1})$ can be studied by solving it for different values of the parameter r sampling a certain range with a quasi-random number sequence:

```
# compute the logistic map for a range of the parameter r
DEFAULT_ARGUMENT_VALUE 1 2.6  # by default compute r in [2.6:4]
DEFAULT_ARGUMENT_VALUE 2 4

# sweep the parameter r between the arguments given in the commandline
# sample 1000 values from a halton quasi-random number sequence
PARAMETRIC r MIN $1 MAX $2 OUTER_STEPS 1000 TYPE halton

static_steps = 800     # for each r compute 800 steps
x_init = 1/2           # start at x = 0.5
x = r*x*(1-x)          # apply the map

# only print x for the last 50 steps to obtain the asymptotic behaviour
IF step_static>static_steps-50
 PRINT %g r x
ENDIF
```

Listing 9: logistic.was

Instead of sweeping the parameter space, one may want wasora to automatically find the best suitable value for one or more parameters following a certain recipe (e.g. conjugate gradients or Nelder & Mead simple method). The figure to be minimized can be any result computed by wasora, including the result of solving a system of non-linear DAE equations. The following example computes what is the needed reactivity step in order to increase the flux level of a nuclear reactor exactly 2% in 20 seconds:

```
nprec = 6   # six groups of neutron precursors
VECTOR c      SIZE nprec
VECTOR lambda SIZE nprec   DATA   1/7.8e1 1/3.1e1 1/8.5   1/3.2   1/7.1e-1 1/2.5e-1
VECTOR beta   SIZE nprec   DATA   2.6e-4  1.5e-3  1.4e-3  3.0e-3  1.0e-3   2.3e-4
CONST lambda Lambda beta Beta
Lambda = 1e-3
Beta = vecsum(beta)

PHASE_SPACE phi c rho

t_insertion = 1               # reactivity insertion time
end_time = 20 + t_insertion   # target time
min_dt = 0.1                  # fix min and max dt so the DAE
max_dt = 0.1                  # solver doesn't choose dt by himself
target_phi = 1.02             # target level
rhostep = 1e-5                # initial step

# initial conditions for the DAE system
rho_0 = 0
phi_0 = 1
c_0(i) = phi_0 * beta(i)/(Lambda*lambda(i))

# DAE system (reactor point kinetics)
rho .= rhostep * heaviside(t-t_insertion)
phi_dot .= (rho - Beta)/Lambda * phi + sum(lambda(i)*c(i), i, 1, nprec)
c_dot(i) .= beta(i)/Lambda * phi - lambda(i)*c(i)

# Record the time history of a variable as a function of time.
HISTORY phi flux

# the function to be minimized is the quadratic deviation
# of the flux level with respect to the target at t = end_time
f(rhostep) := (target_phi - flux(end_time))^2
MINIMIZE f METHOD nmsimplex STEP 1e-5 TOL 1e-10

# write some information
IF done
  PRINT FILE_PATH flux-iterations.dat TEXT "\#␣" %g step_outer %e rhostep f(rhostep)
ENDIF
IF done_outer
  PRINT t phi HEADER
ENDIF
```

Listing 10: targetflux.was

A particular case of multidimensional minimization problems is that of parameter fitting. For example, the following input uses the binding energy per nucleon as a function of N and Z to fit Weizsäcker's semi-empirical mass formula to predict the mass of isotopes [4]:

$$\frac{B}{A}(A,Z) \approx a_1 - a_2 \cdot A^{-1/3} - a_3 \cdot Z(Z-1)A^{-4/3} - a_4 \cdot (A-2Z)^2 A^{-2} + a_5 \cdot \delta \cdot A^{-\gamma}$$

where

$$\delta = \begin{cases} +1 & \text{for even-}A\text{ and even-}Z \\ 0 & \text{for odd-}A \\ -1 & \text{for even-}A\text{ and odd-}Z \end{cases}$$

```
a1 = 1  # initial guess
a2 = 1
a3 = 1
a4 = 1
a5 = 1
gamma = 1.5

# the functional form of weiszäcker's formula
delta(A,Z) := if(is_odd(A), 0, if(is_even(Z), +1, -1))
W(A,Z) := a1 - a2*A^(-1/3) - a3*Z*(Z-1)*A^(-4/3) - a4*(A-2*Z)^2*A^(-2) + delta(A,Z) * a5*A^(-gamma)

FUNCTION D(A,Z) FILE_PATH binding-2012.dat   # the experimental data
```

```
FIT W TO D VIA a1 a2 a3 a4 a5 gamma          # fit W to D using the six parameters

IF done_outer                               # write the result!
 PRINT "a1␣=␣" %.3f a1 "MeV"
 PRINT "a2␣=␣" %.3f a2 "MeV"
 PRINT "a3␣=␣" %.3f a3 "MeV"
 PRINT "a4␣=␣" %.3f a4 "MeV"
 PRINT "a5␣=␣" %.3f a5 "MeV"
 PRINT "γ␣␣=␣" %.3f gamma
 PRINT_FUNCTION D W D(A,Z)-W(A,Z) FILE_PATH binding-fit.dat
ENDIF
```

Listing 11: fsm.was

To summarize, wasora solves one or more outer iterations (parametric, minimization or fit), each one consisting of

1. one or more static steps, up to `static_steps`
2. zero or more transient steps, until $t >$ `end_time` or `done` $\neq 0$ (one step for each t)

2.2 Input

As already seen in the examples reviewed in the previous section, wasora reads a plain-text input file containing keywords that define the problem to be solved. There are some basic rules that wasora follows, namely

1. the problem definition and its associated math should be entered as naturally as possible,
2. whenever a numerical value is expected, any valid algebraic expression may be entered,
3. arguments should not be position-dependent, they have to be preceded by a self-explanatory keyword, and
4. simple problems ought to need simple inputs.

Input files contain English-based keywords that are either definitions (such as `PHASE_SPACE`) or instructions (such as `PRINT`). These keywords take zero or more arguments, usually by means of other secondary keywords. For example, when defining a matrix one may explicitly state the number of rows and columns using the secondary keywords `ROWS` and `COLS` of the primary keyword `MATRIX`:

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

Some other mathematical tools may give a keyword or an API call with three arguments: a name, a number of rows and a number of columns that should be given in a certain order and one has to refer to the manual to check which one is the appropriate. This behavior in non-compact[3] and in principle is deliberately avoided by the wasora design.

The input file is parsed by wasora at run-time. The following example illustrates and annotates some features of the parser:

```
# this file shows some particularities about the wasora parser

# there are primary and secondary keywords, in this case
# PRINT is the primary keyword and TEXT is the secondary, which
# takes a single token as an argument, in this case the word hello
PRINT TEXT hello
```

[3]Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: Does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact. See section "Compactness and Orthogonality" in chapter 4 of reference [5]

```
# if the text to be printed contains a space, double quotes should be used:
PRINT TEXT "hello␣world"
# if the text to be printed contains quotes, they should be escaped:
PRINT TEXT "hello␣\"world\""

# it does not matter if the argument is a string or an expression, whenever
# a certain argument is expected, either spaces are to be remove or
# the arguments should be enclosed in double quotes:
PRINT 1 + 1    # the parser will read three different keywords
PRINT "1␣+␣1"  # this is the correct way to compute 1+1
PRINT 1+1      # this line also works becasue there are no spaces

# you already guessed it, to insert comments, use the hash `#` character
PRINT sqrt(2)/2     # comments may appear in the same line as a keyword

# in case a hash character is expected to appear literally in an argument
# it should be escaped to prevent wasora to ignore the rest of the line:
PRINT TEXT "\#␣this␣is␣a␣commented␣output␣line"  # this is a wasora comment

# secondary keywords and/or arguments can be given in different lines either by
# a. using a continuation marker composed of a single backslash:
PRINT sqrt(2)/2 \
      sin(pi/4) \
      cos(pi/4)
# b. enclosing the lines within brackets `{` and `}`
PRINT sqrt(2)/2 {
      sin(pi/4)
      # comments may appear inside brackets (but not within continued lines)
      cos(pi/4) }

# arguments may be given in the command line after the input file
# they are referred to as $1, $2, etc. and are literally used
# i.e. they can appear as arguments or even keywords
# if a $n expressions appears in the input file but less than n
# arguments were provided, wasora complains
# this behavior can be avoided by giving a default value:
DEFAULT_ARGUMENT_VALUE 1 world
DEFAULT_ARGUMENT_VALUE 2 2

PRINT TEXT "hello␣$1"
PRINT sqrt($2)/$2

# try executing this input as
# $ wasora parser.was WORLD
# $ wasora parser.was WORLD 3

# if a literal dollar sign is part of an argument, quote it with a backslash:
PRINT TEXT "argument␣\$1␣is␣$1"
```

Listing 12: parser.was

In general, the term table is avoided throughout wasora. Functions are functions and vectors are vectors. Functions may be algebraically-defined and then evaluated to construct a vector whose components can be copied into a shared-memory object. Or a function can be defined point-wise from a set of values given by a vector (maybe read from a shared-memory object) and then interpolated:

```
# read the mesh 'square.msh' and name it "square"
MESH NAME square FILE_PATH square.msh DIMENSIONS 2

# define a function defined over the mesh whose independent values
# are given by the contents of the vector "in" (the size is
# automatically computed from the number of cells in the mesh)
# to define f at the nodes, replace CELLS with NODES
FUNCTION f(x,y) MESH square CELLS VECTOR in

# fill in the values of the vector "in" (probably by reading them
# from a file or from shared memory)
in(i) = sqrt(i)

# define a vector that will hold a vector of another function over
# the mesh. The special variable cells (nodes) contains the number
# of cells (nodes) of the last mesh read.
VECTOR out SIZE cells
# you can use the vecsize() function over "in" to achieve the same result
# VECTOR out SIZE vecsize(in)

# do some computing here
```

```
# PRINT %g nodes cells elements
# PRINT_FUNCTION f
g(x,y) := x^2

# fill in the vector "out" with the function g(x,y)
MESH_FILL_VECTOR MESH square CELLS VECTOR out FUNCTION g
# alternatively one may use an expression of x, y and z
# MESH_FILL_VECTOR MESH square CELLS VECTOR out EXPRESSION x^2

# you can now write out to a shared memory object
PRINT_VECTOR out
```

Listing 13: mesh-fun-vec.was

Again, functions are functions and vectors are vectors. This is one loose example of the application of the *rule of representation* and the *rule of least surprise*, which are two of the seventeen rules of UNIX Philosophy listed in appendix C.

2.3 Output

The main design decision in wasora regarding output is

1. output is completely defined by the user

In particular, if no instructions about what to write as the computation output, nothing is written (*rule of silence*). In principle, output refers to plain-text output including both the terminal (which may be redirected to a file nevertheless) and ASCII files. But it also refers to binary files and to POSIX shared-memory objects and semaphores, which wasora is able to write to (and of course read from also).

This feature is actually a thorough implementation of the *rule of economy*. Back in the seventies, when memory was scarce and CPU time expensive, it made sense in scientific/engineering software to compute and output as many results as possible in a single run. Nowadays (mid 2010s), most of the every-day computations we engineers have to perform take just a few seconds. And as indeed our cost far exceeds current CPU time, it now makes sense to compute just what the user needs instead of having to find a needle in a haystack (i.e. post-processing a fixed-format output file). Should another result be needed, another `PRINT` instruction is added and wasora is re-run to obtain the desired figure. It will be exactly in the expected location.

Besides the fact that output is user defined, some instructions will actually write information in a pre-defined way. For example, the instruction `PRINT_FUNCTION` writes the values that one or more functions take at certain points of the independent variables as an ASCII representation of numbers in a column-wise function—first the independent variables and then the dependent one (or ones if the user asked for more than one function). In this regard, attention was paid to the *rule of composition*, the *rule of separation* and the *rule of parsimony*. If the user wants to plot an interpolated two-dimensional function of a certain data set, she would be better off by feeding the ASCII output generated by wasora into a dedicated plotting program such as Gnuplot, Pyxplot or Paraview instead of trying to use whatever lame plotting capabilities that may be coded into wasora.

```
# define a two-dimensional scalar field
FUNCTION g(x,y) INTERPOLATION rectangle DATA {
0    0    1-1
0    1    1-0.5
0    2    1
1    0    1
1    1    1+0.25
1    2    1
2    0    1-0.25
2    1    1+0.25
2    2    1+0.5
}

# print g(x,y) at the selected range to the standard output
PRINT_FUNCTION g MIN 0 0 MAX 2 2 STEP 0.05 0.05
```

Listing 14: interp2d.was

2.4 Implementation

Wasora is implemented as an executable that reads one plain-text input file (which may further include other files) and executes a set of instructions. Essentially it is best suited for execution in GNU/Linux, as wasora was born and designed within the UNIX philosophy (see section 2.5 below). The usage of wasora in non-UNIX and/or non-free environments is highly discouraged. It is really worth any amount of time and effort to get away from Windows if you are doing computational science.

I first started coding it in C [6] (see appendix B) because it is the language I feel most comfortable with. But then I stumbled upon the concept of *glue layer* [5] and everything started to add up. In effect, wasora is a glue layer between the user at a high level and a bunch of low-level numerical algorithms, most of them from the GNU Scientific Library [2] and the SUNDIALS IDA library [3]. It is therefore appropriate to use C as the programming language. Besides, wasora makes extensive use of complex data structures such as linked lists, hash tables and function pointers in order to reduce the complexity of the algorithms involved (*rule of representation*). Again, the C Programming Language is the appropriate choice for this endeavor because for example Fortran was not designed to manage complex data structures and does not provide flexible mechanisms for handling such added complexity (or at least not in a thorough and native way). On the other hand C++ adds much more complexity than the threshold needed without a net gain.

The development originally started in 2009 as a re-write of some real-time fuzzy-logic control software I wrote for my undergraduate and masters' thesis [7, 8] in my free time. Shortly after, I realized that the code was suitable for the usage in my everyday chores at the company TECNA working as a contractor for the completion of the Atucha II Nuclear Power Plant. Further development was continued both at TECNA and in my free time, with other people contributing with bug reports, ideas for enhancements and actual code. First versions (series 0.1.x) used Subversion as the version control system. Then we switched to Bazaar for the 0.2.x versions to finally converge to Mercurial since the 0.3.x series. Current version (as of 2016) is 0.4.x.

As knowing exactly which version of the code is being used to run a certain computation, versions in wasora change with each commit to the control version system. So the x above increases with each commit, including merges. Of course, when using distributed control version systems one cannot guarantee that there are no independent commits that result in the same version number with the proposed scheme. However, by hosting the repository in Bitbucket we can minimize the issues of duplicate commits, that will appear only in private forks. Nevertheless, not only does wasora report the major-minor-revision version number but it also reports the actual SHA1 hash of the changeset used to compile the binary. Moreover, if the Mercurial working tree contains uncommitted changes, a $+\Delta$[4] is appended to the hash string. So, if wasora is called with no arguments, it reports the version, the hash and the date of the last commit:

```
$ wasora
wasora 0.4.47  (c6f81e76e3f9 + Δ 2015-12-30 15:23 -0300)
wasora's an advanced suite for optimization & reactor analysis
$
```

The Δ shows that this particular version of wasora was compiled from a tree that has some modifications with respect to the last commit. After committing the changes and calling the executable with the `-v` (or `--version`) argument in the command line, we get rid of the Δ and obtain further details about the binary executable:

```
$ wasora -v
wasora 0.4.48  (aa1175af1ed6 2016-01-06 10:05 -0300)
```

[4] This glyph can be seen only in operating systems with native UTF8 support.

```
wasora's an advanced suite for optimization & reactor analysis

 rev hash aa1175af1ed6c6d34c57cb4cb476f0e3b17d8bbd
 last commit on 2016-01-06 10:05 -0300 (rev 202)

 compiled on 2016-01-06 10:05:15 by gtheler@frink (linux-gnu x86_64)
 with gcc (Debian 4.9.2-10) 4.9.2 using -O2 and linked against
  GNU Scientific Library version 1.16
  SUNDIALs Library version 2.5.0
  GNU Readline version 6.3

 wasora is copyright (C) 2009-2016 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.
$
```

2.5 Wasora and the UNIX philosophy

As can be seen in the detailed output of `wasora -v`, wasora is linked against three libraries. The first is the GNU Scientific Library, which implements most of the numerical method used to solve the mathematical problem defined in the input file (function interpolation, integration, differentiation, root finding, data fitting, etc.). The second library is SUNDIALs IDA and is optional. It is used to solve differential-algebraic equations (referred to as DAEs, i.e. a generalization of ordinary differential equations or ODEs), which is a very useful feature wasora provides and may be the central issue for many users. The third one is also optional, and is the GNU Readline library which is used for a debugger-like interactive interface that wasora can provide for transient problems. A very basic scheme of breakpoints and watches can be used to track the evolution of complex time-dependent problems, normally needed only by advanced users.

Although it may be difficult for new users to get all the needed libraries compiled and installed, the usage of third-party libraries—especially free and open high-quality math libraries designed by mathematicians and coded by computer scientists— instead of hard-coding particular poorly-coded routines into the source code is one of the most important aspects of the UNIX philosophy [5], in which wasora was first born and conceptually designed. In effect, appendix C summarizes the seventeen rules of UNIX philosophy compiled by Raymond. Some of them were deliberately used when programming wasora, but some others were just implicit consequences of the programming style used in wasora.

3 License

Wasora is free software—both as in free speech and as in free beer, although the first meaning is far more important than the second one—and is distributed under the terms of the GNU General Public License version 3. In words of the Free Software Foundation,

> Nobody should be restricted by the software they use. There are four freedoms that every user should have:
>
> 0. the freedom to use the software for any purpose,
> 1. the freedom to change the software to suit your needs,
> 2. the freedom to share the software with your friends and neighbors, and
> 3. the freedom to share the changes you make.
>
> When a program offers users all of these freedoms, we call it free software.

> Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

Not only does wasora provide all the four basic freedoms to the software user, but it also encourages her to study, understand, analyze and hack it. And of course, to share the associated discoveries, suggestions, improvements and fixed bugs under the terms of the GNU GPL—especially with wasora's original author. To sum up:

> Wasora is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> Wasora is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

References

[1] Edward N. Lorenz. "Deterministic non-periodic flow". In: *Journal of the Atmospheric Sciences* 20 (1963), pp. 130–141.

[2] M. Galassi et al. *GNU Scientific Library Reference Manual*. 3rd. ISBN: 0954612078.

[3] A. C. Hindmarsh et al. "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers". In: *ACM Transactions on Mathematical Software* 31.3 (2005), pp. 363–396.

[4] C. F. von Weizsäcker. "Zur Theorie der Kernmasse". German. In: *Zeitschrift für Physik* 96 (1935), pp. 431–458.

[5] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall, 1988.

[7] Germán Theler. *Controladores basados en lógica difusa y loops de convección natural caóticos*. Spanish. Proyecto Integrador de la Carrera de Ingeniería Nuclear, Instituto Balseiro. 2007.

[8] Germán Theler. *Análisis no lineal de inestabilidades en el problema acoplado termohidráulico-neutrónico*. Spanish. Tesis de la Carrera de Maestría en Ingeniería, Instituto Balseiro. 2008.

[9] G. Theler and F. J. Bonetto. "On the stability of the point reactor kinetics equations". In: *Nuclear Engineering and Design* 240.6 (June 2010), pp. 1443–1449.

[10] Donald E. Knuth. *The Art of Computer Programming*. Vol. 1–4. Addison-Wesley, 1968–2006.

A How to refer to wasora

Wasora means “Wasora’s an Advanced Suite for Optimization & Reactor Analysis”, which is of course a recursive acronym as in “GNU’s Not Unix” and in “to understand recursion one has first to understand recursion.” The code name should always be written using lowercase letters, except when it starts a sentence. In such case, the ‘W’ should be capitalized. The expression “WASORA” ought to be avoided because

1. words written in uppercase letters ANNOY READERS
2. names written in uppercase letters remind of old-fashioned inflexible poorly-coded Fortran-based engineering programs

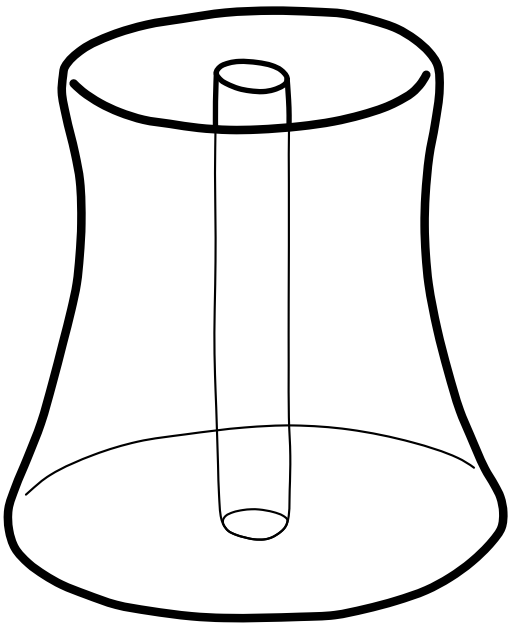
A.1 Pronunciation

The name is originally Spanish, so it should be pronounced /wɒ'sɔɾɑ/ although the English variation /wɒ'soʊɾɑ/ and even the German version /vɒ'sɔɾɑ/ are accepted.

A.2 Logo and graphics

The official wasora logotype is shown in figure 1a. The original is a vector image in SVG format that can be found in the `doc` subdirectory of the wasora repository 1b. Usage in the form of other vector formats (e.g. PDF or EPS) is allowed. Conversion to lossless-compressed bitmap formats (e.g. PNG or TIFF) is discouraged but may be needed if the media format does not support vector graphics (note that HTML does support plain SVG). Conversion to compressed bitmap formats with pixel-level degradation (i.e. JPEG) is forbidden.

The logo is distributed under the terms of the GNU GPLv3. It may be freely modified as long as the distribution satifies the license and the author of the modifications claims copyright on the changes only.



(a)

(b)

Figure 1: The wasora logotype (a) available in the `doc` subdirectory of the wasora repository (b)

B Development history

It was at the movies when I first heard about dynamical systems, non-linear equations and chaos theory. The year was 1993, I was ten years old and the movie was Jurassic Park. Dr. Ian Malcolm (the character played by Jeff Goldblum) explained sensitivity to initial conditions in a memorable scene, which is worth to watch again and again (figure 2). Since then, the fact that tiny variations may lead to unexpected results has always fascinated me. During high school I attended a very interesting course on fractals and chaos that made me think further about complexity and its mathematical description. Nevertheless, not until college was I able to really model and solve the differential equations that give rise to chaotic behavior.

Figure 2: Dr. Ian Malcolm (Jeff Goldblum) explaining the concept of sensitivity to initial conditions in chaotic systems in the 1993 movie *Jurassic Park*.

In fact, initial-value ordinary differential equations arise in a great variety of subjects in science and engineering. Classical mechanics, chemical kinetics, structural dynamics, heat transfer analysis and dynamical systems, amongst other disciplines, heavily rely on equations of the form

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x})$$

During my years of undergraduate student, whenever I faced these kind of equations, I had to choose one of the following three options:

1. program an ad-hoc numerical method such as Euler or Runge-Kutta, matching the requirements of the system of equations to solve
2. use a standard numerical library such as the GNU Scientific Libary and code the equations to solve into a C program (or maybe in Python)
3. use a high-level system such as Octave, Maxima, or some non-free (and worse) programs[5]

Of course, each option had its pros and its cons. But none provided the combination of advantages I was looking for, namely flexibility (option one), efficiency (option two) and reduced input work (partially given by option three). Back in those days I ended up wandering between options one and two, depending on the type of problem I had to solve. However, even though one can with some effort make the code read some parameters from a text file, any other drastic change usually requires a modification in the source code—some times involving a substantial amount of work—and a further recompilation of the code. This was what I most disliked about this way of working, but I could nevertheless live with it.

[5]I will not name such privative programs so I do not encourage new generations to even know their name. Non-free software, especially scientific and academic software, is evil. Avoid it at any cost.

Regardless of this situation, during my last year of Nuclear Engineering, I ran into a nuclear reactor model that especially called my attention and forced me to re-think the ODE-solving problem issue. The model was implemented in a certain non-free software which I had been told was the actual panacea for the engineering community—and yet I was using for the very first time. When I opened the file and took a look at something that I was told was a graphical representation of the model, I was not able to understand any of the several screens the model contained. Afterward, somebody explained to me that a set of unintelligible blocks that were somehow interconnected in a rather cumbersome way was how the reactor power was computed. I wish I had a copy of the screen in order to illustrate how shocking it was to me.

The equation represented by what seemed to me as a complex topology problem was as simple as [9]

$$\frac{d\phi}{dt} = \frac{\rho - \beta}{\Lambda} + \sum_{i=1}^{I} \lambda_i \cdot c_i$$

My first reaction was to ask why someone would prefer such a cumbersome representation instead of writing something like

```
phi_dot .= (rho - beta)/Lambda * phi + sum(lambda(i)*c(i), i, 1, I)
```

in a plain-text file and let a computer program parse and solve it. I do not remember what the teacher's answer was, and I still do not understand why would somebody prefer to solve a very simple differential equation by drawing blocks and connecting them with a mouse with no mathematical sense whatsoever.

That morning I realized that in order to transform a user-defined string representing a differential equation into something that an ODE-solving library such as the GNU Scientific Library would understand, only a good algebraic parser plus some simple interface routines were needed. The following two years were very time-consuming for me, so I was not able to undertake such a project. Nevertheless, eventually I earned a Master's Degree in 2008 [8] and afterward my focus shifted away from academic projects into the nuclear industry and some gaps of time for freelance programming popped up. I started to write wasora from scratch in my free time, and one of the first features I included was an adaptation of a small algebraic parser posted online[6] (which should be replaced by a more efficient tree-based parser), freely available under the Creative Commons License. Before I became aware, I was very close to arriving at a tool that would have met my needs when I was an engineering student. Moreover, a tool like this one would have been extremely helpful during the course on non-linear dynamics I took back in 1999. With some luck, it would also meet somebody else's needs as well. This is how wasora entered into the scene.

From this point onward, the development continued as explained in section 2.4 in page 13.

[6] http://stackoverflow.com/questions/1384811/code-golf-mathematical-expression-evaluator-that-respects-pemdas

C Raymond's 17 rules of UNIX philosophy

These are briefly Eric Raymond's 17 rules of UNIX philosophy as discussed in "The Art of UNIX Programming" [5], which of course is a word game to Donald Knuth's "The Art of Computer Programming" [10]. Both references are a great source of inspiration for wasora in particular and for my professional life in general.

Rule of Modularity Developers should build a program out of simple parts connected by well defined interfaces, so problems are local, and parts of the program can be replaced in future versions to support new features. This rule aims to save time on debugging code that is complex, long, and unreadable.

Rule of Clarity Developers should write programs as if the most important communication is to the developer, including themself, who will read and maintain the program rather than the computer. This rule aims to make code readable and comprehensible for whoever works on the code in future.

Rule of Composition Developers should write programs that can communicate easily with other programs. This rule aims to allow developers to break down projects into small, simple programs rather than overly complex monolithic programs.

Rule of Separation Developers should separate the mechanisms of the programs from the policies of the programs; one method is to divide a program into a front-end interface and back-end engine that interface communicates with. This rule aims to let policies be changed without destabilizing mechanisms and consequently reducing the number of bugs.

Rule of Simplicity Developers should design for simplicity by looking for ways to break up program systems into small, straightforward cooperating pieces. This rule aims to discourage developers' affection for writing "intricate and beautiful complexities" that are in reality bug prone programs.

Rule of Parsimony Developers should avoid writing big programs. This rule aims to prevent overinvestment of development time in failed or suboptimal approaches caused by the owners of the program's reluctance to throw away visibly large pieces of work. Smaller programs are not only easier to optimize and maintain; they are easier to delete when deprecated.

Rule of Transparency Developers should design for visibility and discoverability by writing in a way that their thought process can lucidly be seen by future developers working on the project and using input and output formats that make it easy to identify valid input and correct output. This rule aims to reduce debugging time and extend the lifespan of programs.

Rule of Robustness Developers should design robust programs by designing for transparency and discoverability, because code that is easy to understand is easier to stress test for unexpected conditions that may not be foreseeable in complex programs. This rule aims to help developers build robust, reliable products.

Rule of Representation Developers should choose to make data more complicated rather than the procedural logic of the program when faced with the choice, because it is easier for humans to understand complex data compared with complex logic. This rule aims to make programs more readable for any developer working on the project, which allows the program to be maintained.

Rule of Least Surprise Developers should design programs that build on top of the potential users' expected knowledge; for example, '+' should always mean addition in a calculator program. This rule aims to encourage developers to build intuitive products that are easy to use.

Rule of Silence Developers should design programs so that they do not print unnecessary output. This rule aims to allow other programs and developers to pick out the information they need from a program's output without having to parse verbosity.

Rule of Repair Developers should design programs that fail in a manner that is easy to localize and diagnose or in other words "fail noisily". This rule aims to prevent incorrect output from a program from becoming an input and corrupting the output of other code undetected.

Rule of Economy Developers should value developer time over machine time, because machine cycles today are relatively inexpensive compared to prices in the 1970s. This rule aims to reduce development costs of projects.

Rule of Generation Developers should avoid writing code by hand and instead write abstract high-level programs that generate code. This rule aims to reduce human errors and save time.

Rule of Optimization Developers should prototype software before polishing it. This rule aims to prevent developers from spending too much time for marginal gains.

Rule of Diversity Developers should design their programs to be flexible and open. This rule aims to make programs flexible, allowing them to be used in other ways than their developers intended.

Rule of Extensibility Developers should design for the future by making their protocols extensible, allowing for easy plugins without modification to the program's architecture by other developers, noting the version of the program, and more. This rule aims to extend the lifespan and enhance the utility of the code the developer writes.

D Examples execution and results

D.1 The Lorenz system

Edward Lorenz introduced the nowadays-famous dynamical system in his seminal 1963 paper Deterministic Nonperiodic Flow

$$\begin{aligned}\dot{x} &= \sigma\,(y - x)\\ \dot{y} &= x\,(r - z) - y\\ \dot{z} &= xy - bz\end{aligned}$$

Wasora can be used to solve it by writing the equations in the input file as naturally as possible, as illustrated in the input file that follows.

```
# lorenz' seminal dynamical system solved with wasora
PHASE_SPACE x y z
end_time = 40

# parameters that lead to chaos
sigma = 10
r = 28
b = 8/3

# initial conditions
x_0 = -11
y_0 = -16
z_0 = 22.5

# the dynamical system (note the dots before the '=' sign)
x_dot .= sigma*(y - x)
y_dot .= x*(r - z) - y
z_dot .= x*y - b*z

# write the solution to the standard output
PRINT t x y z
```

Listing 15: lorenz.was

```
$ wasora lorenz.was > lorenz.dat
$ head lorenz.dat
0.000000e+00    -1.100000e+01   -1.600000e+01   2.250000e+01
2.384186e-07    -1.100001e+01   -1.600001e+01   2.250003e+01
4.768372e-07    -1.100002e+01   -1.600002e+01   2.250006e+01
9.536743e-07    -1.100005e+01   -1.600004e+01   2.250013e+01
1.907349e-06    -1.100010e+01   -1.600008e+01   2.250024e+01
3.814697e-06    -1.100019e+01   -1.600017e+01   2.250047e+01
7.629395e-06    -1.100038e+01   -1.600034e+01   2.250091e+01
1.525879e-05    -1.100076e+01   -1.600068e+01   2.250179e+01
3.051758e-05    -1.100153e+01   -1.600136e+01   2.250356e+01
6.103516e-05    -1.100305e+01   -1.600271e+01   2.250708e+01
$ tail lorenz.dat
3.996319e+01    9.358722e+00    1.541198e+00    3.568235e+01
3.996745e+01    9.025737e+00    1.240715e+00    3.533280e+01
3.997172e+01    8.694644e+00    9.654912e-01    3.497443e+01
3.997599e+01    8.366400e+00    7.146306e-01    3.460905e+01
3.998025e+01    8.041884e+00    4.871387e-01    3.423833e+01
3.998452e+01    7.721893e+00    2.819431e-01    3.386374e+01
3.998879e+01    7.407148e+00    9.791065e-02    3.348664e+01
3.999306e+01    7.098288e+00    -6.613236e-02   3.310819e+01
3.999732e+01    6.795877e+00    -2.113808e-01   3.272946e+01
4.000159e+01    6.500405e+00    -3.390346e-01   3.235134e+01
$ gnuplot -e "set terminal pdf; set output 'lorenz.pdf'; set ticslevel 0; splot 'lorenz.dat' u 2:3:4 w l ti ''"
$
```

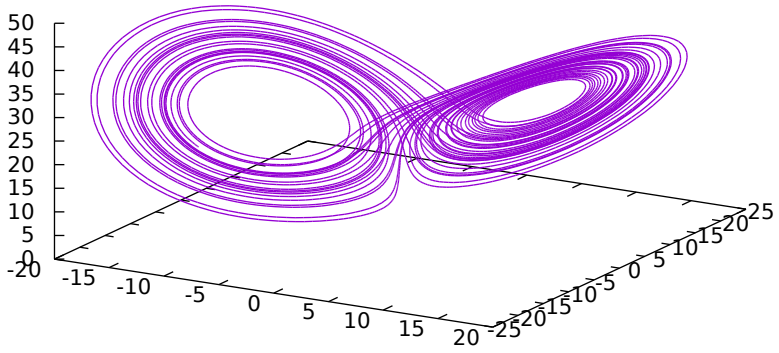


Figure 3: The attractor obtained by wasora after solving the Lorenz system with the nominal parameters.

By using the `PHASE_SPACE` keyword, a three-dimensional phase-space spanned by variables x, y and z is defined. Therefore, wasora expects now three differential-algebraic equations involving these three variables and its time derivatives. The special variable `end_time` is set to forty, thus the system will be solved for the non-dimensional time range $0 < t < 40$. Parameters σ, r and b are assigned constant values, which by the way are the ones used by Lorenz in his original paper. The initials conditions are set by assigning values to the special symbols `x_0`, `y_0` and `z_0` which represent the initial values of said variables. These assignments are evaluated and processed only when $t = 0$ and are ignored for $t > 0$. The following lines define the dynamical system by adding a dot before the equal sign, i.e. "`.=`". This construction tells wasora that the assignent is not a regular one but rather that a differential-algebraic expression is being defined. The postfix `_dot` indicates that the time derivative of the function is being referenced. Finally, the `PRINT` instruction writes into the standard output the current non-dimensional time t and the three variables that constitute the solution of the dynamical system as time advances. Starting from $t = 0$, wasora (actually the IDA library) chooses an appropriate time step so as to keep the numerical error bounded.

D.2 One-dimensional minimization

Find the minimum of the function $f(x) = \cos(x) + 1$ whitin the interval $[0, 2\pi]$.

```
VAR x
PRINT %.7f func_min(cos(x)+1,x,0,6)
```

Listing 16: min.was

```
$ wasora min.was
3.1415914
$
```

The arguments of the keyword `PRINT` can be expressions that are evaluated whenever the instruction is executed. In this case we use the functional `func_min` as the expression, with the first argument equal to the expression we want to minimize. In the second argument we must tell wasora which is the variable it has to vary in order to have a minimum. In this case it is x, which wasora understands it is a variable because it was defined with the `VAR` keyword. The third and fourth arguments a and b give the interval $a < x < b$. The first argument to `PRINT` is a float format specifier as in the C standard library function `printf`.

D.3 Root of a one-dimensional function

Find the root of the function $f(x) = (x + 1) \cdot x - 5$ within the interval $[0, 5]$.

```
f(x) := (x+1)*x-5
PRINT %.7f root(f(x),x,0,5)
```

Listing 17: roots.was

```
$ wasora roots.was
1.7912878
$
```

This example is similar to the previous one. The difference is that x does not need to be declared as a variable because the function $f(x)$ is defined using the "`:=`" operator and it implicitly defines the argument as a variable. The functional `root` takes an expression as the first argument (which in this case is the function f evaluated at the point x), and the variable over which the root is to be found as the second one. The third a and fourth b arguments give the range $a < x < b$ where the root is to be sought.

D.4 A system of algebraic equations

Solve the following system of three non-linear algebraic equations

$$
\begin{aligned}
0 &= y - \exp\left[-f(z) \cdot x\right] \\
0 &= \int_0^z f(z')\, dz' - x \\
0 &= x + y + z - 2
\end{aligned}
$$

for x, y and z where $f(z)$ is a pointwise-defined function such that the following values

z	$f(z)$
0.0	0.0
0.2	0.2
0.5	0.1
0.7	0.8
0.7	0.8
1.0	0.5

are interpolated using an Akima-based scheme.

```
# solves the system of equations
#  y = exp(-f(z)*x)
#  x = integral(f(z'), z', 0, z)
#  2 = x+y+z
# where f(z) is a point-wise defined function

FUNCTION f(z) INTERPOLATION akima DATA {
0    0
0.2  0.2
0.5  0.1
0.7  0.8
1    0.5
}

VAR z'
```

```
SOLVE 3 UNKNOWNS x y z METHOD hybrids RESIDUALS {
 y-exp(-f(z)*x)
 integral(f(z'),z',0,z)-x
 x+y+z-2
}

PRINT "␣x␣=␣" %f x
PRINT "␣y␣=␣" %f y
PRINT "␣z␣=␣" %f z
```

Listing 18: solve.was

```
$ wasora solve.was
 x =    0.319603
 y =    0.784170
 z =    0.896227
$
```

D.5 Print only prime numbers

The astonishing function $f(x) = x^2 - x + 41$ gives only prime numbers for integer values of the argument x.

```
f(x) := x^2 - x + 41
PRINT_FUNCTION f MIN 1 MAX 40 STEP 1 FORMAT %g
```

Listing 19: allprimes.was

```
$ wasora allprimes.was
1       41
2       43
3       47
4       53
5       61
6       71
7       83
8       97
9       113
10      131
11      151
12      173
13      197
14      223
15      251
16      281
17      313
18      347
19      383
20      421
21      461
22      503
23      547
24      593
25      641
26      691
27      743
28      797
29      853
30      911
31      971
32      1033
33      1097
34      1163
35      1231
36      1301
37      1373
38      1447
39      1523
40      1601
$
```

The `PRINT_FUNCTION` keyword takes at least one function name and prints in a column-based fashion first the independent variables (in this case only x) and then the evaluated functions (in this case only f) at those points. As $f(x)$ is an algebraic function, it is mandatory to provide an explicit range where the function is to be evaluated. This is given with the `MIN`, `MAX` and `STEP` keywords. In this case, we ask wasora to evaluate $f(x)$ for $x = 1, 2, \ldots, 40$ and print it as neatly as possible with the format specifier `%g`.

D.6 The Fibonacci sequence as an iterative problem

The Fibonacci sequence f_n defined as

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_n &= f_{n-2} + f_{n-1} \qquad \text{for } n > 2 \end{aligned}$$

can be solved in many different ways.[7] The following input uses the straightforward iterative one.

```
static_steps = 25

IF step_static=1|step_static=2
 f_n = 1
 f_nminus1 = 1
 f_nminus2 = 1
ELSE
 f_n = f_nminus1 + f_nminus2
 f_nminus2 = f_nminus1
 f_nminus1 = f_n
ENDIF

PRINT %g step_static f_n
```

Listing 20: fibo-iterative.was

```
$ wasora fibo-iterative.was
1       1
2       1
3       2
4       3
5       5
6       8
7       13
8       21
9       34
10      55
11      89
12      144
13      233
14      377
15      610
16      987
17      1597
18      2584
19      4181
20      6765
21      10946
22      17711
23      28657
24      46368
25      75025
$
```

[7] See The wasora Realbook for three ways using wasora.

First, the special variable `static_steps` is set to a non-zero value to indicate that we want to solve an iterative (but static, i.e. with no time dependence) problem. The problem is solved by using three variables named `f_n`, `f_nminus1` and `f_nminus2` (remember that a variable name cannot contain the character '-', nor any other operator sig). For the first two steps, where the special variable `step_static` is equal to one and two respectively, we initialize the three variable to one. In the rest of the step, we updated the two `f_nminusN` variables and compute `f_n` as the sum of the other two. The `PRINT` instruction is executed in each of the twenty five static steps, giving two columns with the step number and the value of f_n.

D.7 A transient problem

This example illustrates how wasora can be used to solve a transient problem. This case defines a time-dependent variable y equal to a first-order lag of a step at $t = 1$, and another variable z sampled from a gaussian random generator. It also prints the functions $sin(t)$ and $cos(t)$.

```
end_time = 2*pi
dt = 1/10

y = lag(heaviside(t-1), 1)
z = random_gauss(0, sqrt(2)/10)

PRINT t sin(t) cos(t) y z HEADER
```

Listing 21: tran.was

```
$ wasora tran.was | qdp -o tran
$
```

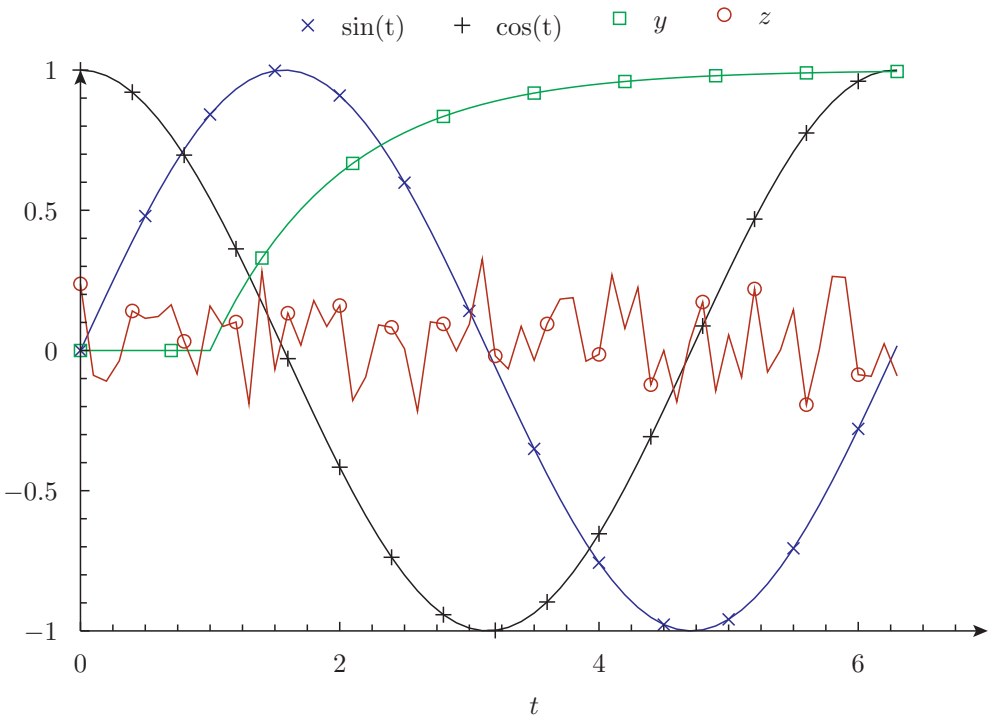


Figure 4: The results of 'tran.was': sine and cosine of t, a first-order lag of a step and a gaussian random variable.

If the special variable `end_time` is set to a non-zero value, wasora assumes that the problem defined in the input is a transient problem. The special variable `t` holds the current value of the time. The special variable `dt` can be set to the time step. In this example it is fixed to one tenth of whatever units the variable `t` is assumed to be (it defaults to $dt = 1/16$ to avoid truncation errors when summing up time steps due to

the periodic representation of decimal fractions in a base-two floating point representation), but its value can be re-assigned at every time step to obtain a variable time step. The variable y is set to a first-order lag (function `lag`) of a heaviside step at $t = 1$ (function `heaviside`) with a characteristic time equal to one. Variable z is a gaussian random value, sampled at each time step. The keyword `PRINT` writes, for each time step, the time t, the sine and the cosine of t, and both variables y and z. This output format is already suited to be plotted as variables vs first-column with tools such as Gnuplot or Pyxplot. The result in figure 7 was obtained with by piping the output of wasora to the tool `qdp`.[8]

D.8 The differential equation for negative feedback

The "Hello World!" case for differential equations is

$$\frac{dx}{dt} = -x \tag{1}$$

which for $x(0) = 1$ gives $x(t) = \exp(-t)$ as a solution. Of course, a simple problem needs a simple input:

```
PHASE_SPACE x       # DAE problem with one variable
end_time = 1        # running time
x_0 = 1             # initial condition
x_dot .= -x         # differential equation
PRINT t x HEADER
```

Listing 22: exp.was

```
$ wasora exp.was | qdp -o exp
$
```

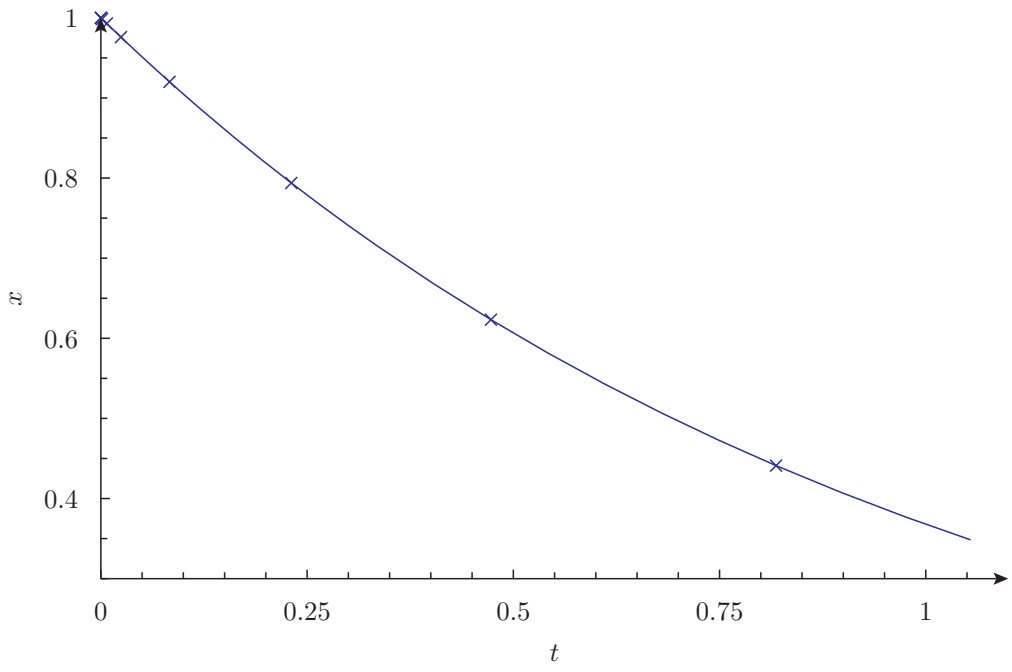


Figure 5: Numerical solution of $\dot{x} = -x$ with $x(0) = 1$ (`exp.was`).

We tell wasora that our phase space is composed just by the scalar variable x. Then, that the running time is `end_time` whose value is set to one time unit, whatever units time has. Then, that the initial value of x is also one unit of whatever units this magnitude is expected to use. Then, we write the differential

[8] `https://bitbucket.org/gtheler/qdp`

equation as naturally as possible. Two things to note here: (a) the equal sign is prepended by a dot to indicate that this line is actually a DAE-equation and not just a plain assignment, and (b) the time derivative of a variable is denoted by appending `_dot` to the variable name, thus $\dot{x}$ becomes `x_dot`. Finally, we print the time t and the value of the variable x to be quickly and dirty plotted by qdp.

D.9 The logistic map

The logistic map is a polynomial mapping often cited as an archetypal example of how complex, chaotic behavior can arise from very simple non-linear dynamical equations. The map was popularized in a seminal 1976 paper by the biologist Robert May. Mathematically, the logistic map is written

$$x_{n+1} = r \cdot x_n(1 - x_n)$$

where x_n is a number between zero and one that represents the ratio of existing population to the maximum possible population. The values of interest for the parameter r are those in the interval $(0, 4]$.

The following input solves eight-hundred iterations of the logistic map parametrically sweeping a certain interval of the range r using a quasi-random number sequence generator to obtain the map's bifurcation diagram.

```
# compute the logistic map for a range of the parameter r
DEFAULT_ARGUMENT_VALUE 1 2.6  # by default compute r in [2.6:4]
DEFAULT_ARGUMENT_VALUE 2 4

# sweep the parameter r between the arguments given in the commandline
# sample 1000 values from a halton quasi-random number sequence
PARAMETRIC r MIN $1 MAX $2 OUTER_STEPS 1000 TYPE halton

static_steps = 800      # for each r compute 800 steps
x_init = 1/2            # start at x = 0.5
x = r*x*(1-x)           # apply the map

# only print x for the last 50 steps to obtain the asymptotic behaviour
IF step_static>static_steps-50
 PRINT %g r x
ENDIF
```

Listing 23: logistic.was

```
$ wasora logistic.was > logistic.dat
$ gnuplot -e "set terminal pdf; set output 'logistic.pdf'; plot 'logistic.dat' ps 0.05 pt 5 lt 3 ti ''"
$
```

By default, the parameter r is swept over the interval $[2.6, 4]$. Other values can be given in the commandline after the input filename `logistic.was` that will replace the constructions `$1` and `$2` in the `PARAMETRIC` line. One thousand values of r are sampled from said interval using a Halton sequence. From an initial value $x_0 = 1/2$ the logistic map is iteratively solved up to $n = 800$. For the last 50 steps, r and x_n are printed to the standard output such that a plot of the second column vs. the first one gives the expected bifurcation diagram.

D.10 Setting a target flux

This input solves the neutron point kinetic equations

$$\begin{cases} \dfrac{d\phi}{dt} = \dfrac{\rho(t) - \sum_{i=1}^{N} \beta_i}{\Lambda} \cdot \phi + \displaystyle\sum_{i=1}^{I} \lambda_i \cdot c_i \\ \dfrac{c_i}{dt} = \dfrac{\beta_i}{\Lambda} \cdot \phi - \lambda_i \cdot c_i \qquad\qquad i = 1, \ldots, I \end{cases}$$

to answer the following question: "Which reactivity step applied at $t = 1$ sec. gives rise to a 2% increase of the flux after 20 seconds?"

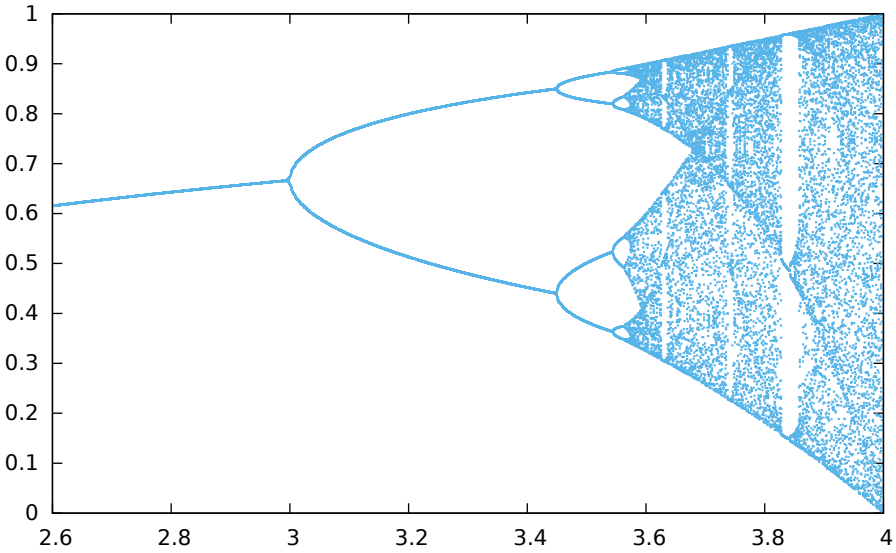


Figure 6: The logistic map bifurcation diagram for $2.6 \leq r \leq 4$ solved parametrically by wasora.

```
nprec = 6   # six groups of neutron precursors
VECTOR c      SIZE nprec
VECTOR lambda SIZE nprec   DATA   1/7.8e1 1/3.1e1 1/8.5   1/3.2   1/7.1e-1 1/2.5e-1
VECTOR beta   SIZE nprec   DATA   2.6e-4  1.5e-3  1.4e-3  3.0e-3  1.0e-3   2.3e-4
CONST lambda Lambda beta Beta
Lambda = 1e-3
Beta = vecsum(beta)

PHASE_SPACE phi c rho

t_insertion = 1               # reactivity insertion time
end_time = 20 + t_insertion   # target time
min_dt = 0.1                  # fix min and max dt so the DAE
max_dt = 0.1                  # solver doesn't choose dt by himself
target_phi = 1.02             # target level
rhostep = 1e-5                # initial step

# initial conditions for the DAE system
rho_0 = 0
phi_0 = 1
c_0(i) = phi_0 * beta(i)/(Lambda*lambda(i))

# DAE system (reactor point kinetics)
rho .= rhostep * heaviside(t-t_insertion)
phi_dot .= (rho - Beta)/Lambda * phi + sum(lambda(i)*c(i), i, 1, nprec)
c_dot(i) .= beta(i)/Lambda * phi - lambda(i)*c(i)

# Record the time history of a variable as a function of time.
HISTORY phi flux

# the function to be minimized is the quadratic deviation
# of the flux level with respect to the target at t = end_time
f(rhostep) := (target_phi - flux(end_time))^2
MINIMIZE f METHOD nmsimplex STEP 1e-5 TOL 1e-10

# write some information
IF done
  PRINT FILE_PATH flux-iterations.dat TEXT "\#␣" %g step_outer %e rhostep f(rhostep)
ENDIF
IF done_outer
  PRINT t phi HEADER
ENDIF
```

Listing 24: targetflux.was

```
$ wasora targetflux.was | qdp -o targetflux --xrange "[0:21]"
```

```
$ tail -n1 flux-iterations.dat
#       18      3.761658e-05    9.915842e-16
$
```

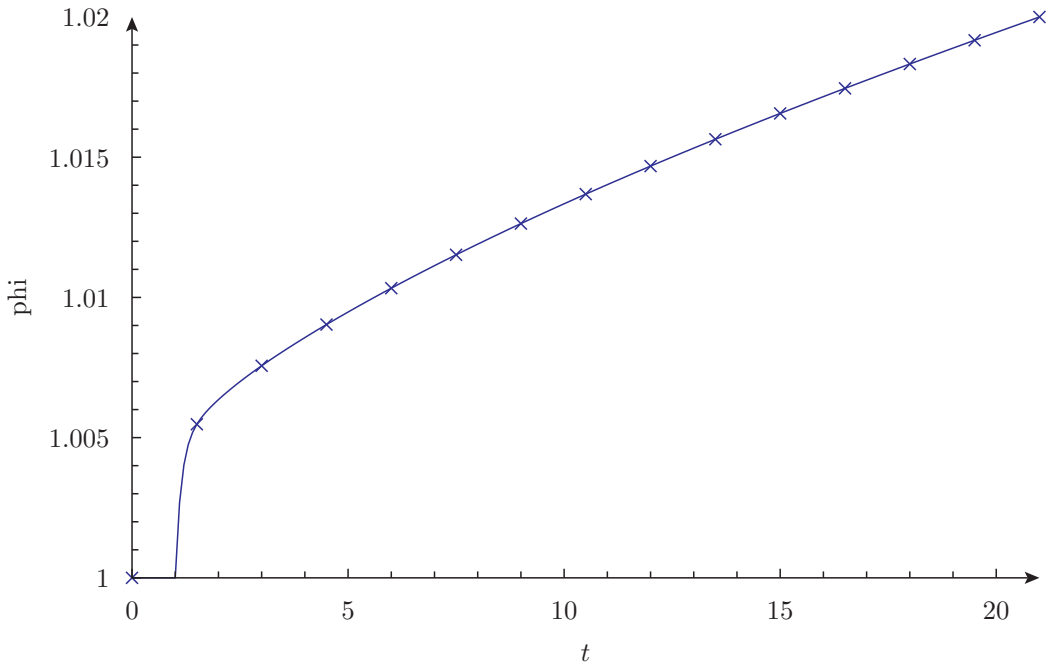


Figure 7: Converged solution to answer the question "which step reactivity gives a 2% increase in the flux after 20 seconds?"

The point kinetics equations are solved as usually using the variable ρ_{step} as a parameter. A function flux(t) with the actual history of the phase-space function ϕ is defined using the keyword `HISTORY`. An additional function $f(\rho_{\text{step}})$ is defined as the square of the difference between the actual flux ϕ and the target flux ϕ_{target}. This function, which is evaluated when $t =$`end_time`, is minimized over its single argument ρ_{step} using the `MINIMIZE` keyword. Nelder & Mead's simplex method is used to solve the minimization problem.

D.11 Semi-empirical mass formula fit

Given a certain nuclide characterized by the number Z of protons and the total number A of nucleons (i.e. protons plus neutrons), the Weizsäcker's mass formula based on the liquid drop model gives a expected dependence of the binding energy per nucleon B/A in terms of algebraic sums of powers of Z and A that involve some multiplicative factors that are to be determined experimentally.

To make a long story short, the semi-empirical mass formula says that the binding energy per nucleon of a nuclide of mass number A with atomic number Z is

$$\frac{B}{A}(A,Z) \approx a_1 - a_2 \cdot A^{-1/3} - a_3 \cdot Z(Z-1)A^{-4/3} - a_4 \cdot (A-2Z)^2 A^{-2} + a_5 \cdot \delta \cdot A^{-\gamma}$$

with

$$\delta = \begin{cases} +1 & \text{for even-}A\text{ and even-}Z \\ 0 & \text{for odd-}A \\ -1 & \text{for even-}A\text{ and odd-}Z \end{cases}$$

and a_i for $i = 1, \ldots, 5$ and γ are six real constants to be empirically determined.

```
a1 = 1  # initial guess
a2 = 1
a3 = 1
a4 = 1
a5 = 1
gamma = 1.5

# the functional form of weiszäcker's formula
delta(A,Z) := if(is_odd(A), 0, if(is_even(Z), +1, -1))
W(A,Z) := a1 - a2*A^(-1/3) - a3*Z*(Z-1)*A^(-4/3) - a4*(A-2*Z)^2*A^(-2) + delta(A,Z) * a5*A^(-gamma)

FUNCTION D(A,Z) FILE_PATH binding-2012.dat   # the experimental data
FIT W TO D VIA a1 a2 a3 a4 a5 gamma          # fit W to D using the six parameters

IF done_outer                                # write the result!
 PRINT "a1 = " %.3f a1 "MeV"
 PRINT "a2 = " %.3f a2 "MeV"
 PRINT "a3 = " %.3f a3 "MeV"
 PRINT "a4 = " %.3f a4 "MeV"
 PRINT "a5 = " %.3f a5 "MeV"
 PRINT "γ   = " %.3f gamma
 PRINT_FUNCTION D W D(A,Z)-W(A,Z) FILE_PATH binding-fit.dat
ENDIF
```

Listing 25: fsm.was

```
$ wasora fsm.was
a1 =    15.031  MeV
a2 =    16.059  MeV
a3 =    0.662   MeV
a4 =    20.354  MeV
a5 =    17.020  MeVγ
  =     1.621
$ gnuplot -e "set terminal pdf; set output 'fsm.pdf'; set cbrange [0:9]; set view map; set xlabel 'A'; set ylabel 'Z'; ↩
      splot 'binding-fit.dat' u 1:2:4 w p pt 50 ps 0.15 palette ti ''"
$
```

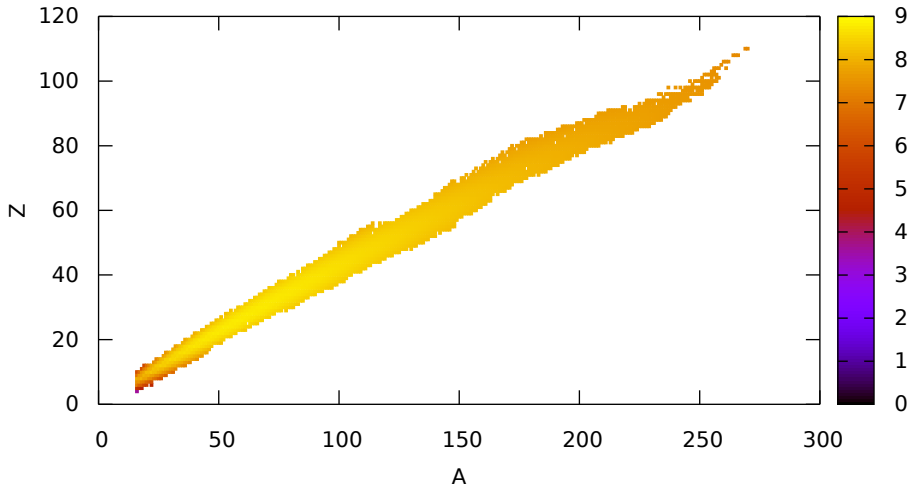


Figure 8: Binding energy per nucleon given by the semi-empirical mass formula as fitted by wasora.

In this case, measured data from the 2012 Atomic Mass Evaluation is used as the experimental data. A point-wise defined function $D(A,Z)$ is defined, reading the data from a file with three columns, namely A, Z and B/A called `binding-2012.dat`. Another algebraic function $W(A,Z)$ is defined following the proposed functional form and leaving the to-be-determined constants as parameters. Using the `FIT` keyword, we ask wasora (actually GSL) to find the values of the constants that better fit $W(A,Z)$ to $D(A,Z)$. Using

the secondary keyword `VERBOSE` we can see how the iterative procedure advances. When a result is finally obtained, we print the fit results to the standard output.

D.12 How the wasora parser works

```
# this file shows some particularities about the wasora parser

# there are primary and secondary keywords, in this case
# PRINT is the primary keyword and TEXT is the secondary, which
# takes a single token as an argument, in this case the word hello
PRINT TEXT hello

# if the text to be printed contains a space, double quotes should be used:
PRINT TEXT "hello world"
# if the text to be printed contains quotes, they should be escaped:
PRINT TEXT "hello \"world\""

# it does not matter if the argument is a string or an expression, whenever
# a certain argument is expected, either spaces are to be remove or
# the arguments should be enclosed in double quotes:
PRINT 1 + 1    # the parser will read three different keywords
PRINT "1 + 1"  # this is the correct way to compute 1+1
PRINT 1+1      # this line also works becasue there are no spaces

# you already guessed it, to insert comments, use the hash '#' character
PRINT sqrt(2)/2     # comments may appear in the same line as a keyword

# in case a hash character is expected to appear literally in an argument
# it should be escaped to prevent wasora to ignore the rest of the line:
PRINT TEXT "\# this is a commented output line"  # this is a wasora comment

# secondary keywords and/or arguments can be given in different lines either by
# a. using a continuation marker composed of a single backslash:
PRINT sqrt(2)/2 \
      sin(pi/4) \
      cos(pi/4)
# b. enclosing the lines within brackets '{' and '}'
PRINT sqrt(2)/2 {
      sin(pi/4)
      # comments may appear inside brackets (but not within continued lines)
      cos(pi/4) }

# arguments may be given in the command line after the input file
# they are referred to as $1, $2, etc. and are literally used
# i.e. they can appear as arguments or even keywords
# if a $n expressions appears in the input file but less than n
# arguments were provided, wasora complains
# this behavior can be avoided by giving a default value:
DEFAULT_ARGUMENT_VALUE 1 world
DEFAULT_ARGUMENT_VALUE 2 2

PRINT TEXT "hello $1"
PRINT sqrt($2)/$2

# try executing this input as
# $ wasora parser.was WORLD
# $ wasora parser.was WORLD 3

# if a literal dollar sign is part of an argument, quote it with a backslash:
PRINT TEXT "argument \$1 is $1"
```

Listing 26: parser.was

```
$ wasora parser.was
hello
hello world
hello "world"
1.000000e+00    +       1.000000e+00
2.000000e+00
2.000000e+00
7.071068e-01
# this is a commented output line
7.071068e-01    7.071068e-01    7.071068e-01
7.071068e-01    7.071068e-01    7.071068e-01
```

```
hello world
7.071068e-01
argument $1 is world
$
```