SKI Technical Report 92:21

SKI-TR--92-21.

SOLISOL-handling of Solid Solutions. Version 1.1

S. Börjesson
A. Emrén

September 1992

SKi
STATENS KÄRNKRAFTINSPEKTION
SWEDISH NUCLEAR POWER INSPECTORATE

SOLISOL-handling of solid solutions. Version 1.1

S. Börjesson - A. Emrén

SKI TR 92:21

Department of Nuclear Chemistry
Chalmers University of Technology
S-412 96 Göteborg

September 1992

"SOLISOL-handling of solid solutions. Version 1.1"

Susanne Börjesson and Allan Emrén

Department of Nuclear Chemistry
Chalmers University of Technology
September 1992

Table of Contents

Table of Contents (Contd.)

ABSTRACT

SOLISOL is a C computer program designed to model geochemical reactions involving solid solutions. The program searches equilibrium concentrations of the components in the aqueous phase and the solid solution given by limited quantities of the solid solution components.

The equilibrium code PHREEQE is used as a subprogram in SOLISOL. Subprograms external to PHREEQE extract information from PHREEQE results, take care of conserved properties, calculate solubilities and produce inputdata for PHREEQE. The essential idea in this process is to calculate solubilities for the components in terms of saturation indices, and give directions to PHREEQE on how to search for the equilibrium under those constraints.

SUMMARY

Geochemical computer models can be used to increase the knowledge about reactions that may occur in connection with an underground repository. The models used are often simplified due to limited computer capacity

The occurrence of solid solutions is an important phenomenon that can be found not only in the bedrock but also around a leaking repository for nuclear waste, e.g. co-precipitation. A radionuclide sorbed upon a precipitating mineral surface will cause a solid solution to be formed since the mineral then will contain trace quantities of the radionuclide.
Spent nuclear fuel, cement and mineral phases like chlorite and plagioclase can all be described in terms of solid solutions.

This report presents a computer program, SOLISOL, which can handle solid solutions in geochemical calculations. The program is constructed to use the equilibrium code PHREEQE as a subprogram to which the main program produces indata and directives. Subprograms external to PHREEQE check results and use these to search equilibrium within the constraints given by the solubilities and quantities of the components.

Normally, the solubility of a component in a solid solution is smaller than that of the corresponding pure solid. Let us assume that dissolution of a pure solid, A, results in a number, n, of solution species, denoted by A_i. We may then describe a solid solution in equilibrium with an aqueous phase by

$$\mu_A^{\text{solid solution}} = \sum_{i=1}^{n} (\mu_{A_i}^0 + RT \ln a_{A_i}^s)$$

$\mu_A^{\text{solid solution}}$ defines the chemical potential of component A in a solid solution. The symbol 0 denotes the standard state of a species, A_i, in solution and $a_{A_i}^s$ the activity of an aqueous species in equilibrium with a solid solution. If the left hand side is rewritten in terms of mole fraction, X, and activity coefficient, γ, of the component A in the solid solution, the result will be

$$\lg X_A \gamma_A = SI_A$$

Here SI_A denotes the saturation index of component A.
The program SOLISOL is written in the programming language C. By calling the subprograms PHREEQE, SOL1 and SOL2, SOLISOL searches for the equilibrium concentrations of the two phases, cf. Figure 1.

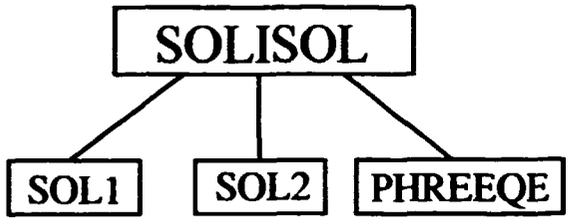


Figure 1. Hierarchical diagram of SOLISOL.

PHREEQE is used to solve equilibrium equations. The main task of the other two sub programs is to check results produced by PHREEQE and use the information to search equilibrium within the constraints given by limited quantities of the components in the solid solution.

The report also contains user instructions on how to run the SOLISOL program as well as a program list.

SAMMANFATTNING

Geokemiska datormodeller kan användas för att öka kunskapen om de kemiska reaktioner som kan förekomma i samband med ett underjordiskt förvar. De använda modellerna är ofta förenklade på grund av begränsad datorkapacitet.

Förekomsten av fasta lösningar är ett viktigt fenomen som kan förekomma inte bara i berget utan också kring ett läckande förvar med kärnavfall, t ex medfällning. En radionuklid som sorberas på en utfallande mineralyta kommer att bilda en fast lösning då mineralet kommer att innehålla spårmängder av radionukliden.

Denna rapport presenterar ett program, SOLISOL, som kan hantera fasta lösningar i geokemiska beräkningar. Programmet är konstruerat efter samma riktlinjer som CHECKMATE och CRACKER vilket betyder att jämviktskoden PHREEQE används som ett subprogram till vilken huvudprogrammet producerar indata och direktiv. Subprogram utanför PHREEQE kontrollerar resultaten och använder dem till att söka jämvikt inom de begränsningar som ges av komponenternas löslighet.

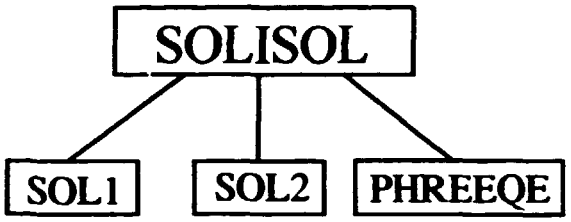
Vanligtvis är lösligheten av en komponent i en fast lösning mindre än den för den motsvarande rena fasta fasen. Låt oss antaga att upplösningen av en ren fast fas, A, resulterar i ett antal, n, specier i lösningen, betecknade med A_i. Vi kan sedan beskriva en fast lösning i jämvikt med en vattenfas genom ekvationen

$$\mu_A^{\text{solid solution}} = \sum_{i=1}^{n} (\mu_{A_i}^0 + RT \ln a_{A_i}^s)$$

$\mu_A^{\text{solid solution}}$ definierar den kemiska potentialen av komponent A i den fasta lösningen.
Symbolen 0 betecknar standardtillståndet av en specie i vätskefasen och $a_{A_i}^s$ aktiviteten av komponenten A_i in jämvikt med en fast lösning. Om vänsterledet skrivs om med hjälp av molbråk, X, och aktivitetsfaktor, γ, av komponent A i den fasta lösningen blir resultatet

$$\lg X_A \gamma_A = SI_A$$

Här betecknar SI_A mättnadsindex för komponent A i den fasta lösningen.
Programmet SOLISOL är skrivet i programspråket C. Genom att kalla på subprogrammen PHREEQE, SOL1-och SOL2, söker SOLISOL jämviktskoncentrationerna i de två faserna, jfr. figur 1.



Figur 1. Hierarkisk uppbyggnad av SOLISOL.

PHREEQE används till att lösa jämviktsekvationer. Huvuduppgiften för de andra två subprogrammen är att kontrollera resultatet producerat av PHREEQE och använda informationen för att söka jämvikt inom de restriktioner som ges av att komponeneterna i den fasta lösningen förekommer i begränsande mängder .

Rapporten innehåller även instruktioner för användning av SOLISOL och en programlista.

INTRODUCTION

In the assessment of safety of nuclear waste repositories computer simulations of several kinds are needed. To simulate different chemical phenomena that may occur in a leaking repository for spent nuclear fuel several computer codes are required. "Simple" codes as well as coupled codes are needed.

Since the computer capacity often is limited, an accessible way is to use codes that use simplified models of the underlying chemistry or physics.

The occurrence of solid solutions is an important phenomenon that can be found not only in the bedrock but also around a leaking repository, e.g. co-precipitation. A radionuclide sorbed upon a precipitating mineral surface will cause a solid solution to be formed since the mineral then contains trace quantities of the radionuclide.
Spent nuclear fuel, cement and mineral phases like chlorite and plagioclase can all be described as solid solutions.

The equilibrium code PHREEQE [PAR 84] is often used in geochemical calculations. It is well tested and it solves equilibrium problems comparatively fast [PUI 90]. Unfortunately, PHREEQE can not handle solid solutions. A way of using PHREEQE in calculations concerning solid solutions is to modify the code [MUL 85].
A quite different approach has been used in some programs, like CHECKMATE [MUL 85] and CRACKER [EMR 91], to solve other kinds of problems. The programs mentioned here are coupling chemistry and transport, and they both use PHREEQE as a subprogram to which the main program produces inputdata and directives.

This report presents a computer program, SOLISOL, which is able to handle solid solutions in geochemical calculations. The SOLISOL program is constructed along the same guidelines as the programs CHECKMATE and CRACKER.
In SOLISOL PHREEQE is used as a subprogram and subprograms external to PHREEQE check results and use these to search equilibrium within the constraints given by the solubilities and quantities of the components.

THEORY

Normally, the solubility of a component in a solid solution is smaller than that of the

corresponding pure solid. Let us assume that dissolution of a pure solid, A, results in a number, n, of solution species, denoted by A_i. We may describe the saturated solution in terms of chemical potentials.

$$\mu_A^* = \sum_{i=1}^{n} (\mu_{A_i}^0 + RT \ln a_{A_i}^p) \quad (1)$$

The symbol * denotes the standard state of A (pure solid), 0 the standard state of a species in solution and $a_{A_i}^p$ the activity of component A_i in equilibrium with a pure substance. We may rearrange this expression as

$$\mu_A^* - \sum_{i=1}^{n} \mu_{A_i}^0 = \sum_{i=1}^{n} RT \ln a_{A_i}^p \quad (2)$$

In the case that A is a component of a solid solution with the components A, B, C, ..., we may describe the equilibrium between the solid solution and an aqueous phase by an equation corresponding to (1) as

$$\mu_A^{\text{solid solution}} = \sum_{i=1}^{n} (\mu_{A_i}^0 + RT \ln a_{A_i}^s) \quad (3)$$

Here $a_{A_i}^s$ denotes the activity of an aqueous species in equilibrium with a solid solution. Similarly, the chemical potential of A in the solid solution itself is described by

$$\mu_A^{\text{solid solution}} = \mu_A^* + RT \ln a_A^{(\text{solid solution})} \quad (4)$$

At equilibrium, the chemical potential of the component is the same in the two phases. Thus, combining (3) and (4) we get

$$RT \ln a_A^{(\text{solid solution})} = -\mu_A^* + \sum_{i=1}^{n} \mu_{A_i}^0 + RT \sum_{i=1}^{n} \ln \mu_{A_i}^s \quad (5)$$

Inserting (2) in (5) gives

$$\ln a_A^{(\text{solid solution})} = \sum_{i=1}^{n} (\ln \mu_{A_i}^s - \ln a_{A_i}^p) \quad (6)$$

which may be rewritten as

$$\lg a_A^{(\text{solid solution})} = \lg \frac{\prod_{i=1}^{n} a_{A_i}^s}{\prod_{i=1}^{n} a_{A_i}^p} \quad (7)$$

The right hand side is by definition the saturation index, SI, of the solution with respect to the pure substance A. If the left hand side is rewritten in terms of mole fraction, X, and activity coefficient, γ, of the component A in the solid solution, the result finally is

$$\lg X_A \gamma_A = SI_A \quad (8)$$

Often the activity coefficients for components of solid solutions are unknown. Thus, the present version of SOLISOL assumes them to be unity.

MODEL

The code SOLISOL has been developed to take care of solid solutions in equilibrium calculations since the standard version of PHREEQE can not handle that type of problem. The SOLISOL program makes use of the saturation indices of the components in the solid solution to reach equilibrium between the solid solution and the aqueous phase.

For a pure solid phase the saturation index equals zero at equilibrium with an aqueous phase. The saturation index of a component in a solid solution will deviate from zero at equilibrium. So if the PHREEQE program is to be used in equilibrium calculations involving solid solutions, the saturation index must be operated from outside the program.

Between consecutive iteration steps, to reach equilibrium, the indata file for PHREEQE is modified, that means, new values of the saturation indices are determined. Further, the conditions, for example pH, and the composition of the solution may be changed.

The PHREEQE standard code can not be given constraints concerning limited quantities of the components in the solid solution. Consequently PHREEQE might try to dissolve more of a component than is available. Due to this circumstance the SOLISOL program has to treat easily soluble components and components with low solubility in different ways.

Low solubility

A component with low solubility is not likely to dissolve completely in a single step and the solubility can be given directly in terms of the saturation index.

From known quantities of the components in the solid solution, SOLISOL calculates the mole fraction of each component and hence the saturation index, SI, is known for all the components, cf. equation 8.

The quantities needed for the calculation are determined from the result in an earlier iteration step. SOLISOL checks the result file to find the dissolved or precipitated quantity of each component. The observed amount is then subtracted or added to the quantity of the component before the last step. These new calculated quantities are stored in the file QUANTITY.DAT and used in the calculation of the saturation indices.

An indata file for PHREEQE is created with a modified solution and the new calculated saturation indices.

High solubility

Apart from components with high solubility (large K_S), components that are completely dissolved by PHREEQE in one iteration step can also be considered as easily soluble.

If SOLISOL finds that too much has been dissolved of a component, a flag is assigned to that component in the quantity file, QUANTITY.DAT.

Reactions are directed in PHREEQE by option 3 in the indata file. For a more detail information please look in a PHREEQE manual [PAR 84]. If this option is assigned the value 5, solid phases are equilibrated with the solution. When handling easily soluble components, this method can not be used. The problem is solved by assigning the value 4 to option 3, which means that the component is added to the aqueous phase by a net stoichiometric reaction.

SOLISOL searches in the result file of PHREEQE for the head line LOOK MIN where the saturation state, SI, of the solution with respect to the desired component is to be found. The amount to be added to the solution is the present quantity of the component minus the amount obtained from the saturation state when using equation 8. SOLISOL then creates an indata file for PHREEQE with the new option but retains the solution and the conditions from the previous iteration.

When the easily soluble component eventually reaches equilibrium, the solution obtained from this last step is equilibrated with both labelled and unlabelled components. If the system still hasn't reached equilibrium, the two procedures described above are repeated.

GENERAL PROGRAM DESCRIPTION

The program SOLISOL is written in the language C. By calling the subprograms PHREEQE, SOL1 and SOL2, SOLISOL searches for the equilibrium concentrations of the two phases (Figure 1).

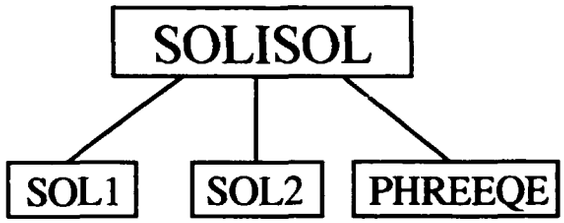


Figure 1. Hierarchical diagram of SOLISOL.

PHREEQE is used to solve equilibrium equations. The main task of the other two sub programs is to check results produced by PHREEQE and use the information to search equilibrium within the constraints given by limited quantities of the components in the solid solution.

SOLISOL starts by calling PHREEQE to perform a speciation calculation of the solution. At that stage no reaction is modeled. The result produced by PHREEQE is stored in the file OUTDATA. The next subprogram to be called is SOL1. This program extracts data from the file OUTDATA and among others makes use of mineral data in the files QUANTITY.DAT and *.MIN to create an indata file for PHREEQE.

After SOLISOL has called PHREEQE again the subprogram SOL2 checks the result, tests if equilibrium has been reached and if not, creates a modified indata file for PHREEQE that will be called again by SOLISOL (cf. Figure 2).

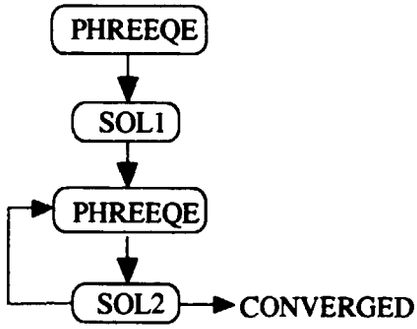


Figure 2. Flowchart of SOLISOL.

The main outlines of the subprogram SOL2 is shown in Figure 3. The SOL1 program has a more simple content consisting mainly of the functions; Read_PHREEQE_result, Saturation_index and a less complex Make_PHREEQE_file.

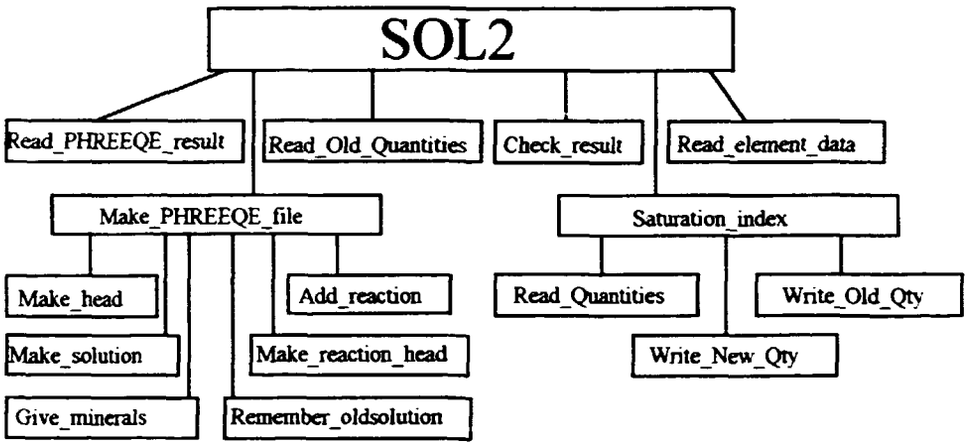


Figure 3. Hierarchical diagram of SOL2.

USER INSTRUCTIONS

The program is available from the Department of Nuclear Chemistry, Chalmers University of Technology, S-412 96 Göteborg, Sweden, Attn: S. Börjesson/ A. Emrén.
Fax number: +46 31 772 29 31.

The SOLISOL program is an MS-DOS program distributed on $3^1/2"$ or $5^1/4"$ diskettes. The distribution disk contains compiled programs as well as source codes, test example and data base.
The compiled version of SOLISOL is compiled for IBM compatible personal computers containing 80286 and 80287 processors or higher.

Installation

To install the SOLISOL program:
1. Insert the distribution disk in the proper drive, *A* or *B*.
2. Type:
a:\installa
or in the case of a *B* drive:
b:\installb
A directory, SOLISOL, is created and all necessary files are installed into that directory. The following text will then be seen on the screen:
C:\SOLISOL\WORK>.

Test problem

To check that the installation has been successful a test problem, equilibrating cement with synthetic groundwater [BÖR 92], may be run. In the test problem cement is considered as a solid solution consisting of the two components $Ca(OH)_2(s)$ and $CaH_2SiO_4(s)$. As can be seen, this is a simplified model of cement.

To start the program give the command *SOLISOL*. The progress of the simulation can be followed upon the screen. The first text to be seen is:

```
Stop-Program terminated
Found TOTAL MOLALITIES
Found ELEMENTS
```

Blocks of similar text are repeated 19 times. The last block will be concluded with the text; **CONVERGED**.
In the file QUANTITY.DAT the remaining quantities of the components $Ca(OH)_2(s)$ and $CaH_2SiO_4(s)$ at equilibrium are found.
If the command *TYPE QUANTITY.DAT* is given, the following text will be seen on the screen:

```
2
CA(OH)2
2.099678e-01
CAH2SIO4
3.540677e+02
```

The composition and condition of the solution can be found in the result file OUTDATA, which is a standard outdata file from PHREEQE.

To run the test program again the files TEST.DAT and TEST.QTY has to be copied and renamed to INFILE.DAT and QUANTITY.DAT, respectively.

Description of input

To use the SOLISOL program the following files have to be prepared, checked or modified; INFILE.DAT, QUANTITY.DAT, HEADER.DAT, REAHEAD.DAT, *.MIN and *.REA. A description of the files is given below:

INFILE.DAT
The file INFILE.DAT is a standard indata file of PHREEQE, describing the solution to be used in the simulation. NOTE! The indata file must be balanced with respect to the charge and the concentrations must be given in molalities.

Below is an example of what a PHREEQE indata file for SOLISOL may look like. Since no reaction is to be modeled the third option is assigned the value zero.(Line two, digit number three).

```
STANDARD
000001000 0 0        .00000
SOLUTION 1
STANDARD
10 0 0      8.200    5.430    20.000  1.000
  4 2.5000E-04   5 8.8000E-05   6 1.6885E-03   7 7.1600E-05   8 5.3700E-06
 10 1.1100E-07  13 3.3200E-04  14 2.2100E-03  15 1.0000E-16  16 1.0900E-04
END
```

For further explanation of the figures please look in a PHREEQE manual.

QUANTITY.DAT

In the file QUANTITY.DAT the number of components in the solid solution is given as well as the name and quantity of each component. The mineral name or formula, must be identical to that found in the library file and in the files .MIN and .REA and the quantity be given in mmole. Each value has it own line. An example of a quantity file is given below:

```
4
CA(OH)2
787.6
CAH2SIO4
342.9
NAOH
6.46
KOH
23.36
```

HEADER.DAT, REAHEAD.DAT,NAMES

Files containing directions for PHREEQE. They should not normally be modified by the user.

LIBRARY.DAT

The PHREEQE library file must have this name to be recognised by the program. The normal limitation of number of species, element numbers and look minerals are valid since a standard version of PHREEQE is used.

MINERAL directory

The subdirectory MINERAL contains two kinds of files; *.MIN and *.REA. Each mineral/component mentioned in the QUANTITY.DAT file has to be present in this subdirectory as a .MIN and .REA file.

In the .MIN file thermodynamic data for the mineral or component "*" is given. Usually, the

.MIN files are created by making a copy of a few lines from the database.
The .REA files contain parts of the information needed to describe dissolution of the mineral "*" in the language used by PHREEQE to define a REACTION.

FILES AND FUNCTIONS

Directory description

<SOLISOL>
  <SOURCE>
  <WORK>
    <MINERAL>

Files description

SOURCE directory

The three first files listed below are source codes compilable with the Borland Turbo C compiler.

SOLISOL.C	The main program.
SOL1.C	Checks result from PHREEQE and creates an indata file for PHREEQE.
SOL2.C	Checks result from PHREEQE and modifies an indata file for PHREEQE.
ALLAN.H	Contains a number of tools, needed only if the source code has to be compiled.

WORK directory

SOLISOL.EXE	The main program.
SOL1.EXE	Checks result from PHREEQE and creates an indata file for PHREEQE.
SOL2.EXE	Checks result from PHREEQE and modifies an indata file for PHREEQE.
PHREEQE.EXE	The PHREEQE program.
HEADER.DAT	Contains the PHREEQE options for equilibrating a solid phase with a solution.
INFILE.DAT	The indata file for PHREEQE.
INFILE.TMP	A temporary indata file for PHREEQE.
LIBRARY.DAT	The PHREEQE database. The library file on the distribution disk is

	extracted from HATCHES [CRO 87].
QUANTITY.DAT	Contains the number of components in the solid solution as well as the formula, or name, and quantity of each component.
QUANTITY.OLD	Similar as QUANTITY.DAT.
QUANTITY.SI	Contains the quantities needed for the calculation of the saturation indices.
REAHEAD.DAT	Contains the PHREEQE options for adding a stoichiometric reaction to a solution.
OUTDATA	Result from PHREEQE.
START.DAT	The indata file needed to run the test problem.
TEST.QTY	The quantity file needed to run the test problem.
NAMES	A file containing names of the resultfile, indatafile and libraryfile.
MINERAL subdirectory	
*.MIN	Contains the mineral data extracted from the library file.
*.REA	Contains the number and the operative valence of the master species to be used if a stoichiometric reaction is to be added to the solution.

Important functions

Add_reaction()	Generates the REACTION group and modifies the option line in a suitable way.
Give_minerals()	Generates the MINERALS group together with the SI value.
Make_PHREEQE_file()	Generates the indata file for PHREEQE.
Search_SI_utdata()	Checks the saturation state of the solution with respect to the desired components in the result file.
Read_PHREEQE_result()	Reads the result produced by PHREEQE.
Saturation_index()	Calculates the saturation indices.
Test_SI()	Compares the calculated SI's with numbers obtained by Search_SI_utdata().
Change_number_of_components()	Writes the number of master species, involved in the net stoichiometric reaction added to the solution, into the indata file for PHREEQE.
Element_numbers()	Compares the components of the liquid solution with the elements in the library file to match names and numbers of the elements.

Make_solution()	Makes the SOLUTION part of the indata file for PHREEQE.
Make_head()	Writes the standard option line into the PHREEQE indata file.
Make_reaction_head()	Writes the option line for the case that a net stoichiometric reaction is added to the solution.
Read_Quantities()	Reads the quantity of the components in the solid solution.
Read_element_data()	Reads the elements from the library file.
Read_Qty_for_SI_calc()	Reads the quantity of a component needed for the calculation of the saturation index.
Remember_old_solution()	Stores the composition of the solution.
Write_New_Qty()	Writes new quantities to the file QUANTITY.DAT.
Write_Old_Qty()	Writes old quantities to the file QUANTITY.DAT.
Write_Qty_for_SI_calc()	Writes quantities to the file QUANTITY.SI to be used in calculation of the saturation indices.

MAIN VARIABLES

Mineral data structure

The mineral variables declared are:

Min[ ]	Components in the solid solution
Old_Min[ ]	Components in the old solid solution
Look_Min[ ]	Components in the LOOK MIN list where the saturation state of the solution is given.

The arrays are declared as structures of the following content:

Numbers	
Name[ ]	Mineral name
Quantity	Mineral quantities in different stages of the iteration
Qty_New	process.
Qty_Old	
Dissolved	The quantity dissolved by PHREEQE, the value can be positive (dissolved) or negative (precipitation).
X	Mole fraction of the mineral.
SI	Calculated SI from data obtained in the result file.
SI_Utdata	SI found in the result file.
SI_Calc	Used in calculation to determine the quantity to be added in the REACTION group.

Solution data structure

The solution variable Current_Solution is declared as a structure with the following content:

Name[ ]	
ElementName[ ][ ]	
Concentration[ ]	
pH	
pE	
Temperature	
Components	Number of components in the solution.
ElementNo[ ]	Used when making the solution.
Iterations	
Element[ ][ ]	Maximum 4 characters in element names.
Number_Of_Minerals	The number of components in the solid solution.
Number_Of_Look_Mins	
Reaction_Test	
Tot_Reaction_strings	The number of constituents in a net stoichiometric reaction.
Tot_Qty_New	
Tot_Quantity	

TROUBLE SHOOTING

Charge balance

It is important that the start solution is charge balanced since there is no routine in the program to take care of charge balance problems.

If there is any doubt concerning the electroneutrality of the solution an external run with PHREEQE can be done. In the indata file for PHREEQE the option number two and three are then assigned the values 2 and 0, respectively. This will cause the initial charge imbalance to be adjusted by adding one of the elements to the solution and that no reaction is modeled. A correct PHREEQE indata file for SOLISOL is obtained by modifying the concentration of the element used in the adjustment of the solution and changing option number two from 2 to 0. This new value on option number two makes PHREEQE maintain the charge balance of the solution.

FUTURE IMPROVEMENTS

Activity coefficient

As already mentioned the activity coefficients of components in solid solutions are often unknown. Hence, the SOLISOL code assumes them to be unity.
The program will be modified to include activity coefficients if they are known. Data of the coefficients will be stored in files like *.MIN and *.REA and be placed in the subdirectory MINERAL. Thus, each component will have its own file, *.ACT, where data on how to handle the activity coefficients will be stored.

Number of components

The present version of the SOLISOL code can only handle solid solutions consisting of up to four components. There are plans to modify the code to handle an arbitrary number of components.

Convergence parameters

In the code SOL2 convergence parameters are used in the functions main, Give_minerals, Saturation_index and Test_SI. In the present version of SOL2 they are fixed. In the future there are plans to modify the code to allow the parameters to be set to a desired value.

REFERENCES

BÖR 92 Börjesson, K.S., Emrén, A.T., "SOLISOL, a program using PHREEQE to solve solid solutions problems", submitted to COMPUTERS & GEOSCIENCES, 1992

CRO 87 Cross, J.E., Ewart, F.T., Tweed, C.J. "Thermochemical Modelling with Application to Nuclear Waste Processing and Disposal", AERE R 12324, UKAEA, Harwell (1987), Modified: Börjesson, S; Dept. of Nuclear Chemistry, CTH, 910310

EMR 91 Emrén, A., "CRACKER - A Code Modelling Chemical Changes of Ground Water Passing a System of Cracks", Radiochim. Acta. 52/53, 473 (1991)

MUL 85 Muller, A.B., Parkhurst, D.L., Tasker, P.W. "The Use of the PHREEQE Code in Modelling Environmental Geochemical Problems Encountered in Performance Assessment Modelling", Proc. The Symposium on Groundwater Flow and Transport Modeling for Performance Assessment of Deep Geologic Disposal of Radioactive Waste: A Critical Evaluation of the State of the Art, Albuquerque (1985)

PAR 84 Parkhurst, D.L., Thorstenson,D.C., Plummer, L.N. "PHREEQE, A Computer Program for Geochemical Calculations", USGS/WRI p 80-96 (1980).
Version current, August 3, 1984. Changes: a/ Yamaguchi, Y., NEA DB December 17, 1984 and b/ Fridemo, L., Dept. of Nuclear Chemistry, CTH, July 5, 1985

Appendix A

SOLISOL.C Program list

SOLISOL.C PROGRAM LIST

```
/*
*          SOLISOL. version 1992 - 09 - 04
*                    by
*        Susanne Borjesson and Allan T Emrén
*        Dept Nuclear Chemistry
*        Chalmers University of Technology
*        S-41296 Goteborg
*        SWEDEN
*/


#include "allan.h"

#include <process.h>


void main()
{
 system("phreeqe<names");
 system ("sol1");
 system("del quantity.old");
 do
 {
 system("phreeqe<names");
 }
while( spawnl(P_WAIT,"sol2.exe","sol2.exe",NULL )!= 4);
printf("**CONVERGED**\n");
}
```

Appendix B

SOL1.C Program list

SOL1.C PROGRAM LIST

```
/*
*          SOL1, version 1992 - 09 - 04
*                    by
*          Allan T Emrén and Susanne Borjesson
*          Dept Nuclear Chemistry
*          Chalmers University of Technology
*          S-41296 Goteborg
*          SWEDEN
*/

#include "allan.h"

#define FAILURE 0
#define SUCCESS 1
#define MAX_NUMBER_OF_MINERALS 50

struct Mineral {
                char Name[10];
                float Quantity,
                X,
                SI;
               }
               Min[5];  /* Components in the solid solution */

struct Solution {
                 char Name[130],
                 ElementName[30][5];
                 float Concentration[30],
                 pH,
                 pE,
                 Temperature;
                 short Components,    /* Number of components in the solution */
                 ElementNo[30],
                 Iterations;
                }
                Current_Solution;
```

```
char Element[30][5];          /* Max 4 characters in element names */

short Number_Of_Minerals;
float Tot_Quantity=0;

FILE *In,
     *Out;

void Give_minerals(),
     Make_solution(),
     Make_head(),
     Make_PHREEQE_file(),
     Element_numbers(),
     Saturation_index(),
     Read_element_data(),
     Read_Quantities(),
     Trim();

double atoDf();


void main()
{
 if( Read_PHREEQE_result() == FAILURE )
 {
  printf("PHREEQE failed!!! Please check manually!\n");
  exit(0);
 }
 Read_element_data();
 Element_numbers();
 Saturation_index();
 Make_PHREEQE_file();
}
```

```
Read_PHREEQE_result()
{
short i=1,
      result,
      maxlength=128;
char string[130],
     part[20],
     *found;

if( (In = fopen("OUTDATA", "r")) == NULL) exit(0);

if ( ( result= Search_text("TOTAL MOLALITIES") ) == SUCCESS )
printf("Found TOTAL MOLALITIES\n");

     /* Go to first concentration */
{
fgets(string,maxlength,In); /* underline */
fgets(string,maxlength,In); /* empty     */
fgets(string,maxlength,In); /* headline  */
fgets(string,maxlength,In); /* empty     */

i=1;                /* Reset counter */

do                  /* Read molalities */
{
 fgets(string,maxlength,In);
 found = string + 16;                          /* Position of element name */
 stccpy( Current_Solution.ElementName[i],
          found,
          3 );
 found = string + 29;                          /* Position of element concentration */
 stccpy( part,
         found,
         16 );
 Current_Solution.Concentration[i] = atoDf(part);
 i++;
}
while( (string[16] != 32) && (strlen(string) > 15) );
```

```
Current_Solution.Components=i-2;
Search_text("DESCRIPTION OF SOLUTION");

  /* Go to pH-line */
fgets(string,maxlength,In);    /* empty */
fgets(string,maxlength,In);
found = string + 44;            /* Position of pH value */

stccpy( part,
        found,
        9 );

Current_Solution.pH = atoDf(part);
fgets(string,maxlength,In);
found = string + 44;            /* Position of pE value */

stccpy( part,
        found,
        9 );

Current_Solution.pE = atoDf(part);

  /* Go to temperature */
fgets(string,maxlength,In); /* eH              */
fgets(string,maxlength,In); /* activity        */
fgets(string,maxlength,In); /* ionic strength  */
fgets(string,maxlength,In); /* temperature     */
found = string + 44;      /* Position of temperature */
stccpy( part,
        found,
        9 );
Current_Solution.Temperature = atof(part);

  /* Go to # of iterations */
fgets(string,maxlength,In); /* electrical balance */
fgets(string,maxlength,In); /* thor            */
fgets(string,maxlength,In); /* total alcalinity */
```

```
 fgets(string,maxlength,In);
 found = string + 44;      /* Position of iteration value */

 stccpy( part,
         found,
         9 );

 Current_Solution.Iterations = atoi(part);
}
fclose(In);
return ( result );
}


void Read_element_data()
{
 short elementno,
       maxlength=128;

 char string[130],
      part[20],
      *found;

 if( (In = fopen("LIBRARY.DAT", "r")) == NULL) exit(0);
 if( Search_text("ELEMENTS") == SUCCESS )
 {
  printf("Found ELEMENTS\n");
  fgets(string,maxlength,In);
  while( (string[11] != 32) && (strlen(string) > 15) )
  {
   found = string + 10;      /* Position of element number */
   stccpy( part,
           found,
           3 );
   elementno = atoi(part);
   stccpy( Element[ elementno ],     /* Read element name */
           string,
           4 );
```

```
  fgets(string,maxlength,In);
 }
}
fclose(In);
}


void Element_numbers()
{
 short i,
       j,
       test;

 for( j = 1; j <= Current_Solution.Components; j++ )
 {
  for( i = 4; i < 30; i++ )
  {
   test = strncmp( Element[i],
                   Current_Solution.ElementName[j],
                   2 );


    if( test == 0 )
   {
    Current_Solution.ElementNo[j] = i;
   }
  }
 }
}

void Make_PHREEQE_file()
{
 if( (Out = fopen("INFILE.DAT", "w")) == NULL) exit(0);
 Make_head();
 Make_solution();
 Give_minerals();
 fprintf(Out,"END    \n");
 fclose(Out);
}
```

```
Search_text( desired_text )
    char *desired_text;
{
 short maxlength=128,
       result;
 char string[130];

 Next_record:
 fgets(string,maxlength,In);
 if( strstr( string, "TERMINATED" ) != NULL)
 {
  result = FAILURE;
  goto Stop_searching;
 }
 if( strstr( string, desired_text ) == NULL)
 goto Next_record;
 result = SUCCESS;
 Stop_searching:
 return( result );
}


void Read_Quantities()
{
 short maxlength=128,
       n;
 char string[130],
      part[20],
      *found;

 if( ( In= fopen("QUANTITY.DAT", "r")) == NULL) exit(0);
 fgets(string, maxlength, In);      /* read numbers */
 Number_Of_Minerals=atoi(string);

 for (n=1; n<= Number_Of_Minerals; n++)
 {
  fgets(Min[n].Name, 16, In);      /* read mineralname */
  Trim(Min[n].Name);
```

```
 fgets(string, maxlength, In);       /* read quantity  */
 Min[n].Quantity=atof(string);
 }
 fclose(In);
}


void Make_head()
{
 char line[130];     /* Next line from header file */
 short maxlength=128; /* Max line length           */

 char *buf; /* for diagnostic purpose */

 if( (In = fopen("HEADER.DAT", "r")) == NULL)
 {
  printf("Could not open HEADER.DAT\n");
  exit(0);
 }
 buf=line;
 while(1)
 {
  buf=fgets(line, maxlength, In);
  if( buf == NULL) break;        /* End of file */
  if( (buf = strchr(line, 10) ) != NULL)
  {
   *buf = 0;     /* get rid of LINE FEED character */
  }
  if( (buf = strchr(line, 13) ) != NULL)
  {
   *buf = 0;     /* get rid of RETURN character */
  }
  fprintf(Out,"%s\n",line);
 }
 fclose(In);
}
```

```
void Make_solution()
{
 short component,
       on_line=0;

 fprintf(Out,"SOLUTION 1\n");
 fprintf(Out,"Standard\n");
 fprintf(Out,"%2d  0 0   %9.3f %9.3f %9.3f  1.000\n",
       /* antal    pH       pE      T      dens  */
         Current_Solution.Components,
         Current_Solution.pH,
         Current_Solution.pE,
         Current_Solution.Temperature);

 for( component=1;
      component <= Current_Solution.Components;
      component++ )
 {
  fprintf(Out,"  %2d%11.4E", Current_Solution.ElementNo[component],
                    Current_Solution.Concentration[component]);    /*  i4,11.3D */
  on_line++;
  if( on_line == 5 ) /* max 5 concentrations pr line */
  {
   if( component <= Current_Solution.Components )
   {
    fprintf(Out,"\n");
    on_line = 0;
   }
  }
 }
 if(( Current_Solution.Components  % 5 ) != 0)
 {
  fprintf(Out,"\n");
 }
}
```

```
void Give_minerals()
{
 char line[130],     /* Next line from header file */
      data_file[25],
      directory[] = "MINERAL\\",
       mineral_ext[] = ".MIN",
       *buf;

 short maxlength=128,       /* Max line length  */
       next_mineral;

 fprintf(Out,"MINERALS\n");
 for (next_mineral = 1; next_mineral<= Number_Of_Minerals; next_mineral++)
 {
  strcpy( data_file, directory );
  strcat( data_file, Min[next_mineral].Name);
  strcat( data_file, mineral_ext );

  if( (In = fopen( data_file, "r")) == NULL)
  {
   printf("Number of next mineral: %d\n", next_mineral );
   printf(" Cannot open file\n");
   exit(0);
  }
  fgets(line, maxlength, In);
  buf=line+60;                    /* Position of Sl */
  gcvt(Min[next_mineral].Sl,10,buf);
  fprintf(Out,"%s\n",line);
  while(1)
  {
   buf=fgets(line, maxlength, In);
   if( buf == NULL) break;
   if( (buf = strchr(line, 10) ) != NULL)
   {
    *buf = 0;     /* get rid of LINE FEED character */
   }
   if( (buf = strchr(line, 13) ) != NULL)
   {
```

```
  *buf = 0;     /* get rid of RETURN character */
  }
  fprintf(Out,"%s\n",line);
 }
 fclose(In);
}
fprintf(Out,"\n");
}


double atoDf(string) /* Change FORTRAN D-form into e-form */
char string[];
{
 char c='D',
      *position;

 position=strchr(string,c);
 *position='e';
 return(atof(string));
}


void Saturation_index()
{
 short n;

 Read_Quantities();
 for(n=1; n<= Number_Of_Minerals; n++)
 {
  Tot_Quantity+=fabs(Min[n].Quantity);
 }
 for(n=1; n<= Number_Of_Minerals; n++)
 {
  Min[n].X= (fabs(Min[n].Quantity) / Tot_Quantity);
  Min[n].SI= log10(Min[n].X);
 }
}
```

Appendix C

SOL2.C Program list

SOL2.C PROGRAM LIST

```
/*
 *        SOL2 version 1992 - 09 - 04
 *                by
 *        Susanne Börjesson and Allan T Emrén
 *        Dept Nuclear Chemistry
 *        Chalmers University of Technology
 *        S-41296 Goteborg
 *        SWEDEN
 */

#include "allan.h"

#define FAILURE 0
#define SUCCESS 1
#define TOO_MUCH 0
#define OK 1
#define CONVERGED 4
#define NOT_READY -1
#define MAX_NUMBER_OF_MINERALS 50
#define MAKE_REACTION -2
#define NO_REACTION 3
#define FOUND_REACTION 2
#define NOT_FOUND_REACTION -4

struct Mineral {
                short Numbers;
                char Name[15];
                float Qty_New,
                      Qty_Old,
                      Q_Old,
                      Dissolved,
                      X,
                      SI_Utdata,
                      SI_Calc,
                      SI;
                }

        Min[5],         /* Components in the solid solution */
        Old_Min[5],  /* Components in the old solid solution */
        Look_Min[41];

struct Solution {
                char Name[130],
                ElementName[30][5];
                float Concentration[30],
                pH,
                pE,
                Temperature;
                short Components,    /* Number of components */
                                     /* in the solution      */
                ElementNo[30],
                Iterations;
                }

              Current_Solution;

char  Element[30][5]; /* Max 4 characters in element names */
```

```
short Number_Of_Minerals,
      Number_Of_Look_Mins,
      Reaction_Test,
      Tot_Reaction_strings=0;

float Tot_Qty_New=1,
      Tot_Quantity=0;

FILE *In,
     *Out;

void Give_minerals(),
     Make_solution(),
     Remember_old_solution(),
     Make_head(),
     Make_PHREEQE_file(),
     Element_numbers(),
     Change_number_of_components(),
     Read_element_data(),
     Read_Quantities(),
     Read_Old_Quantities(),
     Read_Qty_for_SI_calc(),
     Read_reaction_line(),
     Write_Old_Qty(),
     Write_New_Qty(),
     Write_Qty_for_SI_calc(),
     Search_SI_utdata(),
     Make_reaction_head(),
     Add_reaction(),
     Copy_TMP_to_DAT(),
     Trim_blank(),
     Trim();
double atoDf();
```

```
Read_PHREEQE_result()
{
 short n=1,
       i=1,
       result,
       maxlength=128;

 char string[130],
      part[20],
      reaction_test[15],
     *found;

 if( (In = fopen("OUTDATA", "r")) == NULL) exit(0);

 for(n=1; n<= 30; n++)
 {
  fgets(string maxlength,In);
  if( strstr(string,"REACTION" ) != NULL )
  {
   Reaction_Test=FOUND_REACTION;
   n=35;
  }
 }
 if(Reaction_Test == FOUND_REACTION )
 {
  printf("Search_STEP_NUMBER:\n");
  if(( result= Search_text("STEP NUMBER") ) == SUCCESS )
  {
   printf("Search_LOOK_MIN:\n");
   Search_text("LOOK MIN");
   Search_SI_utdata();
  }
 }
 else
 {
  printf("Search_PHASE_BOUNDARIES:\n");
  if( ( result=Search_text("PHASE BOUNDARIES") ) == SUCCESS )
  {
   fgets(string,maxlength,In); /* empty    */
   fgets(string,maxlength,In); /* headline */
   fgets(string,maxlength,In); /* empty    */

   do                    /* Read dissolved /precipitated quantities*/
   {
    fgets(string,maxlength,In);
    found = string + 8;      /* Position of mineral name */

    stccpy( Min[i].Name,
            found,
            9 );
    found = string + 18;     /* Position of dissolved/precipitated*/

    stccpy( part,
            found,
            16 );
    Min[i].Dissolved = atoDf(part);
    i++;
   }
   while( (string[8] != 32) && (strlen(string) > 7) );
```

```
  printf("Search_LOOK_MIN:\n");
  Search_text("LOOK MIN");
  Search_SI_utdata();
 }
}
printf("Search_TOTAL_MOLALITIES:\n");
Search_text("TOTAL MOLALITIES");

  /* Go to first concentration */
fgets(string,maxlength,In); /* underline */
fgets(string,maxlength,In); /* empty     */
fgets(string,maxlength,In); /* headline  */
fgets(string,maxlength,In); /* empty     */

i=1;                    /* Reset counter */

do                      /* Read molalities */
{
 fgets(string,maxlength,In);
 found = string + 16;       /* Position of element name */

 strcpy( Current_Solution.ElementName[i],
          found,
          3 );
 found = string + 29;       /* Position of element concentration */

 strcpy( part,
          found,
          16 );

 Current_Solution.Concentration[i] = atoDf(part);
 i++;
}
while( (string[16] != 32) && (strlen(string) > 15) );

Current_Solution.Components=i-2;

Search_text("DESCRIPTION OF SOLUTION");
printf("DESCRIPTION OF SOLUTION\n");

  /* Go to pH-line */
fgets(string,maxlength,In); /* empty */
fgets(string,maxlength,In);
found = string + 44;       /* Position of pH value */

strcpy( part,
          found,
          9 );

Current_Solution.pH = atoDf(part);
fgets(string,maxlength,In);
found = string + 44;       /* Position of pE value */

strcpy( part,
          found,
          9 );

Current_Solution.pE = atoDf(part);
```

```
   /* Go to temperature */
 fgets(string,maxlength,In); /* eH              */
 fgets(string,maxlength,In); /* activity        */
 fgets(string,maxlength,In); /* ionic strength  */
 fgets(string,maxlength,In); /* temperature     */
 found = string + 44;      /* Position of temperature */

 stccpy( part,
         found,
         9 );

 Current_Solution.Temperature = atof(part);

   /* Go to # of iterations */
 fgets(string,maxlength,In); /* electrical balance */
 fgets(string,maxlength,In); /* thor              */
 fgets(string,maxlength,In); /* total alcalinity  */

 fgets(string,maxlength,In);
 found = string + 44;      /* Position of iteration value */
 stccpy( part,
         found,
         9 );

 Current_Solution.Iterations = atoi(part);

 fclose(In);
 return( result );
}


void Read_element_data()
{
 short elementno,
       maxlength=128;

 char string[130],
      part[20],
      *found;

 if( (In = fopen("LIBRARY.DAT", "r")) == NULL) exit(0);

 if( Search_text("ELEMENTS") == SUCCESS )
 {
  fgets(string,maxlength,In);
  while( (string[11] != 32) && (strlen(string) > 15) )
  {
   found = string + 10;      /* Position of element number */

   stccpy( part,
           found,
           3 );
   elementno = atoi(part);

   stccpy( Element[ elementno ], /* Read element name */
           string,
           4 );
   fgets(string,maxlength,In);
  }
 }
```

```
fclose(In);
}


void Element_numbers()
{
 short i,
       j,
       test;

 for( j = 1; j <= Current_Solution.Components; j++ )
 {
  for( i = 4; i < 30; i++ )
  {
   test = strncmp( Element[i],
                   Current_Solution.ElementName[j],
                    2 );

    if( test == 0 )
   {
    Current_Solution.ElementNo[j] = i;
   }
  }
 }
}


void Make_PHREEQE_file(mineral_test)
          short mineral_test;
{
 short n;
 switch( mineral_test )
 {
  case OK:
    printf("case OK!\n");
    Make_head();
    Make_solution();
    Give_minerals();
    fprintf(Out,"END     \n");
    fclose(Out);
    break;
     case TOO_MUCH:
       printf("case TOO_MUCH!\n");
       Remember_old_solution();
       Give_minerals();
       fprintf(Out,"END     \n");
       fclose(Out);
       Copy_TMP_to_DAT();
       break;
          case MAKE_REACTION:
            if(Reaction_Test== FOUND_REACTION)
            {
              printf("REACTION iteration is going on!\n");
              Remember_old_solution();
              Add_reaction();
              break;
            }
```

```
            else
            {
             printf("Making REACTION file \n");
             Write_Qty_for_SI_calc();
             Make_reaction_head();
             Make_solution();
             Add_reaction();
             break;
            }
                case NO_REACTION:
                   for (n=1; n<= Number_Of_Minerals; n++)
                   {
                    if ( Min[n].Qty_Old < 0)
                    {
                     Min[n].Qty_New = fabs( Min[n].Qty_New);
                    }
                   }
                   Write_New_Qty();
                   system("copy quantity.dat quantity.old");
                   Make_head();
                   Make_solution();
                   Give_minerals();
                   fprintf(Out,"END     \n");
                   fclose(Out);
                   break;
 }
}


Search_text( desired_text )
     char *desired_text;
{
 short maxlength=128,
       result;

 char string[130];

 Next_record:
 fgets(string,maxlength,In);
 if( strstr( string, "TERMINATED" ) != NULL)
 {
  result = FAILURE;
  goto Stop_searching;
 }
 if( strstr( string, desired_text ) == NULL)   goto Next_record;
 result = SUCCESS;

 Stop_searching:
 return( result );
}
```

```
void Read_Quantities()
{
 short maxlength=128,
       n;
 char string[130],
      part[20],
     *found;
 if( ( In= fopen("QUANTITY.DAT", "r")) == NULL) exit(0);

 fgets(string, maxlength, In);      /* read numbers */
 Number_Of_Minerals=atoi(string);

 for (n=1; n<= Number_Of_Minerals; n++)
 {
  fgets(Min[n].Name,14, In);      /* read mineralname */
  Trim(Min[n].Name);
  fgets(string, maxlength, In);       /* read quantity  */
  Min[n].Qty_Old=atof(string);
  Min[n].Qty_New=Min[n].Qty_Old;
 }
 fclose(In);
}


void Read_Old_Quantities()
{
 short maxlength=128,
       n;
 char string[130],
      part[20],
      *found;


 fgets(string, maxlength, In);      /* read number of minerals  */
 Number_Of_Minerals=atoi(string);


 for(n=1; n<= Number_Of_Minerals; n++)
 {
  fgets(Old_Min[n].Name,14, In);      /* read mineralname */
  Trim(Old_Min[n].Name);
  fgets(string, maxlength, In);       /* read quantity  */
  Old_Min[n].Qty_Old=atof(string);
  Old_Min[n].Qty_New=Old_Min[n].Qty_Old;
 }
 fclose(In);
}
```

```
void Read_Qty_for_SI_calc()
{
 short maxlength=128,
       n;
 char string[130],
      part[20],
     *found;

 if( ( In= fopen("QUANTITY.SI", "r")) == NULL) exit(0);

 fgets(string, maxlength, In);      /* read numbers */
 Number_Of_Minerals=atoi(string);

 for (n=1; n<= Number_Of_Minerals; n++)
 {
  fgets(Min[n].Name,14, In);      /* read mineralname */
  Trim(Min[n].Name);
  fgets(string, maxlength, In);       /* read quantity  */
  Min[n].Q_Old=atof(string);
 }
 fclose(In);
}


void Write_Old_Qty()
{
 short n;
 if( ( Out=fopen("QUANTITY.DAT", "w")) == NULL) exit(0);

 fprintf(Out,"%d\n",Number_Of_Minerals);
 for( n=1; n<= Number_Of_Minerals; n++)
 {
  fprintf(Out,"%s\n",Min[n].Name);
  fprintf(Out,"%e\n",Min[n].Qty_Old);
 }
 fclose(Out);
}


void Write_New_Qty()
{
 short n;
 if( ( Out=fopen("QUANTITY.DAT", "w")) == NULL) exit(0);

 fprintf(Out,"%d\n",Number_Of_Minerals);

 for( n=1; n<= Number_Of_Minerals; n++)
 {
  fprintf(Out,"%s\n",Min[n].Name);
  fprintf(Out,"%e\n",Min[n].Qty_New);
 }
 fclose(Out);
}
```

```
void Write_Qty_for_SI_calc()
{
 short n;
 if( ( Out=fopen("QUANTITY.SI", "w")) == NULL) exit(0);
 fprintf(Out,"%d\n",Number_Of_Minerals);
 for( n=1; n<= Number_Of_Minerals; n++)
 {
  fprintf(Out,"%s\n",Min[n].Name);
  fprintf(Out,"%e\n",Min[n].Qty_New);
 }
 fclose(Out);
}


void Make_head()
{
 char line[130];     /* Next line from header file */
 short maxlength=128; /* Max line length          */

 char *buf;   /* for diagnostic purpose */

 if( (Out = fopen("INFILE.DAT", "w")) == NULL) exit(0);


 if( (In = fopen("HEADER.DAT", "r")) == NULL)
 {
  printf("Could not open HEADER.DAT\n");
  exit(0);
 }
 buf=line;
 while(1)
 {
  buf=fgets(line, maxlength, In);
  if( buf == NULL) break;       /* End of file */

  Trim(line);
  fprintf(Out,"%s\n",line);
 }
 fclose(In);
}


void Make_solution()
{
 short component,
       on_line=0;

 fprintf(Out,"SOLUTION 1\n");
 fprintf(Out,"Standard\n");
 fprintf(Out,"%2d  0 0   %9.3f %9.3f %9.3f  1.000\n",
       /* antal    pH       pE      T      dens  */
          Current_Solution.Components,
          Current_Solution.pH,
          Current_Solution.pE,
          Current_Solution.Temperature);
```

```
for( component=1;
   component <= Current_Solution.Components;
   component++ )
{
        /*  i4,11.3D */
 fprintf(Out," %2d%11.4E", Current_Solution.ElementNo[component],
                  Current_Solution.Concentration[component] );

 on_line++;
 if( on_line == 5 )     /* max 5 concentrations pr line */
 {
  if( component <= Current_Solution.Components )
  {
   fprintf(Out,"\n");
   on_line = 0;
  }
 }
}
if(on_line != 0)
{
 fprintf(Out,"\n");
}
}


void Give_minerals()
{
 char line[130],     /* Next line from header file */
      data_file[25],
      directory[] = "MINERAL\\",
      mineral_ext[] = ".MIN",
      *buf;

 short maxlength=128,      /* Max line length  */
       next_mineral;

 fprintf(Out,"MINERALS\n");

 for (next_mineral = 1; next_mineral<= Number_Of_Minerals; next_mineral++)
 {
  if (  Min[next_mineral].Qty_New > 0 )
  {
   strcpy( data_file, directory );
   strcat( data_file, Min[next_mineral].Name);
   strcat( data_file, mineral_ext );

   if( (In = fopen( data_file, "r")) == NULL)
   {
    printf("Number of next mineral: %d\n", next_mineral );
    printf(" Cannot open file\n");
    exit(0);
   }
   fgets(line, maxlength, In);
   buf=line+60;              /* pointing at the position of SI */
   Min[next_mineral].SI =
      ( fabs(Min[next_mineral].SI) < 0.0001 )? 0
                                  : Min[next_mineral].SI;
   gcvt(Min[next_mineral].SI,10,buf);
   fprintf(Out,"%s\n",line);
```

```
  while(1)
  {
   buf=fgets(line, maxlength, In);
   if( buf == NULL) break;

   if( (buf = strchr(line, 10) ) != NULL)
   {
    *buf = 0;     /* get rid of LINE FEED character */
   }

   if( (buf = strchr(line, 13) ) != NULL)
   {
    *buf = 0;     /* get rid of RETURN character */
   }
   fprintf(Out,"%s\n",line);
  }
  fclose(In);
 }
}
fprintf(Out,"\n");
}


double atoDf(string) /* Change FORTRAN D-form into e-form */
char string[];
{
 char c='D',
      *position;

 position=strchr(string,c);
 *position='e';
 return(atof(string));
}


void Trim_blank(string)
char string[];
{
 short t,
       l;

 l=strlen(string);
 for( t=l-1; t>0; t--)
 {
  if( string[t] != 32) break;
  string[t]=0;
 }
}
```

```
Saturation_index()
{
 short mineral_test,
       abs_test,
       n;
 Read_Quantities();

 mineral_test=OK;
 abs_test=CONVERGED;

 for (n=1; n<= Number_Of_Minerals; n++)
 {
  if ( Min[n].Qty_Old > 0)
  {
   if ( Min[n].Dissolved*1000 > Min[n].Qty_Old * 0.80 )
   {
    Min[n].Qty_Old = -fabs( Min[n].Qty_Old);
    Min[n].Qty_New =  Min[n].Qty_Old;
    mineral_test=TOO_MUCH;
   }
   else
   {
    if ( fabs ( Min[n].Dissolved*1000 / Min[n].Qty_Old ) > 0.05 )
    {
     abs_test=NOT_READY;
    }
    Min[n].Qty_New = Min[n].Qty_Old - Min[n].Dissolved*1000;
    /* New quantities, mole to mmole*/
   }
  }
 }

 if( mineral_test == TOO_MUCH )
 {
  Write_Old_Qty();

  for(n=1; n<= Number_Of_Minerals; n++)
  {
   Tot_Quantity+=fabs(Min[n].Qty_Old);
  }
  for(n=1; n<= Number_Of_Minerals; n++)
  {
   Min[n].X= (fabs( Min[n].Qty_Old ) / Tot_Quantity);
   Min[n].SI= log10(Min[n].X);
  }
 }
 else
 {
  Write_New_Qty();
  for(n=1; n<= Number_Of_Minerals; n++)
  {
   Tot_Quantity+= fabs ( Min[n].Qty_New );
  }
  for(n=1; n<= Number_Of_Minerals; n++)
  {
   Min[n].X= ( fabs( Min[n].Qty_New ) / Tot_Quantity);
   Min[n].SI= log10(Min[n].X);
  }
 }
```

```
for(n=1; n<= Number_Of_Minerals; n++)
{
 Tot_Qty_New*=Min[n].Qty_New;
 if(Tot_Qty_New <0) goto stop1;
}
stop1:
if ( abs_test == CONVERGED && mineral_test == OK) return( CONVERGED);
if ( mineral_test == TOO_MUCH) return( TOO_MUCH);
if ( abs_test == NOT_READY && mineral_test == OK) return( OK);
}


void Remember_old_solution()
{
char line[130];          /* Next line from header file */
short maxlength=128; /* Max line length          */

char *buf; /* for diagnostic purpose */

if( (Out = fopen("INFILE.TMP", "w")) == NULL) exit(0);

if( (In = fopen("INFILE.DAT", "r")) == NULL)
{
 printf("Could not open INFILE.DAT\n");
 exit(0);
}
buf=line;
while(1)
{
 buf=fgets(line, maxlength, In);
 if( buf == NULL) break;        /* End of file */

 if( strstr( line, "REACTION" ) != NULL)  goto end_of_solution;
 if( strstr( line, "MINERALS" ) != NULL)  goto end_of_solution;

 Trim(line);

 fprintf(Out,"%s\n",line);
}
end_of_solution:
fclose(In);
}


void Search_SI_utdata()
{
short i=1,
      maxlength=128;
char string[130],
     *found,
      part[15];

fgets(string,maxlength,In); /* empty    */
fgets(string,maxlength,In); /* headline */
fgets(string,maxlength,In); /* empty    */

do
{
 fgets(string,maxlength,In);
 found= string+ 14;          /* position of mineralname*/
```

```
 stccpy( Look_Min[i].Name,
         found,
         8);

 Trim_blank(Look_Min[i].Name);
 found=string+52;          /* position of SI */

 stccpy( part,
         found,
         9);

 Look_Min[i].SI_Utdata=atoDf(part);
 i++;
 }
 while( (string[14] !=32) && (strlen(string) > 13) );
 Number_Of_Look_Mins=i-2;
}


Test_SI()
{
 short mineral_test=NO_REACTION,
       i,
       n,
       test;

 for ( n=1; n<= Number_Of_Minerals; n++ )
 {
  for ( i=1; i<= Number_Of_Look_Mins; i++ )
  {
   test= strncmp( Look_Min[i].Name,
                  Min[n].Name,
                  9);
   if (test == 0 )
   {
    if ( fabs ( Look_Min[i].SI_Utdata - Min[n].SI ) > 0.00055 )
    {
     if( Min[n].Qty_New < 0 )
     {
      mineral_test= MAKE_REACTION;
      Min[n].SI_Calc=Look_Min[i].SI_Utdata ;
     }
    }
   }
  }
 }
 if (mineral_test==NO_REACTION) return(CONVERGED);
 if (mineral_test==MAKE_REACTION) return(MAKE_REACTION);
}
```

```
void Make_reaction_head()
{
 char line[130];     /* Next line from header file */
 short maxlength=128; /* Max line length          */

 char *buf; /* for diagnostic purpose */

 if( (Out = fopen("INFILE.TMP", "w")) == NULL) exit(0);

 if( (In = fopen("REAHEAD.DAT", "r")) == NULL)
 {
  printf("Could not open REAHEAD.DAT\n");
  exit(0);
 }
 buf=line;
 while(1)
 {
  buf=fgets(line, maxlength, In);
  if( buf == NULL) break;        /* End of file */
  Trim(line);
  fprintf(Out,"%s\n",line);
 }
 fclose(In);
}


void Add_reaction()
{
 short n;

 Read_Qty_for_SI_calc();
 Read_Quantities();
 Tot_Quantity=0;
 for( n=1; n<= Number_Of_Minerals; n++ )
 {
  if( Min[n].Qty_New > 0 )
  {
   Tot_Quantity+=Min[n].Qty_New;  /*The quantity of the labeled component neglected*/
  }
 }
 for( n=1; n<= Number_Of_Minerals; n++ )
 {
  if( Min[n].Qty_New < 0 )
  {
   Min[n].Qty_New=fabs(Min[n].Q_Old)-(Tot_Quantity*pow(10,Min[n].SI_Calc));
  }
 }
 Read_reaction_line();

 fprintf(Out,"\n");
 for( n=1; n<= Number_Of_Minerals; n++ )
 {
  if( Min[n].Qty_Old < 0 )
  {
   Min[n].Qty_New=-(fabs(Min[n].Q_Old)- Min[n].Qty_New);
  }
  else
  {
   Min[n].Qty_New=Min[n].Q_Old;
  }
```

```
}
fprintf( Out,"STEPS   \n");
fprintf( Out,"1.0     \n");
fprintf(Out,"END    \n");
fclose(Out);

Write_New_Qty();
Change_number_of_components();
}


void Read_reaction_line()
{
char line[130],
     data_file[25],
     directory[] = "MINERAL\\",
     part[20],
     mineral_ext[] = ".REA",
     *buf;

float opv;

short maxlength=128, /* Max line length  */
      on_line=0,
      n;

fprintf(Out,"REACTION\n");

for (n = 1; n<= Number_Of_Minerals; n++)
{
 if ( Min[n].Qty_Old < 0 )
 {
  strcpy( data_file, directory );
  strcat( data_file, Min[n].Name);
  strcat( data_file, mineral_ext );
  if( (In = fopen( data_file, "r")) == NULL)
  {
   printf("Number of next mineral: %d\n", n );
   printf(" Cannot open file\n");
   exit(0);
  }
  while( fgets(line,maxlength,In)  != NULL )
  {
   Trim(line);
   buf=line+4;      /*position of moles added in the REACTION */
   sprintf(buf,"%8.3e",Min[n].Qty_New/1000);   /* mmoles to moles  */
   *(buf+7)=*(buf+8);
   *(buf+8)=' ';
   fprintf(Out,"%s",line);
   buf=line+12;
   strcpy( part,
           buf,
           7 );
   opv= atoDf(part);
   fprintf(Out,"%7.3f",opv);
   on_line++;
```

```
   if( on_line == 3 )
   {
    fprintf(Out,"\n");
    on_line = 0;
   }
   Tot_Reaction_strings++;
  }
  fclose(In);
 }
}
}


void Change_number_of_components()
{
 char line[130];         /* Next line from header file */
 short maxlength=128; /* Max line length          */

 char *buf; /* for diagnostic purpose */

 if( (Out = fopen("INFILE.DAT", "w")) == NULL) exit(0);

 if( (In = fopen("INFILE.TMP", "r")) == NULL)
 {
  printf("Could not open INFILE.TMP\n");
  exit(0);
 }
 buf=line;
 fgets(line, maxlength, In);
 Trim(line);
 fprintf(Out,"%s\n",line);
 fgets(line, maxlength, In);
 Trim(line);
 buf=line+12;      /* position of "number of components" in the  option line*/
 sprintf(buf,"%2d",Tot_Reaction_strings);
 fprintf(Out,"%s     0.0\n",line);
 while(1)
 {
  buf=fgets(line, maxlength, In);
  if( buf == NULL) break;       /*  End of file */

  Trim(line);
  fprintf(Out,"%s\n",line);
 }
 fclose(In);
 fclose(Out);
}


void Copy_TMP_to_DAT()
{
 char line[130];         /* Next line from header file */
 short maxlength=128; /* Max line length          */

 char *buf; /* for diagnostic purpose */

 if( (Out = fopen("INFILE.DAT", "w")) == NULL) exit(0);
```

```
if( (In = fopen("INFILE.TMP", "r")) == NULL)
{
 printf("Could not open INFILE.DAT\n");
 exit(0);
}
buf=line;
while(1)
{
 buf=fgets(line, maxlength, In);
 if( buf == NULL) break;        /* End of file */

 Trim(line);
 fprintf(Out,"%s\n",line);
}
end_of_solution:
fclose(In);
fclose(Out);
}
```

SKi STATENS KÄRNKRAFTINSPEKTION
Swedish Nuclear Power Inspectorate

Postal address	Office	Telephone	Telex	Telefax
Box 27106 S-102 52 Stockholm	Sehlstedtsgatan 11	+46-8 665 44 00	11961 SWEATOM S	+46-8 661 90 86