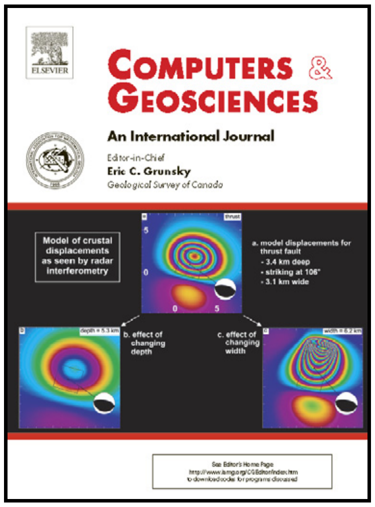


Modules based on the geochemical model PHREEQC for use in scripting and programming languages

Scott R. Charlton, David L. Parkhurst

PII:	S0098-3004(11)00065-3
DOI:	doi:10.1016/j.cageo.2011.02.005
Reference:	CAGEO 2539

To appear in: *Computers & Geosciences*

Received date:	8 November 2010
Revised date:	1 February 2011
Accepted date:	7 February 2011

Modules Based on the Geochemical Model PHREEQC for Use in Scripting and Programming Languages

By Scott R. Charlton and David L. Parkhurst*
U.S. Geological Survey
Denver Federal Center, P.O. Box 25046, MS 413, Denver, CO, USA

E-mail addresses: charlton@usgs.gov and dlpark@usgs.gov*
*Corresponding author
Phone 303 236 5098
Fax 303 236 5034

Submitted to Computers & Geosciences, November 8, 2010
Revised and resubmitted February 1, 2011

Abstract

The geochemical model PHREEQC is capable of simulating a wide range of equilibrium reactions between water and minerals, ion exchangers, surface complexes, solid solutions, and gases. It also has a general kinetic formulation that allows modeling of non-equilibrium mineral dissolution and precipitation, microbial reactions, decomposition of organic compounds, and other kinetic reactions. To facilitate use of these reaction capabilities in scripting languages and other models, PHREEQC has been implemented in modules that easily interface with other software. A Microsoft COM (Component Object Model) has been implemented, which allows PHREEQC to be used by any software that can interface with a COM server—for example, Excel®, Visual Basic®, Python, or MATLAB®. PHREEQC has been converted to a C++ class, which can be included in programs written in C++. The class also has been compiled in libraries for Linux and Windows that allow PHREEQC to be called from C++, C, and Fortran. A limited set of methods implement the full reaction capabilities of PHREEQC for each module. Input methods use strings or files to define reaction calculations in exactly the same formats used by PHREEQC. Output methods provide a table of user-selected model results, such as concentrations, activities, saturation indices, or densities.

The PHREEQC module can add geochemical reaction capabilities to surface-water, groundwater, and watershed transport models. It is possible to store and manipulate solution compositions and reaction information for many cells within the module. In addition, the object-oriented nature of the PHREEQC modules simplifies implementation of parallel processing for reactive-transport models.

The PHREEQC COM module may be used in scripting languages to fit parameters; to plot PHREEQC results for field, laboratory, or theoretical investigations; or to develop new models that include simple or complex geochemical calculations.

Keywords

Geochemical modeling; PHREEQC; Reactive-transport modeling; COM, Component Object Model; C++, C, and Fortran.

Software Requirements

- COM Module—Microsoft Windows operating system, COM client software such as Excel®, Visual Basic®, Python, or MATLAB®
- Windows Library Module—C++, C, or Fortran compiler for Windows operating system; Visual Studio® and C++ are needed to link with the library
- Linux Library Module—C++, C, or Fortran compiler for Linux operating system; C++ is needed to link with the library
- C++ Module—C++ compiler

All modules are available at http://wwwbrr.cr.usgs.gov/projects/GWC_coupled/phreeqc.

Any use of trade, product, or firm names in this publication is for descriptive purposes only and does not imply endorsement by the U.S. Government.

1 Introduction

PHREEQC (Parkhurst and Appelo, 1999) is a geochemical reaction model that simulates a variety of geochemical processes including equilibrium between water and minerals, ion exchangers, surface complexes, solid solutions, and gases. The general kinetic formulation allows modeling of non-equilibrium mineral dissolution and precipitation, microbial reactions, decomposition of organic compounds, and other kinetic reactions. PHREEQC has capabilities for 1D reactive transport, including such processes as multicomponent diffusion and transport of surface-complexing species. Finally, PHREEQC has inverse-modeling capabilities for the evaluation of the geochemical reactions that account for changes in water chemistry.

Because of the general geochemical speciation and reaction capabilities and the modular organization of input, PHREEQC often has been used as a geochemical calculation module (server) in other software programs (clients). PHREEQC has been used to calculate saturation indices, activities, and pH in water-quality data management software (Scientific Software Group, 2010, AquaChem), to generate predominance diagrams and estimate parameters (Kinniburgh and Cooper, 2010, PhreePlot), and to consider geochemical effects in watershed processes (Hartman et al., 2007, DayCent-Chem). Most commonly, PHREEQC has been used as the geochemical module for reactive-transport models. Reactive-transport environments include the unsaturated zone (Jacques and Šimůnek, 2004, HP1; Szegedi et al., 2008, RhizoMath; Wissmeier and Barry, 2010a, 2010b), the saturated zone (Mao et al., 2006, PHWAT; Parkhurst et al., 2004, 2010, PHAST; Prommer et al., 1999, PHT3D), radionuclide isolation (Källvenius and Ekberg, 2003, TACK), and acid mine drainage (Malmström et al., 2004, LaSAR-PHREEQC).

The coupling of PHREEQC to client programs has been both soft—reading and writing files by the client and server—and hard—modifying the source codes to add routines that transfer data between the client and server. Soft coupling is likely to be slow because of file writing and reading and because PHREEQC must read a database and perform extra calculations to redefine solution compositions as it is initialized at each geochemical step. PHREEQC lacks a facility to define directly essential solution data, particularly the solution charge balance, total moles of hydrogen, and total moles of oxygen. Hard coupling using specialized methods to set and retrieve data values can be difficult because of the complicated data structures in PHREEQC and because of complicated data dependencies among these structures.

This report presents PHREEQC modules designed to be used in scripting languages and integrated into C++, C, and Fortran programs. The modules are a hybrid between soft coupling—strings (or files) of PHREEQC input are used to specify calculations—and hard coupling—all data transfer between server and client can be done through a well-defined set of methods that do not require writing of files. The new modules rely on reorganization of the original PHREEQC code and addition of several new keyword data blocks that simplify extracting and modifying data within PHREEQC data structures. The interface to each module is a limited number of methods that are simple and intuitive for PHREEQC users, but retain the full capabilities of PHREEQC. Three examples are presented of geochemical tasks in different software environments to demonstrate a few of the possible uses for the new modules.

2 Methods

A C++ class for PHREEQC (hereafter, “IPhreeqc” is used to refer to the class or any PHREEQC modules) was implemented in three stages. The first stage was the development of a series of C++ classes that are equivalent to the original C structures that contain the data for solutions and reactants—equilibrium phases, gas phases, exchangers, surface complexers, solid solutions, and kinetic reactions. These classes were written during the development of PHAST (Parkhurst et al., 2004, 2010) and could be used directly by C++ programs that incorporate the IPhreeqc class. Most of the enhancements to PHREEQC discussed in section 2.1 are based on these additional C++ classes.

The second stage required much less development and was generally a rearrangement of the data and functions that comprise PHREEQC. All global and static data for PHREEQC were included in a header file for the IPhreeqc class. Similarly, all C functions were defined as methods of the class. The final stage was adding the interface, which is a series of methods described in section 2.2, and adding the wrappers necessary for the COM and library modules.

Thus, the IPhreeqc class is not a complete rewrite of PHREEQC with C++ classes and methods for all calculations; rather, it is an encapsulation to limit access to the data and functions of the original C code. The C code is essentially intact within the C++ class, but interactions with the class are limited to a well-defined set of methods.

2.1 Additions to PHREEQC

The reaction capabilities of PHREEQC and examples of their use are described in detail in Parkhurst and Appelo (1999). In its simplest form, a reaction in PHREEQC can be conceptualized as a solution plus a set of reactants that are put into a beaker and allowed to react. All of the moles of elements in the solution and in the reactants are combined in the beaker and a new system equilibrium is calculated. The reactants can include minerals, gases, ion exchangers, reactive surfaces, and solid solutions, which react to equilibrium, and kinetic reactions, which are functions of time and chemical compositions. PHREEQC allows definition of the initial compositions of the solution and reactants, calculates new compositions at the end of a reaction step, and finally saves these new compositions for use in subsequent reaction calculations. Compositions of all solutions and reactants are identified by a user-specified cell number.

In developing the reactive-transport model PHAST (Parkhurst and others, 2004, 2010), several new capabilities were added to PHREEQC, primarily to facilitate saving the compositional state of a simulation and restarting it. To that end, a series of input data blocks were devised that allow input of the exact contents of the data structures for solutions and other reactants. For solutions, the data block is named SOLUTION_RAW (for clarity, PHREEQC keywords are written with all capital letters); correspondingly named data blocks exist for equilibrium phases, exchangers, surfaces, solid solutions, gas phases, and kinetics.

A new keyword data block, DUMP, is used to write the state of any solution or reactant in the RAW format. Thus, the output from dumping a solution composition is a string or file that contains a SOLUTION_RAW data block, and is suitable for use as input to IPhreeqc.

In addition to the SOLUTION_RAW input data block, a SOLUTION_MODIFY data block is available. It uses exactly the same format as SOLUTION_RAW, but does not require a complete set of data. Thus, only data items that need to be changed can be updated. It is expected that the SOLUTION_MODIFY will be used to update the element composition of a solution following a transport calculation, without redefining some parts of the solution structure (for example, calculated quantities such as total alkalinity, mass of water, Pitzer activity coefficients, or, optionally, initial estimates of activities of the master species). Equivalent MODIFY data blocks are available for all other reactants.

The DELETE data block allows deleting some or all solution and reactant definitions. The COPY data block allows solutions and reactants to be replicated. Together, DUMP, MODIFY, DELETE, and COPY data blocks allow direct management of the solutions and reactants defined to PHREEQC.

The RUN_CELLS data block streamlines the process of setting up, running, and saving the results of a calculation for a cell. For cells selected by the data block specifications, all of the reactants with a given cell number are brought together and reacted, after which, the resulting compositions of the solution and reactants are saved back to the given cell number. Thus, "RUN_CELLS; 1-2" will cause solution 1 to react with all reactants numbered 1 and the compositions of the solution and reactants in cell 1 will be redefined to be the result of the reaction; similarly for cell 2.

2.2 IPhreeqc Class Methods

A client interacts with an IPhreeqc module through a set of methods. The key methods are listed in Table 1. These methods allow initializing the module and reading a thermodynamic database, running PHREEQC input (strings or files), and retrieving results from simulations. Other methods provide error and warning messages, get lengths of data items—number of rows, number of columns, number of lines—and control the writing of PHREEQC output files. Appendix 1 contains a complete list of methods for a Fortran module.

An IPhreeqc module is created in different ways depending on the software environment where it is used. Multiple instances of an IPhreeqc class can be created within the client program in all programming environments, even in C and Fortran. After a module is created, the **LoadDatabase** (for clarity, all IPhreeqc method names are written in bold font) or **LoadDatabaseString** method reads a thermodynamic database from a file or string, respectively. When the database has been read, a module is ready to perform PHREEQC calculations. Using **LoadDatabase** or **LoadDatabaseString** a second time will re-initialize the module and remove all data stored in it.

PHREEQC input can be defined and run in three different ways with an IPhreeqc module. First, the **AccumulateLine** method can be called sequentially to append PHREEQC input to an input buffer in IPhreeqc. When the entire input has been accumulated, it is run with the **RunAccumulated** method. The second way to run simulations is to define PHREEQC input in a string within the client program. This string is then submitted and run with the **RunString** method. Finally, it is possible to run PHREEQC input that has been saved in a file by using the **RunFile** method. Because reading and writing files to disk is slow, running simulations with many calls to **RunFile** is expected to be slower than using **RunString** and **RunAccumulated** with internally generated strings.

The SELECTED_OUTPUT and USER_PUNCH data blocks are used in a batch PHREEQC run to identify data to be written to a selected-output file. The data written can include most quantities calculated by the geochemical model—dissolved concentrations of elements, concentrations of aqueous species, activities of aqueous species, moles of minerals, and moles of kinetic reactants, for example. IPhreeqc makes special use of the data defined by the SELECTED_OUTPUT and USER_PUNCH data blocks, and allows this array of data to be returned to the client program by two methods that do not require reading or writing files. The **GetSelectedOutputValue** method is available in all modules and retrieves an individual data item at a given row and column from the array of selected-output results that was generated by the last call to a **RunAccumulated**, **RunString**, or **RunFile** method. The array has a row for every geochemical calculation that was performed and columns as defined by the SELECTED_OUTPUT and USER_PUNCH data blocks. The COM module has an additional method, **GetSelectedOutputArray**, which returns the entire array of the selected-output data.

A data item in the selected-output array may be an integer, real, or string value. IPhreeqc implements a simple variant object, which can contain any of these three data types. The IPhreeqc module requires slightly different handling of this variant object depending on whether the module is called as a COM, or as C++, C, or Fortran program elements.

A new PHREEQC capability to write (DUMP) data values allows access to the complete internal definition of each solution and reactant. The dumped data values are written in keyword data blocks that are suitable for input back into IPhreeqc (RAW data blocks, section 2.1). The **GetDumpString** method allows the raw keyword data blocks to be captured by the client program. (In Fortran, the dump string must be captured line-by-line with the GetDumpStringLine method.) The dumped data can be modified and reintroduced to an IPhreeqc module by use of the MODIFY data blocks (section 2.1) or transferred to another IPhreeqc module. The DUMP and the set of MODIFY keyword data blocks provide the basis for "get" and "set" methods, whereby the client program can control the data items of the module's solutions and reactants.

2.3 The COM Module

The COM module was implemented using Microsoft's Active Template Library (ATL). Through the use of C++ templates ATL provides standard implementations required by all COM objects. Each method and property was implemented by wrapping calls to the underlying IPhreeqc C++ methods. Methods containing string arguments required additional code to handle the necessary conversions between native COM strings (BSTR data type) and standard C strings. It also was necessary to convert the simplified IPhreeqc variant into a COM variant (VARIANT data type) for the **GetSelectedOutputValue** and **GetSelectedOutputArray** methods. The **GetSelectedOutputArray** method additionally uses an array (SAFEARRAY data type) of COM variants to return the selected-output array.

Programming environments designed to support COM objects (Visual Basic®, Python, or MATLAB®, for example) are able to use these COM variants directly and interchange them with their own native data types.

2.4 C++, C, and Fortran Modules

IPhreeqc libraries are available that allow use of IPhreeqc by C++, C, and Fortran programs; a library and equivalent DLL are available for Windows operating systems and source code for a library is available to be compiled for Linux or other Unix operating systems. The same Windows library (or DLL) or Linux library is linked no matter which of the three programming languages is used for the client program. However, each programming language requires a different header or "include" file in the client program. Header files for C++ and C and include files for Fortran77 and Fortran90 are included in the distribution of each of the library modules.

The use of the IPhreeqc methods is slightly different for C++, C, and Fortran to comply with the syntax of each language. The **GetSelectedOutputArray** method is not available in C++, C, or Fortran modules.

2.4.1 C++ Modules

Instances of the IPhreeqc C++ class can be used by linking with the IPhreeqc library. Alternatively, if the client of the IPhreeqc module is a C++ program, then the source code for the module could be compiled directly into the client program. In this case, it is possible to use the internal C++ classes for solutions and equilibrium phase, gas phase, exchange, surface, solid solution, and kinetic reactants. Use of these and other C++ classes included in the source code for IPhreeqc could simplify data storage and manipulation. When compiled into the client, it also is possible to extend the set of methods for the IPhreeqc class (or the other classes) to simplify data communication between the client and the IPhreeqc class.

The header file *IPhreeqc.hpp* is needed to compile C++ code that uses the IPhreeqc class, whether the C++ class is defined by integrating the source code or by using the IPhreeqc library. The class is instantiated by using normal C++ syntax for class objects.

Methods are called by using the standard C++ syntax for methods of objects. For a C++ module, the **GetSelectedOutputValue** method returns the IPhreeqc variant, which can contain an integer, double, string value, or error code. The definition of the variant and its methods are defined in the header file, *Var.h*.

2.4.2 C Modules

All methods for the C modules are functions. The client program must include the header file *IPhreeqc.h*, which includes the prototypes for the methods and the definition of the IPhreeqc variant. The **GetSelectedOutputValue** method returns the IPhreeqc variant.

2.4.3 Fortran Modules

The methods listed in Appendix 1 are subroutine and function calls. Fortran90 client programs must include the file *IPhreeqc.f90.inc*, which defines constants and the Fortran interfaces for the IPhreeqc methods. Fortran77 programs must include the file *IPhreeqc.f.inc* to define the constants and function types.

The IPhreeqc variant was not implemented in Fortran. Instead, the argument list of **GetSelectedOutputValue** contains three additional arguments, an integer type of the selected-output value (indicating integer, real, string, or error code), a real number, and a string value. If the type of the return value is string, the real number is not meaningful. If the type is integer or real, the value is returned as a real number in the real argument and the value is written as a string into the string argument.

3 Discussion

A wide variety of uses are possible for the IPhreeqc modules. Three general classes of users are envisioned: (1) researchers who use PHREEQC for interpretation of laboratory or field data and would like to use Excel® to store and plot results, (2) researchers who need more complex geochemical calculations and could use the flexibility of embedding a geochemical module in a scripting language such as Python or Visual Basic®, and (3) program developers who need a geochemical module for reactive-transport codes or who need to incorporate a geochemical calculation [calcium carbonate precipitation potential (CCPP) or base neutralizing capacity, for example] into their software. Three examples are given to demonstrate how IPhreeqc might be used by each of these three classes of users. The examples are made as simple as possible, while still demonstrating the utility of IPhreeqc in three different software environments.

3.1 Use of a COM Module in Excel®

Once installed on a computer, the IPhreeqc COM module can be used in Excel® Visual Basic for Applications® (VBA) macros. One common use for PHREEQC is to calculate saturation indices for a set of chemical analyses. Figure 1 (top) shows a PHREEQC input file that has been entered on sheet 1 of an Excel® workbook. The analytical data are entered in a set of columns headed by the PHREEQC nomenclature for elements and element valence states. Lines 1-2 and 7-10 are added to make a complete PHREEQC input set that performs speciation calculations and generates selected output that contains the saturation indices for calcite, dolomite, and gypsum and the log partial pressure for $CO_2(g)$.

Table 2 contains a VBA macro that creates the PHREEQC module, formats the data in sheet 1 as a PHREEQC input string, runs the string, and places the results in sheet 2 of the Excel® workbook. The *phreeqc.dat* database is assumed to be available in the directory containing the Excel® spreadsheet, but the macro could be modified with a path to a PHREEQC database. In the example, saturation indices are calculated as shown in figure 1 (bottom). In terms of the macro, no restriction is placed on the input that is defined in sheet 1; any PHREEQC input set could be defined on sheet 1 and the macro would place the selected-output results in sheet 2.

3.2 Use of a Module in Python

This example uses the COM module with the Python scripting language in a Windows environment. The task in the example is to calculate the solubility of gypsum as a function of NaCl concentration for two different aqueous models—the ion-association model, as developed in WATEQ4F (Ball and Nordstrom, 1991) and implemented in *wateq4f.dat*, and the specific ion interaction approach of Pitzer (1973), as originally coded in PHRQPITZ (Plummer et al., 1988) and implemented in *pitzer.dat*.

The Python script for the example is shown in table 3. The main program (last block of code) defines PHREEQC input for the simulation and specifies that the solubility of gypsum be calculated for increments of 0.1 moles of NaCl. The function *show_results* creates an IPhreeqc module for each database, runs the simulation in each module, and retrieves the data in the variables *nacl_conc*, *wateq4f_values*, and *pitzer_values*. The Python utility matplotlib (http://matplotlib.sourceforge.net/) is then used to produce a plot that compares the two results (figure 2). The specific ion interaction approach is a good fit to experimental data (Harvie and Weare, 1980). The ion-association model is generally applicable at lower ionic strengths and, indeed, the results of the ion-association model deviate from the more accurate Pitzer results at high ionic strengths.

3.3 Use of a Module in Fortran

The third example demonstrates use of IPhreeqc in a Fortran90 program. An equivalent C program is provided in Appendix 2. The program works with two cells that represent a reactive-transport model. Initial conditions are defined in the file *ic* (table 4), where both cells initially are filled with pure water. Cell 1 has an equilibrium-phases definition that contains carbon dioxide with a partial pressure of $10^{-1.5}$, whereas cell 2 has an equilibrium phases definition that contains calcite. The file *ic* also contains a definition for SELECTED_OUTPUT that writes the total number of moles of H, O, Ca, and C, plus the pH and saturation ratio (SR) for calcite (*IAP/K*, where *IAP* is ion activity product and *K* is the equilibrium constant).

In the Fortran90 program (table 5), the *phreeqc.dat* database is loaded, and the initial conditions file is run, which places pure water in each of the two cells. Then the solution and reactants (equilibrium phases) for cell 1 are reacted with the RUN_CELLS data block, which produces a water in equilibrium with a soil-zone partial pressure of carbon dioxide.

In place of a true dispersive-transport step, the solution from cell 1 is simply advected to cell 2. The data from cell 1 are retrieved in the subroutine *ExtractWrite* by sequentially retrieving the columns of the selected-output array. After retrieving the data, the pH and saturation ratio for cell 1 are written to the output screen. Returning to the main program, the SOLUTION_MODIFY data block is constructed, which specifies the total moles of elements in cell 2 to be equal to those just retrieved from cell 1. The RUN_CELLS keyword data block is used to equilibrate the new water composition in cell 2 with the reactants in cell 2, namely calcite. The results of this calculation are again retrieved and written by the subroutine *ExtractWrite*. The results show that the water in cell 1 has a pH of 4.66 and a calcite saturation ratio of 0.0 (because calcium is absent), whereas the water in cell 2 has a pH of 7.68 and a calcite saturation ratio of 1.0 (equilibrium with calcite).

Some care is needed with the units of solutions and reactants when using IPhreeqc for reactive-transport simulations. PHREEQC stores all quantities of elements, exchangers, equilibrium phases, and other reactants, in units of moles, not in units of concentration. Although PHREEQC does all of its calculations with solutions in terms of molality (mol/kg water), only the numbers of moles of each element and the mass of water are stored; a solution definition may have a mass of water that differs substantially from 1.0 kg. Thus, solution compositions are defined by the number of moles of elements, including H and O, and the equivalents of charge imbalance. In the file *ic* (table 4), the function TOTMOLE was used, which returns the total number of moles of an element in solution. The total numbers of moles in solution are the quantities needed for the SOLUTION_MODIFY data block that was used in the advection step of the example (table 5). For reactive-transport calculations, it may be necessary to convert the solution compositions to concentration units (mol/L, ppm, or mass fraction, for example) for the transport calculation and then back to moles for the IPhreeqc calculations. Alternatively, fluid flow and solute transport with species-independent diffusion can be considered as an assembage of fluxes of individual elements, and the governing equations can be derived in terms of transport of moles of individual elements (Wissmeier and Barry, 2008). Regardless of the transport equations selected, it is necessary to transport H, O, and charge, in addition to any other elements in the system to maintain complete solution composition and correct charge imbalances.

3.4 Parallelized Calculations Using IPhreeqc Modules

Because IPhreeqc modules are independent objects in the sense of object-oriented programming, parallelization with threads or multiple processes is straightforward. Here, multiple processors are discussed, but the use of threads is similar. In general, the strategy is to start multiple processes, each of which creates an IPhreeqc module. Each module is then assigned part of the geochemical calculation tasks. Data are passed among the processes, either by queues or messages. The passed data would be primarily chemical compositions, which could be DUMP strings, _MODIFY data blocks, or arrays of elemental compositions.

An example calculation (*parallel_advect.py*) using the multiprocessing package of Python is presented in the supplemental material. The example reproduces the results of the advective case of example 11 in the PHREEQC manual (Parkhurst and Appelo, 1999). The Python script uses multiple processes and queues to divide the geochemical calculations for a column of cells equally among a specified number of processes.

4 Summary and Conclusions

PHREEQC can simulate a wide range of reactions between water and solids, including reactions with minerals, gases, ion exchangers, surface complexers, and solid solutions. Irreversible kinetic reactions also can be simulated. Because of the generality and ease of use, PHREEQC has been integrated as the geochemical calculation module in several programs; however, the integration of PHREEQC into other codes has been difficult and time consuming. IPhreeqc is a set of modules that have been developed specifically to allow easy integration of PHREEQC into other software. All of the simulation and data-storage capabilities of PHREEQC are accessible in IPhreeqc modules through a limited set of methods.

IPhreeqc modules can be used in a number of software environments. The COM module can be used by any software that supports the COM interface—Excel® (Visual Basic for Applications®), Python, or MATLAB® for example. The C++ class for IPhreeqc can be compiled into C++ programs, where the module and its underlying classes can be used or subclassed directly. Alternatively, libraries and DLLs allow the IPhreeqc modules to be used in C++, C, and Fortran programs on Windows or Linux operating systems. The modularity of IPhreeqc allows easy implementation of parallel processing for computationally intensive geochemical simulations.

The interface to the modules is a relatively small set of methods, which combined with enhancements to PHREEQC, implements all of the capabilities of PHREEQC and allows all of the underlying data that define solutions and reactants to be retrieved and modified. While it is admittedly somewhat cumbersome to generate strings to perform all of the IPhreeqc calculations, the string approach has the advantage that the interface is simple and intuitive. In addition, the interface methods should not need modification, even if new features are added to PHREEQC.

IPhreeqc can be used for a variety of geochemical simulation tasks, including analysis of field and laboratory data, comparison and fitting of thermodynamic data, and reactive-transport simulations. Two applications have successfully used IPhreeqc modules: Kinniburgh and Cooper (2010) have integrated the library module into PhreePlot to plot predominance diagrams and fit thermodynamic data, and Wissmeier and Barry (2010b) have used the COM module with MATLAB® and COMSOL Multiphysics® to simulate reactive-transport in the unsaturated zone. The module may prove useful in a number of other fields, including water treatment, contaminant mitigation, and chemical engineering.

5 Acknowledgements

The authors thank David Kinniburgh, Honorary Research Associate British Geological Survey, for having inspired the development of the PHREEQC module and for his help enhancing PHREEQC. We also thank Mike Müller, Hydrocomputing.com, for the versions of the Python examples presented in the report.

6 References

Ball, J.W., Nordstrom, D.K., 1991. User's manual for WATEQ4F, with revised thermodynamic data base and test cases for calculating speciation of major, trace, and redox elements in natural waters. U. S. Geological Survey Water-Resources Investigations Report 91-183, 189 pp.

Hartman, M.D., Baron, J.S., Ojima, D.S., 2007. Application of a coupled ecosystem-chemical equilibrium model, DayCent-Chem, to stream and soil chemistry in a Rocky Mountain watershed. Ecological Modeling, 200(3-4), 493-510.

Harvie, C.E., Weare, J.H., 1980. The prediction of mineral solubilities in natural waters—the Na–K–Mg–Ca–Cl–SO_4–H_2O system from zero to high concentration at 25^oC. Geochimica et Cosmochimica Acta, 44, 981-997.

Jacques, D., Šimůnek, J. 2004. User manual of the Multicomponent variably-saturated transport model HP1 (Version 1.0): Description, Verification and Examples. SCK•CEN, Mol, Belgium, BLG-998, 79 pp.

Källvenius, G., Ekberg, C., 2003. TACK—a program coupling chemical kinetics with a two-dimensional transport model in geochemical systems. Computers & Geosciences, 29(4), 511-521.

Kinniburgh, D.G., Cooper, D.M., 2010. PhreePlot—Creating graphical output with PHREEQC. Accessed March 23, 2010. http://www.phreeplot.org.

Malmström, M.E., Destouni, G., Martinet, P., 2004. Modeling expected solute concentration in randomly heterogeneous flow systems with multicomponent reactions. Environmental Science & Technology, 38(9), 2673-2679.

Mao, X., Prommer, H., Barry, D.A., Langevin, C.D., Panteleit, B., Li, L., 2006. Three-dimensional model for multi-component reactive transport with variable density groundwater flow. Environmental Modelling & Software, 21(5), 615-628.

Parkhurst, D.L., and Appelo, C.A.J., 1999, User's guide to PHREEQC (Version 2)—A computer program for speciation, batch-reaction, one-dimensional transport, and inverse geochemical calculations: U.S. Geological Survey Water-Resources Investigations Report 99–4259, 312 pp.

Parkhurst, D.L., Kipp, K.L., and Charlton, S.R., 2010. PHAST version 2 —A program for simulating groundwater flow, solute transport, and multicomponent geochemical reactions. U. S. Geological Survey Techniques and Methods 6—A35, 235 pp.

Parkhurst, D.L., Kipp, K.L., and Engesgaard, P., and Charlton, S.R., 2004. PHAST—A program for simulating ground-water flow, solute transport, and multicomponent geochemical reactions. U. S. Geological Survey Techniques and Methods 6—A8, 154 pp.

Pitzer, K.S., 1973, Thermodynamics of electrolytes, I: Theoretical basis and general equations. Journal of Physical Chemistry, 77(2), 268-277.

Plummer, L.N., Parkhurst, D.L., Fleming, G.W., Dunkle, S.A., 1988. A computer program incorporating Pitzer's equations for calculation of geochemical reactions in brines. U. S. Geological Survey Water-Resources Investigations Report 88-4153, 310 pp.

Prommer, H., Davis, G.B., Barry, D.A., 1999. PHT3D—A three-dimensional biogeochemical transport model for modelling natural and enhanced remediation, in: Johnston, C.D. (Ed.), Contaminated Site Remediation: Challenges Posed by Urban and Industrial Contaminants. Centre for Groundwater Studies, Fremantle, Western Australia, pp. 351-358.

Scientific Software Group, 2010. Aqueous Geochemical Analysis, Plotting and Modeling. Accessed March 23, 2010. http://www.scientificsoftwaregroup.com/pages/software.php

Szegedi, K. Vetterlein, D., Nietfield, H. Jahn, R., Neue, H-U., 2008. New tool RhizoMath for modeling coupled transport and speciation in the rhizosphere. Vadose Zone Journal, 7, 712-720. doi:10.2136/vzj2007.0064

Wissmeier, L., Barry, D.A., 2008. Reactive transport in unsaturated soil: Comprehensive modelling of the dynamic spatial and temporal mass balance of water and chemical components. Advances in Water Resources, 31(5), 858-875.

Wissmeier, L., Barry, D.A., 2010a. Implementation of variably saturated flow into PHREEQC for the simulation of biogeochemical reactions in the vadose zone. Environmental Modelling & Software, 25(4), 526-538.

Wissmeier, L., Barry, D.A., 2010b. Simulation tool for variably saturated flow with comprehensive geochemical reactions in two- and three-dimensional domains. Environmental Modelling & Software, 26(2011), 210-218. doi:10.1016/j.envsoft.2010.07.005

Appendix 1

A complete list of methods for IPhreeqc Fortran modules is given in table A1. The most important methods have been used in the examples in the text. These methods include **CreateIPhreeqc**, **LoadDatabase**, **RunFile**, **RunString**, **RunAccumulated**, **GetSelectedOutputValue**, and **DestroyIPhreeqc**. Additional information for the set of Fortran methods is provided here. Note that additional methods are available to COM, C, and C++ programs that are not available in Fortran: **GetDumpString**, **GetErrorString**, **GetWarningString**, and **GetOutputArray** (COM only).

Most methods return an integer value. Non-negative return values indicate successful completion of the method. If the integer is less than zero, an error has occurred during the invocation of the method and the cause of the error can be determined by using the **OutputErrorString** method or by a call to the **GetErrorStringLineCount** method and sequential calls to the **GetErrorStringLine** method. An IPhreeqc run also can produce warnings, which are conditions that do not cause failure of the run, but may indicate problems with input or difficulties in obtaining a numerical solution to the input definitions. Warnings can be obtained with calls to the **GetWarningStringLineCount** method and sequential calls to the **GetWarningStringLine** method.

An IPhreeqc module has several properties that control file output from the module. An IPhreeqc run can write data to an output file, a selected-output file, an error file, a dump file (complete item-by-item output of solution or reactant data), and a log file (rarely used). The methods **SetOutputFileOn** , **SetSelectedOutputFileOn**, **SetErrorFileOn**, **SetDumpFileOn**, and **SetLogFileOn** can be used to set the properties that activate or suspend writing to the respective files. The status of the properties related to file writing can be obtained by the methods **GetOutputFileOn** , **GetSelectedOutputFileOn**, **GetErrorFileOn**, **GetDumpFileOn**, and **GetLogFileOn**.

Several methods apply to the input buffer that is used to accumulate lines of PHREEQC input. The **AccumulateLine** method appends one or more lines to the input buffer. The **OutputAccumulatedLines** method prints the state of the input buffer and the **ClearAccumulatedLines** method clears the buffer. The input can be run with the **RunAccumulated** method.

Methods related to retrieving results from an IPhreeqc run include: **GetSelectedOutputRowCount**, which returns the number of rows in the selected-output array; **GetSelectedOutputColumnCount**, which returns the number of columns in the selected-output array; and **GetSelectedOutputValue**, which returns a specified row-column value from the selected-output array.

It can be convenient to have a list of elements that have been defined by input to an IPhreeqc module. The **GetComponentCount** and **GetComponent** methods allow retrieval of all the elements that are presently defined in the module in solutions and reactants. This is not the complete list of components defined in the database, but the list of all elements that have been used in SOLUTION, EQUILIBRIUM_PHASES, EXCHANGE, GAS_PHASE, KINETICS, REACTION, SOLID_SOLUTION, and SURFACE data blocks. Solutions or reactants that have been deleted with the DELETE keyword data block are not currently defined and are not considered. This list could be used as the list of components (in addition to H, O, and charge) that need to be transported in multicomponent reactive-transport simulations.

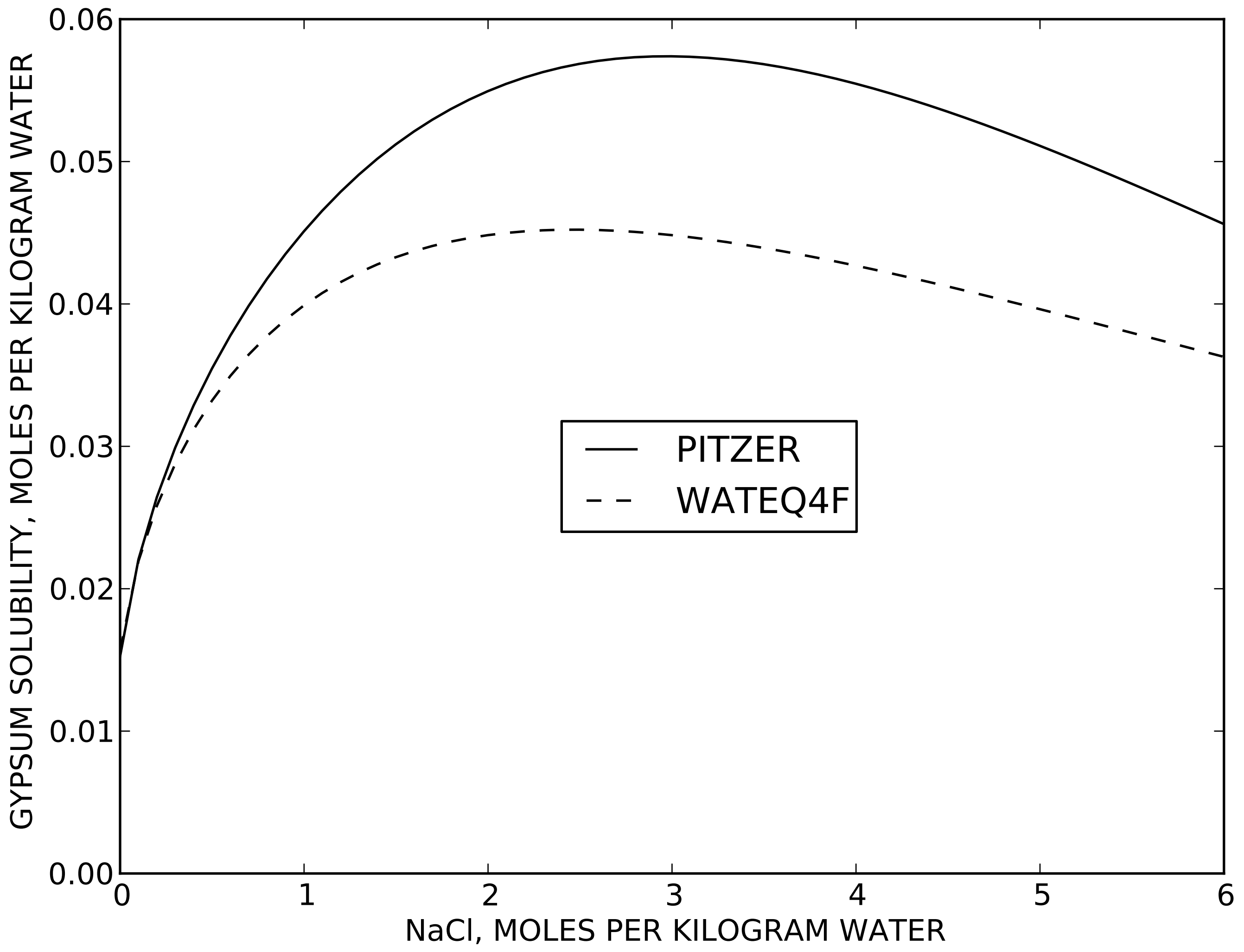
The final methods described here are related to the dump string of the module. The dump string contains the results from using the DUMP keyword in PHREEQC input. First, the dump string must be activated before an IPhreeqc run with a call to the **SetDumpStringOn** method. After the IPhreeqc run, the dump string can be retrieved by the client program line by line. The **GetDumpStringLineCount** method returns the number of lines in the dump string. The **GetDumpStringLine** method returns a specified line from the dump string.

Appendix 2

Table A2 gives a C program that is equivalent to the Fortran program of the third example. Apart from the differences in language syntax, there is one important difference in the C IPhreeqc module related to memory usage. Whereas, no memory problems can occur in Fortran or COM usage, a variable of type VAR will leak memory in C or C++ if it is used to store a string, and it is not cleared before it goes out of scope. A memory leak is a condition where memory is not freed even though it is no longer used. Memory leaks cause an accumulation of unusable computer memory, and a consequent decrease in the memory available for program use. Although the memory leak only will occur in C or C++ when using a variable of type VAR to store a string, it is good practice to clear any type VAR variable with VarClear after each use, as is done near the end of the *void ExtractWrite* function. Note that if a variable of type VAR is assigned a new value, it automatically will be cleared before the new value is stored.

List of Figures:

List of Tables:


0.06
0.05
0.04
0.03
0.02
0.01
0.00
GYPSUM SOLUBILITY, MOLES PER KILOGRAM WATER
0
1
2
3
4
5
6
NaCl, MOLES PER KILOGRAM WATER
PITZER
WATEQ4F

SOLUTION_SPREAD							
-units mg/L							
Temp	pH	Ca	Mg	Na	Cl	S(6)	Alkalinity
18.7	6.86	114.7	8.109	12.03	2.787	19.007	298
18.4	6.9	95.79	49.58	20.39	28.327	31.544	348
18.3	6.91	80.81	39.61	4.934	8.37	10.783	329
SELECTED_OUTPUT							
-reset false							
-SI	Calcite	Dolomite	Gypsum	CO2(g)			
END							

si_Calcite	si_Dolomite	si_Gypsum	si_CO2(g)
-0.10	-1.08	-2.13	-1.36
-0.11	-0.24	-2.06	-1.34
-0.17	-0.39	-2.55	-1.37

Table 1. Key methods for IPhreeqc modules

Method	**Function**
LoadDatabase(*FileName*)	Reads the database from the specified file
LoadDatabaseString(*Input*)	Reads the database from the input string
AccumulateLine(*String*)	Append the input string to the input buffer for the module
RunAccumulated()	Runs PHREEQC based on the input buffer defined by calls to **AccumulateLine**
RunFile(*FileName*)	Runs PHREEQC based on the input in the specified file
RunString(*InputString*)	Runs PHREEQC based on the specified input string
GetSelectedOutputArray()	Returns an array with the selected-output results from the last run (**RunAccumulated**, **RunFile**, or **RunString**). (This method is available only in the COM module)
GetSelectedOutputValue(*Row*, *Column*)	Returns the value from the specified row and column of the selected-output array, which contains results from the last run (**RunAccumulated**, **RunFile**, or **RunString**)
GetDumpString()	Returns a string containing the output as defined by the DUMP data block of the last **RunAccumulated**, **RunFile**, or **RunString** command

Table 2. Excel® Visual Basic for Applications® macro that takes PHREEQC input from sheet 1 of a workbook and puts selected output in sheet 2 of workbook

```
Sub RunPhreeqc()
  On Error GoTo ErrHandler:
  ChDir ActiveWorkbook.Path
  Set Phreeqc = CreateObject("IPhreeqcCOM.Object")
  Db = "phreeqc.dat"
  Phreeqc.LoadDatabase (Db)

  'Format input from sheet1
  Dim Istring As String
  Worksheets("Sheet1").Activate
  FirstRow = ActiveSheet.UsedRange.Row
  FirstColumn = ActiveSheet.UsedRange.Column
  For r = FirstRow To (FirstRow + ActiveSheet.UsedRange.Rows.Count)
    For c = FirstColumn To (FirstColumn + ActiveSheet.UsedRange.Columns.Count)
      Istring = Istring & CStr(Cells(r, c)) & vbTab
    Next c
    Istring = Istring & vbNewLine
  Next r

  'Run and save selected output to sheet2
  Phreeqc.RunString (Istring)
  arr = Phreeqc.GetSelectedOutputArray()
  Worksheets("Sheet2").Activate
  Range(Cells(1, 1), Cells(Phreeqc.RowCount, Phreeqc.ColumnCount)) = arr
  MsgBox "Phreeqc ran successfully."
  Exit Sub

ErrHandler:
  MsgBox "Phreeqc errors: " & Phreeqc.GetErrorString()
End Sub
```

Table 3. Python script that plots the solubility of gypsum as a function of NaCl concentration as calculated by the Pitzer and WATEQ4F databases

```
"""Compares gypsum solubility for WATEQ4F and Pitzer databases.
"""
# Import standard library modules first.
import os
# Then get third party modules.
from win32com.client import Dispatch
import matplotlib.pyplot as plt

def selected_array(db_path, input_string):
    """Load database via COM and run input string.
    """
    dbase = Dispatch('IPhreeqcCOM.Object')
    dbase.LoadDatabase(db_path)
    dbase.RunString(input_string)
    return dbase.GetSelectedOutputArray()

def show_results(input_string):
    """Get results for different databases
    """
    wateq4f_result = selected_array('wateq4f.dat', input_string)
    pitzer_result  = selected_array('pitzer.dat', input_string)
    # Get data from the arrays.
    nacl_conc      = [entry[0] for entry in wateq4f_result][1:]
    wateq4f_values = [entry[1] for entry in wateq4f_result][1:]
    pitzer_values  = [entry[1] for entry in pitzer_result][1:]
    # Plot
    plt.plot(nacl_conc, pitzer_values, 'k', nacl_conc, wateq4f_values,'k--')
    plt.axis([0, 6, 0, .06])
    plt.legend(('PITZER','WATEQ4F'), loc = (0.4, 0.4))
    plt.ylabel('GYPSUM SOLUBILITY, MOLES PER KILOGRAM WATER')
    plt.xlabel('NaCl, MOLES PER KILOGRAM WATER')
    plt.show()

if __name__ == '__main__':
    # This will only run when called as script from the command line
    # and not when imported from another script.
    INPUT_STRING = """
    SOLUTION 1
    END
    INCREMENTAL_REACTIONS
    REACTION
       NaCl 1.0
       0 60*0.1 moles
    EQUILIBRIUM_PHASES
       Gypsum
    USE solution 1
    SELECTED_OUTPUT
       -reset false
       -total Na S(6)
    END"""
    show_results(INPUT_STRING)
```

Table 4. Initial conditions and selected-output definitions for Fortran90 example

```
# File ic
SOLUTION 1-2
END
EQUILIBRIUM_PHASES 1
   CO2(g) -1.5 10

EQUILIBRIUM_PHASES 2
   Calcite 0   10
SELECTED_OUTPUT
   -reset false
USER_PUNCH
   -Heading  charge    H   O   C   Ca  pH  SR(calcite)
   10 PUNCH charge_balance
   20 PUNCH TOTMOLE("H"), TOTMOLE("O"), TOTMOLE("C"), TOTMOLE("Ca")
   30 PUNCH -LA("H+"), SR("calcite")
END
```

Table 5. Fortran90 program that performs advection and chemical reactions for two cells

```
module Subs
  integer    (kind=4), dimension(7) :: vt
  real       (kind=8), dimension(7) :: dv
  character (len=100), dimension(7) :: sv
  integer                           :: Id
  contains

  subroutine ExtractWrite(cell)
    include "IPhreeqc.f90.inc"
    integer    (kind=4), intent(in) :: cell
    do j = 1, 7
      ! Headings are on row 0
      Ierr = GetSelectedOutputValue(Id,1,j,vt(j),dv(j),sv(j))
      if(Ierr .ne. IPQ_OK) call EHandler()
    enddo
    write(*,"(a,i2/2(5x,a,f7.2))") "Cell",cell,"pH:",dv(6),"SR(calcite):",dv(7)
  end subroutine ExtractWrite

  subroutine EHandler()
    include "IPhreeqc.f90.inc"
    call OutputErrorString(Id)
    stop
  end subroutine EHandler
end module Subs
program Advect
  use Subs
  include "IPhreeqc.f90.inc"
  character(len=1024) Istring

!Create module, load database, define initial conditions and selected output
  Id = CreateIPhreeqc()
  if (LoadDatabase(Id, "phreeqc.dat") .ne. 0) call EHandler()
  If (RunFile(Id, "ic") .ne. 0) call EHandler()

!Run cell 1, extract/write result
  if (RunString(Id, "RUN_CELLS; -cells; 1; END") .ne. 0) call EHandler()
  call ExtractWrite(1)

!Advect cell 1 solution to cell 2, run cell 2, extract/write results
  Ierr = AccumulateLine(Id, "SOLUTION_MODIFY 2")
  Ierr = AccumulateLine(Id, "   -cb      " // sv(1))
  Ierr = AccumulateLine(Id, "   -total_h " // sv(2))
  Ierr = AccumulateLine(Id, "   -total_o " // sv(3))
  Ierr = AccumulateLine(Id, "   -totals  ")
  Ierr = AccumulateLine(Id, "      C     " // sv(4))
  Ierr = AccumulateLine(Id, "      Ca    " // sv(5))
  Ierr = AccumulateLine(Id, "RUN_CELLS; -cells; 2; END")
  if (RunAccumulated(Id) .ne. 0) call EHandler()
  call ExtractWrite(2)

 !Destroy module
  if (DestroyIPhreeqc(Id) .ne. 0) call EHandler()
end program Advect
```

Table A1. Complete list of methods for a Fortran90 IPhreeqc module

[*Id*, number returned by the **CreateIPhreeqc** function; *N*, integer used to refer to the *N*th member of a list; *col*, column number; *comp*, variable to hold the *N*th component name, *logical*, a value of true or false; *Vtype,*integer variable; *Dvalue*, real variable ; *Svalue*, string variable]

Method	**Usage**
Function **AccumulateLine**(*Id*, *String*)	Appends one or more lines to the input buffer
Function **AddError**(*Id*, *String*)	Appends the string to the error string in the module and increments the error count
Function **AddWarning**(*Id*, *String*)	Appends the string to the warning string in the module
Function **ClearAccumulatedLines**(*Id*)	Clears the input buffer of the module
Function **CreateIPhreeqc**()	Create and initialize a module
Function **DestroyIPhreeqc**(*Id*)	Destroy a module
Subroutine **GetComponent**(*Id*, *N*, *Comp*)	Retrieve specified component name
Function **GetComponentCount**(*Id*)	Determine number of components currently used in the module
Function **GetDumpFileOn**(*Id*, *Logical*)	Retrieve the print setting for the dump file
Subroutine **GetDumpStringLine**(*Id*, *N*, *Line*)	Retrieve line from the lines generated by the DUMP data block
Function **GetDumpStringLineCount**(*Id*)	Retrieve number of lines generated by the DUMP data block
Function **GetDumpStringOn**(*Id*, *Logical*)	Retrieve the setting for saving dump information in a string
Function **GetErrorFileOn**(*Id*, *Logical*)	Retrieve the print setting for the error file
Subroutine **GetErrorStringLine**(*Id*, *N*, *Line*)	Retrieve specified line from the error messages
Function **GetErrorStringLineCount**(*Id*)	Retrieve number of lines in the error messages
Function **GetLogFileOn**(*Id*, *Logical*)	Retrieve the print setting for the log file
Function **GetOutputFileOn**(*Id*, *Logical*)	Retrieve the print setting for the output file
Function **GetSelectedOutputColumnCount**(*Id*)	Retrieve number of columns in selected output
Function **GetSelectedOutputFileOn**(*Id*, *Logical*)	Retrieve the print setting for the selected-output file
Function **GetSelectedOutputRowCount**(Id)	Retrieve number of rows in selected output
Function **GetSelectedOutputValue**(*Id*, *Row*, *Col*, *Vtype*, *Dvalue*, *Svalue*)	Retrieve selected-output value from specified row and column
Subroutine **GetWarningStringLine**(*Id*, *N*, *Line*)	Retrieve specified line from the warning messages
Function **GetWarningStringLineCount**(*Id*)	Retrieve number of lines in the warning messages
Function **LoadDatabase**(*Id, FileName*)	Reads the database from file
Function **LoadDatabaseString**(*Id*, *String*)	Reads the database from string

Subroutine **OutputAccumulatedLines**(*Id*)	Display the accumulated input buffer
Subroutine **OutputErrorString**(*Id*)	Display errors from the last run
Subroutine **OutputWarningString**(*Id*)	Display warnings from the last run
Function **RunAccumulated**(*Id*)	Run the input accumulated in the input buffer
Function **RunFile**(*Id*, *FileName*)	Run from a file
Function **RunString**(*Id*, *String*)	Run from a string
Function **SetDumpFileOn**(*Id*, *Logical*)	Set the switch for printing to the dump file
Function **SetDumpStringOn**(*Id*, *Logical*)	Set the switch for saving dump information in a string
Function **SetErrorFileOn**(*Id*, *Logical*)	Set the switch for printing to the error file
Function **SetLogFileOn**(*Id*, *Logical*)	Set the switch for printing to the log file
Function **SetOutputFileOn**(*Id*, *Logical*)	Set the switch for printing to the output file
Function **SetSelectedOutputFileOn(*Id*, *Logical*)**	Set the switch for printing to the selected-output file

Table A2. C program that performs advection and chemical reactions for two cells

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <IPhreeqc.h>
int id;
int vt[7];
double dv[7];
char sv[7][100];
char buffer[100];
void ExtractWrite(int cell)
{
        VAR v;
        int j;
        VarInit(&v);
        for (j = 0; j < 7; ++j) {
                GetSelectedOutputValue(id, 1, j, &v);
                vt[j] = v.type;
                switch (vt[j]) {
                case TT_DOUBLE:
                        dv[j] = v.dVal;
                        sprintf(sv[j], "%23.15e", v.dVal);
                        break;
                case TT_STRING:
                        strcpy(sv[j], v.sVal);
                        break;
                }
                VarClear(&v);
        }
        printf("Cell %d \n\tpH: %4.2f\tSR(calcite): %4.2f\n", cell, dv[5], dv[6]);
}
void EHandler(void)
{
        OutputErrorString(id);
        exit(EXIT_FAILURE);
}
const char *ConCat(const char *str1, const char *str2)
{
        strcpy(buffer, str1);
        return strcat(buffer, str2);
}
int main(void)
{
        /* Create module, load database, define initial conditions and selected output */
        id = CreateIPhreeqc();
        if (LoadDatabase(id, "phreeqc.dat") != 0) EHandler();
        if (RunFile(id, "ic") != 0) EHandler();

        /* Run cell 1, extract/write result */
        if (RunString(id, "RUN_CELLS; -cells; 1; END") != 0) EHandler();
        ExtractWrite(1);

        /* Advect cell 1 solution to cell 2, run cell 2, extract/write results */
        AccumulateLine(id, ConCat("SOLUTION_MODIFY 2",          ""   ));
        AccumulateLine(id, ConCat("   -cb      ",               sv[0]));
        AccumulateLine(id, ConCat("   -total_h ",               sv[1]));
        AccumulateLine(id, ConCat("   -total_o ",               sv[2]));
        AccumulateLine(id, ConCat("   -totals  ",               ""   ));
        AccumulateLine(id, ConCat("      C     ",               sv[3]));
        AccumulateLine(id, ConCat("      Ca    ",               sv[4]));
        AccumulateLine(id, ConCat("RUN_CELLS; -cells; 2; END", ""   ));
        if (RunAccumulated(id) != 0) EHandler();
        ExtractWrite(2);

        /* Destroy module */
        if (DestroyIPhreeqc(id) != IPQ_OK) EHandler();
        exit(EXIT_SUCCESS);
}
```

Research Highlights

- Modules for geochemical reaction calculations based on PHREEQC
- Mineral, gas, exchange, surface-complexation, solid-solution, and kinetic reactions
- For use in Excel, Python, C++, C, and Fortran
- Suitable for coupling geochemical reactions with transport models