SOLVEr Suite for Alkalinity-PH Equations
SOLVESAPHE, Version 1.0

—

User Manual

G. Munhoven

Dépt. d'Astrophysique, Géophysique et Océanographie,
Université de Liège,
B–4000 Liège, Belgium,
eMail: Guy.Munhoven@ulg.ac.be

Manual Version 1.0 (21st February 2013)

Abstract

This manual describes the usage and the main technical aspects of the SOLVE*r Suite for Alkalinity-PH Equations* (SOLVESAPHE), Version 1.0). The codes provide a self-contained Fortan 90 implementation of the universal and robust algorithms for solving the total alkalinity-*p*H equation presented in Munhoven (2013), along with new implementations of a few other, previously published solvers, as well as some auxiliary functions and subroutines. Also included are the original driver programs that were used to produce the results reported in Munhoven (2013). The codes of SOLVESAPHE are free software and they are released under the GNU Lesser General Public Licence Version 3. Bug reports, reports about successful builds on other platforms or with other compilers than those listed below, as well as contributions are welcome!

Contents

1 Requirements

1.1 Compiler and Preprocessor

A standard compliant Fortran 90 compiler and a C-preprocessor or a compatible preprocessor (such as, e.g., fpp) are required to compile the codes. Preprocessor directives are included for enabling or disabling specific parts (debugging messages, optional code parts and variants, ...) in the solver modules. In the main driver programs, they are used to select among the cases treated in the paper.

After pre-processing, the modules' source files are strictly standard conforming Fortran 90. The codes take advantage of the configurable precision facilities offered by Fortran 90. A single change in the module mod_precision.f90 is thus sufficient to consistently use the codes in single or double precision, as required by the user.

SOLVESAPHE is self-contained and does not require any external libraries. The codes were tested with

- GNU Fortran (GCC) 4.1.2 on Red Hat 4.1.2 (x86_64)
- GNU Fortran (GCC) 4.4.5 on Debian 6.0.6 (i686)
- GNU Fortran (GCC) 4.6.2 on openSUSE 12.1 (x86_64)
- Intel Fortran 11.0 (32 bit and 64 bit versions) on Red Hat 4.1.2 (x86_64)
- Intel Fortran 12.1.3 on openSUSE 12.1 (x86_64)
- PGI Fortran compiler 8.0.2 (64 bit) on Red Hat 4.1.2 (x86_64)

1.2 Contents and summary description of `solvesaphe.tar.gz`

Core modules

`mod_phsolvers.F90`
: module containing the solvers for the total alkalinity-pH equations; also contains the functions and subroutines for the equation and its derivative

`mod_chemconst.f90`
: module with the parametrizations for the stoichiometric constants; also holds the products of the constants (the Π_j factors in the main paper) for the considered acid systems and the pH scale conversion factor (denoted s in the main paper)

`mod_acb_phsolvers.F90`
: module containing special solvers for the carbonate+borate alkalinity-pH equation

`mod_acbw_phsolvers.F90`
: module containing special solvers for the carbonate+borate+water self-ionization alkalinity-pH equation

`mod_phsolvers_logging.F90`
: extended version of `mod_phsolvers.F90` that does extra bookkeeping regarding the number of iterations, types of limiting events, etc.

Configuration Modules

`mod_precision.f90`
: module to select the working precision (REAL data type) to be used in all the source codes

Drivers and main programs

`main_check.f90`
: program to carry out the constants' value checks provided by the subroutine `checkconstants` in `mod_chemconsts.f90`

`driver_at_general.F90`
: driver for realising the timings of the seawater test cases SW1, SW2 and SW3, and of the Random Time Step (RTS) tests; uses `mod_phsolvers`

`driver_at_logging.F90`
: driver for determining the numbers of iterations for test case SW1, SW2 and SW3; uses `mod_phsolvers_logging`

`driver_at_random.F90`
: driver for the Random Total Concentration (RTC) test series; based upon `mod_phsolvers_logging`

Other files

`mod_random.f90`
: module providing two simple random number generating subroutines, one for normally and one for uniformly distributed variates, each one with zero mean and unit variance

`mod_chemspeciation.f90`
: module providing a collection of subroutines to calculate the speciation of the different acid systems considered, as a function of *p*H (not used in Munhoven (2013))

`makefile`
: makefile for building the test case programs and the stoichiometric constants' checking utility

`COPYING.LESSER`
: text of the GNU Lesser General Public Licence version 3

`COPYING`
: text of the GNU General Public Licence version 3 (underlying the GNU Lesser General Public Licence version 3)

`manual.pdf`
: this manual.

2 Building the test cases

After uncompressing and extracting the contents of the archive, e.g., with

```
$ tar xvfz solvesaphe.tar.gz
```

the various provided programs can be build with

```
$ make target
```

where *target* may be one of

`checkconsts` — for compiling `main_check.f90` and its dependencies;

`at_general` — for compiling `driver_at_general.F90` and its dependencies;

`at_logging` — for compiling `driver_at_logging.F90` and its dependencies;

`at_random` — for compiling `driver_at_random.F90` and its dependencies.

The generated executables have the same name as the target in each instance. Please notice that it may be necessary to proceed to a clean build (i.e., first `make clean`) to consistently rebuild a program after changes in a module. This is due to complications arising from the compilation of Fortran 90 modules, during which two files (the `*.o` and the `*.mod` files are generated, whereas make can only control one of them (here the `*.o`).

`checkconsts` writes its output to the file named `checkconst.log`

Specific test cases and other boundary conditions (temperature, pressure, salinity) and variants (type of initialisation, etc.) are selected by adapting the precompiler directives that can be found right after the copyright and licencing statements at the beginning of each one of the driver files. In addition, to select which one of the first or third iterates should be monitored for the Random Time Step test (RTS) with `driver_at_logging.F90`, the integer parameter `jp_lognth` in `mod_at_logging.F90` should be set to 1 or 3, respectively.

The optionally generated result files (when `#define CREATEFILES` is used, these are created) are Fortran unformatted (binary) files which can be directly read in by some post-processing applications (e.g., IDL).

3 Description and usage of the core modules

All of the solvers are implemented as `FUNCTION` sub-programs. The arguments to provide include the relevant alkalinity and total dissolved acid concentrations, and optionally, one argument (`p_hini`) to provide an initial value to start the iterations and a second one (`p_val`) to retrieve the equation residual if wanted. For floating-point calculations, all of the codes consistently use a configurable `REAL` data type that must be selected in `MOD_PRECISION` (source file `mod_precision.f90`). The identifier of that data type is stored in the `INTEGER` parameter `wp`. The default type is set to `DOUBLE PRECISION` (`wp=KIND(1D0)`). The solver `FUNCTION` sub-programs are accordingly declared to be of type `REAL(KIND=wp)` throughout.

If the optional initialisation argument is left out, the solver calls the cubic polynomial initialisation scheme described in Munhoven (2013). Each module contains an version of that initialisation routine suitable for the respective solvers. Upon completion, the solver functions either return the calculated $[H^+]$, or -1 if no such root could be found. In the latter case, `p_val` is set to `HUGE(1._wp)`.

All of the solvers use the thermodynamic constants' products (Π_j's) stored in the module `MOD_CHEMCONSTS` at the time of the call. These have to be initialised before calling the solver.

The convergence criterion used with all of the solvers requires that the relative variation of two subsequent iterates (in absolute value) falls below a given threshold. The threshold value is set by the parameter `pp_rdel_ah_target` that can be found in each module. Its value is set to 10^{-8} by default.

3.1 Module `MOD_CHEMCONSTS`

The source code of this module is in `mod_chemconsts.f90`.

It provides a basic, but comprehensive set of `FUNCTION` sub-programs (type `REAL(KIND=wp)`) to calculate the stoechiometric constants for

- the self-ionization of water (id.: `wat`)
- the dissociation series of carbonic acid (id.: `dic`)

- the dissociation of boric acid (id.: `bor`)
- the dissociation of silicic acid (id.: `sil`)
- the dissociation series of phosphoric acid (id.: `po4`)
- the dissociation of ammonium (id.: `nh4`)
- the dissociation of hydrogen sulphide (id.: `h2s`)
- the dissociation of bisulphate (id.: `so4`)
- the dissociation of hydrogen fluoride (id.: `flu`)

as a function of temperature (in Kelvin), salinity (no units) and applied pressure (in bar). All of the concentrations are supposed to be expressed in mol/kg-solution. For some acids, several parameterizations may be given. Calculations in the respective `FUNCTION` subprograms use the same *p*H scale as originally published. Auxiliary functions to convert between *p*H scales (free, total and seawater scales) are provided.

The solver modules interact with `MOD_CHEMCONSTS` only via the `api1_`*`aaa`*, `api2_`*`aaa`*, ... and the `aphscale` variables that hold the stoechiometric constants' products, i.e., the Π_j factors in the main paper. The *`aaa`* part in the names identify the respective acid systems on the basis of the three-letter codes given in brackets in the list above. `aphscale` holds the *p*H scale conversion factor s (Munhoven, 2013, eqn. (17)).

The `FUNCTION` sub-programs for calculating the individual constants are kept `PRIVATE` in the module. This helps to avoid potential misuse. A specific `SUBROUTINE` should be used to initialize the relevant `api1_`*`aaa`*, `api2_`*`aaa`*, ...and `aphscale` variables. Special attention must be paid to consistently use a common *p*H scale for the set of constants, and convert where necessary. Two sample subroutines, `SETUP_API4PHTOT` and `SETUP_API4PHSWS` respectively based upon the total and seawater scales are provided in `mod_chemconsts.f90`. For the exact names and detailed characteristics (references, *p*H scale, units, etc.), please refer to the source code file and the comments included.

`mod_chemconsts.f90` further contains `FUNCTION` sub-programs to calculate

- the solubility of CO_2 gas
- the solubility product of calcite
- the solubility product of aragonite
- the concentrations of some conservative seawater solutes as a function of salinity (calcium, boron, fluoride, sulphate)
- the density of seawater as a function of temperature, salinity and pressure.

For further information, please refer to the detailed comments in the source code.

3.2 Module MOD_PHSOLVERS

The source code of this module is in `mod_phsolvers.F90`. It provides the following solvers, suitable for solving the most complete approximations of total alkalinity (eqn. (18) in Munhoven, 2013):

`SOLVE_AT_GENERAL`
: the universal and robust algorithm from Munhoven (2013) with Newton-Raphson iterations;

`SOLVE_AT_ICACFP`
: the classical Iterative Carbonate Alkalinity Correction (ICAC) method with fixed-point iterations;

`SOLVE_AT_BACASTOW`
: the variant of the previous proposed by Bacastow (1981) that uses secant instead of the fixed-point iterations – for most practical applications this is the fastest routine;

`SOLVE_AT_GENERAL_SEC`
: the variant of `SOLVE_AT_GENERAL` that uses secant instead of Newton-Raphson iterations;

`SOLVE_AT_OCMIP`
: the re-implementation of the solver originally provided for the Ocean Carbon Cycle Model Intercomparison Project Phase 2 (OCMIP-2, Orr et al., 2000), with minimal validity checks added (e.g., for bracketing values provided) and with the same optional automatic initialisation scheme as the other solvers, and functionally equivalent to the original for the rest;

`SOLVE_AT_FAST`
: the variant of `SOLVE_AT_GENERAL` that does not include the convergence control devices — slightly faster but possibly divergent.

The module furthermore contains the following auxiliary sub-programs:

`ANW_INFSUP`
: a subroutine to calculate the infimum and the supremum of $Alk_{\mathrm{nW}}([\mathrm{H}^+])$, i.e., of the total alkalinity component not related to water self-ionization — these are required to calculate the safe bounds;

`EQUATION_AT`
: a function sub-program to evaluate the rational function form of the total alkalinity-pH equation, and, optionally also its derivative;

`AHINI_FOR_AT`
: a subroutine that implements the cubic polynomial based initialisation scheme, extended such as to return (units are mol/kg)

  - 10^{-3} if the provided alkalinity is lower than or equal to 0

- 10^{-10} if the provided alkalinity is greater than or equal to $2C_T + B_T$
- 10^{-7} if the provided alkalinity value is within those bounds, but the second order approximation does not have a solution
- the root of the second order approximation around the location of the minimum of the cubic (see Munhoven (2013) for details) else.

`AC_FROM_AT`
: a function to estimate the carbonate alkalinity from total alkalinity for a given $[H^+]$ – required by `SOLVE_AT_ICACFP` and `SOLVE_AT_BACASTOW`;

`SOLVE_AC`
: a function to solve the quadratic that relates $[H^+]$ to total dissolved inorganic carbon and carbonate alkalinity – required by `SOLVE_AT_ICACFP` and `SOLVE_AT_BACASTOW`.

The module offers a few customization options:

- the precompiler token `VARIANT_BACASTOWORIG` can be used to select one of two variants of Bacastow's method:
  - with `#define VARIANT_BACASTOWORIG`, the original version, where secant iterations are carried out for $X = \sqrt{K_1 K_2}/[H^+]$ as the variable, is adopted;
  - with `#undef VARIANT_BACASTOWORIG` (this is default), the secant iterations are carried out for $[H^+]$ directly.
- the maximum number of iterations allowed for each method is controlled by the parameters `jp_maxniter_`*`idmethod`*, where the method identificator *`idmethod`* should be substituted by
  - `atgen` for `SOLVE_AT_GENERAL`
  - `icacfp` for `SOLVE_AT_ICACFP`
  - `bacastow` for `SOLVE_AT_BACASTOW`
  - `atsec` for `SOLVE_AT_GENERAL_SEC`
  - `ocmip` for `SOLVE_AT_OCMIP`
  - `atfast` for `SOLVE_AT_FAST`

  All of these are set to 50 by default.

After the call, the actual number of iterations performed can be retrieved from the `niter_`*`idmethod`* variable related to the used solver, and that is provided in the module (with *`idmethod`* as above). For other details, such as the number, order and type of arguments in the solver function sub-programs, please refer to the comments in the source code.

3.3 Module MOD_ACBW_PHSOLVERS

The source code of this module is in `mod_acbw_phsolvers.F90`. It provides the following solvers, based up the (eqns. (13) or (14) in Munhoven, 2013):

`SOLVE_ACBW_GENERAL`
: a simplified version of `SOLVE_AT_GENERAL`, considering carbonate, borate and water self-ionization contributions to total alkalinity only (based upon the rational function version of the equation).

`SOLVE_ACBW_POLY`
: a solver based upon the quintic equation (eqn. (14) in Munhoven, 2013), using a hybrid Newton-Raphson–bisection method similar to that used in `SOLVE_AT_GENERAL` to ensure convergence;

`SOLVE_ACBW_POLYFAST`
: a variant of the previous without the convergence control device;

`SOLVE_ACBW_ICACFP`
: a simplified version of `SOLVE_AT_ICACFP`, considering carbonate, borate and water self-ionization contributions to total alkalinity only;

`SOLVE_ACBW_BACASTOW`
: a simplified version of `SOLVE_AT_BACASTOW`, considering carbonate, borate and water self-ionization contributions to total alkalinity only.

The module also contains the following auxiliary program:

`AHINI_FOR_ACBW`
: an implementation of the special initialisation scheme for the cubic equation, extended such as to derive an starting value from the classical root bounds of Cauchy (scaled here) or Kojima that provide estimates for the radius of the disc in the complex space that contains all the roots of a polynomial (Stoer, 1989) in case the parabolic expansion around the minimum does not provide a valid starting value.

`ACBW_HINFSUP`
: calculates the upper and lower bounds for the root of the equation following Munhoven (2013, section 5.1)

`SOLVE_AC`
: a function to solve the quadratic that relates $[H^+]$ to total dissolved inorganic carbon and carbonate alkalinity – required by `SOLVE_ACBW_ICACFP` and `SOLVE_ACBW_BACASTOW`.

In comparison with the solvers provided in `mod_phsolvers.F90`, these are simplified (e.g., the approximations $[H^+]_T = [H^+]_f + [HSO_4^-]$ and $[H^+]_{SWS} = [H^+]_f + [HSO_4^-] + [HF]$ are used) and most functions calls are inlined.

3.4 Module MOD_ACB_PHSOLVERS

The source code of this module is in `mod_acbw_phsolvers.F90`. It provides the following three solvers for the equation based upon Alk_{CB} (eqns. (11) or (12) in Munhoven, 2013) or eqn. 1) below:

`SOLVE_ACB_POLY`
: a solver for the cubic equation, based upon a hybrid Newton-Raphson–bisection method similar to that used in `SOLVE_AT_GENERAL` to ensure convergence;

`SOLVE_ACB_POLYFAST`
: a variant of the previous without the convergence control device;

`SOLVE_ACB_GENERAL`
: a simplified version of `SOLVE_AT_GENERAL`, considering carbonate and borate alkalinity only (based upon the rational function version of the equation).

The module also contains the following two auxiliary programs:

`AHINI_FOR_ACB`
: an implementation of the special initialisation scheme for the cubic equation, identical to `AHINI_FOR_ACBW` above;

`ACB_HINFSUP`
: a subroutine to determine a bracketing interval for the root of the equation, following the lines detailed in the next section.

For this module, the same simplifications as in `MOD_ACB_PHSOLVERS` described above are adopted.

Defining brackets for the root

In the subroutine `ACB_HINFSUP`, we calculate a pair of brackets for the root along the following lines. Recall that, for the Alk_{CB} approximation to total alkalinity, the equation to solve for deriving pH is

$$R_{\mathrm{CB}}([\mathrm{H}^+]) \equiv C_{\mathrm{T}} \frac{K_1[\mathrm{H}^+] + 2K_1K_2}{[\mathrm{H}^+]^2 + K_1[\mathrm{H}^+] + K_1K_2} + B_{\mathrm{T}} \frac{K_{\mathrm{B}}}{[\mathrm{H}^+] + K_{\mathrm{B}}} - Alk_{\mathrm{CB}} = 0. \quad (1)$$

As shown by Munhoven (2013), equation (1) has a positive root if and only if $0 < Alk_{\mathrm{CB}} < 2C_{\mathrm{T}} + B_{\mathrm{T}}$; if there is a positive root, it is unique.

Let us denote the positive root of

$$C_{\mathrm{T}} \frac{K_1[\mathrm{H}^+] + 2K_1K_2}{[\mathrm{H}^+]^2 + K_1[\mathrm{H}^+] + K_1K_2} - \frac{2C_{\mathrm{T}}}{2C_{\mathrm{T}} + B_{\mathrm{T}}} Alk_{\mathrm{CB}} = 0.$$

by H_{C} and the root of

$$B_{\mathrm{T}} \frac{K_{\mathrm{B}}}{[\mathrm{H}^+] + K_{\mathrm{B}}} - \frac{B_{\mathrm{T}}}{2C_{\mathrm{T}} + B_{\mathrm{T}}} Alk_{\mathrm{CB}} = 0,$$

by H_B. The two roots exist and are unique again because $0 < Alk_{CB} < 2C_T + B_T$. Since the two first terms of $R_{CB}([H^+])$ are both monotonously decreasing functions (Munhoven, 2013), then

$$H_{inf} = \min(H_B, H_C) \quad \text{and} \quad H_{sup} = \max(H_B, H_C)$$

provide the required brackets. Indeed, because $R_{CB}([H^+])$ is a decreasing function,

$$\begin{aligned} R_{CB}(H_{inf}) &= C_T \frac{K_1 H_{inf} + 2K_1K_2}{H_{inf}^2 + K_1 H_{inf} + K_1K_2} + B_T \frac{K_B}{H_{inf} + K_B} - Alk_{CB} \\ &\geq C_T \frac{K_1 H_C + 2K_1K_2}{H_C^2 + K_1 H_C + K_1K_2} + B_T \frac{K_B}{H_B + K_B} - Alk_{CB} \\ &= \frac{2C_T}{2C_T + B_T} Alk_{CB} + \frac{B_T}{2C_T + B_T} Alk_{CB} - Alk_{CB} = 0 \end{aligned}$$

and

$$\begin{aligned} R_{CB}(H_{sup}) &= C_T \frac{K_1 H_{sup} + 2K_1K_2}{H_{sup}^2 + K_1 H_{sup} + K_1K_2} + B_T \frac{K_B}{H_{sup} + K_B} - Alk_{CB} \\ &\leq C_T \frac{K_1 H_C + 2K_1K_2}{H_C^2 + K_1 H_C + K_1K_2} + B_T \frac{K_B}{H_B + K_B} - Alk_{CB} \\ &= \frac{2C_T}{2C_T + B_T} Alk_{CB} + \frac{B_T}{2C_T + B_T} Alk_{CB} - Alk_{CB} = 0. \end{aligned}$$

3.5 Debugging output

Debugging output from the solvers in the three modules may be activated by passing the option `-DDEBUG_PHSOLVERS` to the Fortran compiler. In the provided `makefile`, this can easily be done by adding `-DDEBUG_PHSOLVERS` to the `OPTIONDEFS` token. Please notice, however, that it is not recommended to use this option with the test cases as they would generate many Gigabytes of debugging information.

3.6 Practical usage: typical sequence

After the user has chosen a suitable solver from one of three modules (either `mod_phsolvers.F90` for the complete version, or `mod_acb_phsolvers.F90` or `mod_acbw_phsolvers.F90` for the simplified versions), the steps required to include it into your own program are as follows.

1. Make sure the adequate precision is selected in the `MOD_PRECISION` module (`mod_precision.f90`).

2. Make your choice for the stoichiometric constants (i.e., chose *p*H scale, etc.): either use one of the `SETUP_API_...` routines that are provided in `mod_chemconsts.f90`, adapt one of them or create a new one – make sure it initializes all of the required `api_...` variables in `mod_chemconsts.f90` and, if necessary, also the `aphscale` variable for the *p*H scale conversion.

3. In the scoping unit of your code that requires *p*H calculation, include the Fortran directives

   ```
   USE MOD_PRECISION
   USE MOD_CHEMCONSTS
   USE MOD_XXXX_PHSOLVERS
   ```

   where *MOD_XXXX_PHSOLVERS* must be replaced by the name of the module that contains the chosen solver (MOD_PHSOLVERS, MOD_ACB_PHSOLVERS or MOD_ACBW_PHSOLVERS). If the exact speciation of athe acid systems are also required, one may furthermore include

   ```
   USE MOD_CHEMSPECIATION
   ```

4. Before each call of the solver, make sure that the set of chemical constants (for the desired temperature, salinity and pressure) is correctly initialized by calling the adequate SETUP_API... subroutine.

5. Call the chosen solver function.

6. If required, call the relevant SPECIATION_*aaa*(...) subroutines from MOD_CHEMSPECIATION to calculate the actual speciations (where the *aaa* parts in the subroutine names identify the respective acid systems on the basis of the three-letter codes given in brackets in section 3.1 above). Notice that the SPECIATION_*aaa*(...) subroutines rely upon the api1_*aaa*, api2_*aaa*, ...values that were used to calculate *p*H. Their values must therefore not be changed before the speciation calculations for the sake of consistency.

3.7 Streamlining, extensions, ...

Recommended streamlining

In case a large number of *p*H calculations for many different temperature, salinity and pressure conditions are required, it is recommended to restrict the SETUP_API_... routine to the strict minimal set of stoichiometric constants required to solve the problem. Because of the exponential or power function evaluations required for the calculation of each single of the chemical constants, calculating even the bare minimum set of constants may take a significant fraction of the total time needed to complete one *p*H determination.

How to extend it

SOLVESAPHE is obviously extensible: users may add extra acid systems if needed. This task is not complicated, but requires changes at different places.

1. First, decide to which extent the dissociation of considered acid is going to be taken into account (define the number of dissociations n and the integer m that sets the zero proton level of the system — see Munhoven (2013) for more details).

2. Add function subprograms in `mod_chemconsts.f90` to make available the required chemical constants. All of the routines in SOLVESAPHE expect concentrations to be expressed in mol/kg. The arguments of the subroutine functions should adhere to the common structure `t_k, s, p_bar` (in this order), where `t_k` is the temperature in Kelvin, `s` is salinity and `p_bar` the applied pressure in bar. It is recommended to keep the actual functions that evaluate the parametrizations private in the module.

3. Add the `api1_`*`aaa`*, `api2_`*`aaa`*, ... variables to `mod_chemconsts.f90`, chosing a unique identificator *`aaa`* for the new contributing acid system.

4. If required, add a new speciation subroutine `SPECIATION_`*`aaa`*`(...)` to `mod_chemspeciation.f90`, distinguished by the unique identificator *`aaa`* for the new contributing acid system.

5. Prepare a `SETUP_API_...` routine to initialize all of the `api*` variables of all the acid systems considered (incl water self-ionization and the *p*H scale conversion factor `aphscale`.

6. Amend `mod_at_phsolvers.F90`, depending on the type of solver used.

   - If the used solver is `SOLVE_AT_GENERAL`, `SOLVE_AT_GENERAL_SEC` or `SOLVE_AT_FAST`, take the effect of the additional acid system into account by amending `ANW_INFSUP` and `EQUATION_AT` (add the dummy `p_`*`aaa`*`tot` to the argument lists and adapt the code), then include `p_`*`aaa`*`tot` in the argument list of the solver function sub-program and adapt the calls to `ANW_INFSUP` and `EQUATION_AT`;

     if your solver is one of `SOLVE_AT_ICACFP` or `SOLVE_AT_BACASTOW`, then take the effect of the additional acid system into account by amending `AC_FROM_AT` (add the dummy `p_`*`aaa`*`tot` to the argument lists and adapt the code), then include `p_`*`aaa`*`tot` in the argument list of the solver function sub-program and adapt the call to `AC_FROM_AT`.

7. Change the calls to the new `SETUP_API_...` (only if its name has changed, the argument list should have remained the same) and adapt the list of arguments wherever the solver is called in the main program to make it conformant with the changes made in the module.

History of this document

21st February 2013
: Initial release (version 1.0 of the manual)

References

Bacastow, R.: Numerical evaluation of the evasion factor, in: Carbon Cycle Modelling, edited by Bolin, B., vol. 16 of *SCOPE*, chap. 3.4, pp. 95–98, John Wiley & Sons, Chichester, NY, 1981.

Munhoven, G.: The Mathematics of the Total Alkalinity–*p*H Equation: Pathway to Robust and Universal Solution Algorithms, Geoscientif. Model Devel. Discuss., 2013.

Orr, J., Najjar, R., Sabine, C., and Joos, F.: Abiotic-HOWTO, URL `http://ocmip5.ipsl.jussieu.fr/OCMIP/phase2/simulations/Abiotic/HOWTO-Abiotic.html`, (retrieved on 20th July 2012), 2000.

Stoer, J.: Numerische Mathematik 1, Springer-Verlag, Berlin, 5 edn., 1989.