The Geochemist's Workbench®
Release 17

ChemPlugin™ User's Guide

The Geochemist's Workbench®
Release 17

ChemPlugin™ User's Guide

Craig M. Bethke
Aqueous Solutions, LLC
Champaign, Illinois

Printed October 9, 2023

This document © Copyright 2023 by Aqueous Solutions LLC. All rights reserved. Earlier editions copyright 2000–2021. This document may be reproduced freely to support any licensed use of the GWB software package.

Software copyright notice: Programs GSS, Rxn, Act2, Tact, SpecE8, Gtplot, TEdit, React, Phase2, P2plot, X1t, X2t, Xtplot, and ChemPlugin © Copyright 1983–2023 by Aqueous Solutions LLC. An unpublished work distributed via trade secrecy license. All rights reserved under the copyright laws.

The Geochemist's Workbench®, ChemPlugin™, We put bugs in our software™, and The Geochemist's Spreadsheet™ are a registered trademark and trademarks of Aqueous Solutions LLC; Microsoft®, MS®, Windows 11®, and Windows 10® are registered trademarks of Microsoft Corporation; PostScript® is a registered trademark of Adobe Systems, Inc. Other products mentioned in this document are identified by the trademarks of their respective companies; the authors disclaim responsibility for specifying which marks are owned by which companies. The software uses zlib © 1995-2005 Jean-Loup Gailly and Mark Adler, and Expat © 1998-2006 Thai Open Source Center Ltd. and Clark Cooper.

The GWB software was originally developed by the students, staff, and faculty of the Hydrogeology Program in the Department of Geology at the University of Illinois Urbana-Champaign. The package is currently developed and maintained by Aqueous Solutions LLC at the University of Illinois Research Park.

Address inquiries to

Aqueous Solutions LLC
301 North Neil Street, Suite 400
Champaign, IL 61820 USA

Warranty: The Aqueous Solutions LLC warrants only that it has the right to convey license to the GWB software. Aqueous Solutions makes no other warranties, express or implied, with respect to the licensed software and/or associated written documentation. Aqueous Solutions disclaims any express or implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Aqueous Solutions does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the Licensed Software or documentation in terms of correctness, accuracy, reliability, currentness, or otherwise. Aqueous Solutions shall not be liable for any direct, indirect, consequential, or incidental damages (including damages for loss of profits, business interruption, loss of business information, and the like) arising out of any claim by Licensee or a third party regarding the use of or inability to use Licensed Software. The entire risk as to the results and performance of Licensed Software is assumed by the Licensee. See License Agreement for complete details.

License Agreement: Use of the GWB is subject to the terms of the accompanying License Agreement. Please refer to that Agreement for details.

Cover photo: Salinas de Janubio by Jorg Hackemann.

Contents

Chapter List

Contents

Introduction

ChemPlugin is a self-linking software object designed to add multicomponent reactive transport capabilities to any program that models the flow of an aqueous fluid. You can use ChemPlugin objects, in other words, to convert a flow model into a multicomponent reactive transport simulator. Whether you are developing a software application from scratch, or adapting an existing modeling program, ChemPlugin is the fastest and easiest way to create full-functioned reactive transport modeling software.

1.1 How it works

ChemPlugin is surprisingly easy to use. Your *client program*—the flow model—spawns any number of ChemPlugin *instances*, an instance being a copy of the ChemPlugin object (**Figure 1.1**). Each instance represents a portion of the system being modeled: a piece of the subsurface, a length of pipe, a volume of water in a lake or ocean, a reactor tank in a water treatment plant, and so on.

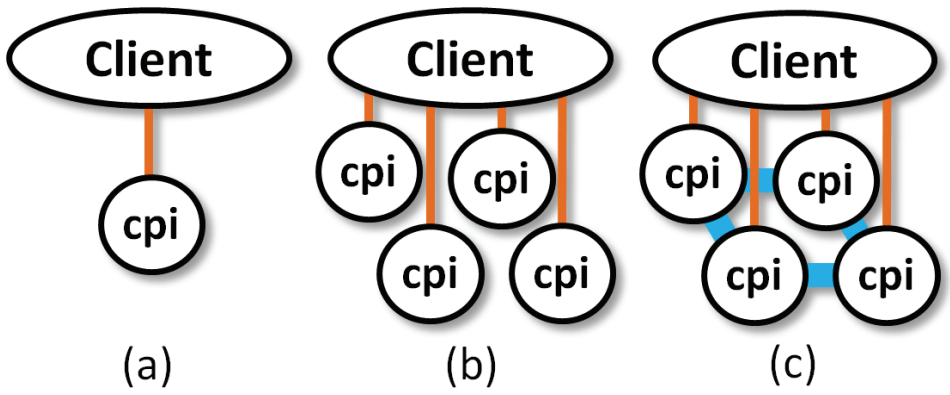


Figure 1.1 A client program may spawn (a) a single ChemPlugin instance, (b) an arbitrary number of ChemPlugin instances, or (c) a number of ChemPlugin instances that self-link into any configuration.

Once created, the ChemPlugin instances self-link into a network that represents the system being modeled (**Figure 1.2**). The client program at a minimum specifies the rate at which fluid flows across each link. The client can also set at each link transmissivities representing diffusion, physical mixing during flow, and heat conduction.

Then, as it marches forward in time, the client prompts each instance to perform essential steps: reporting the optimum time step size, transporting mass, transferring heat, and solving the equations describing chemical reaction. Notably, each such step is accomplished in the client program with a single line of code. Instead of performing the calculations itself, the client simply triggers the instances to do so.

1.2 Advantages

ChemPlugin is designed to save you time and money. In programming multicomponent reaction into a flow model, whether within an existing code or an application under development, ChemPlugin offers compelling advantages to hand coding, or using less-capable software objects:

- **Self-linking**. ChemPlugin objects are self-linking and hence object instances can organize and re organize themselves instantly into any desired geometry.
- **Transport**. Once self-linked, the object instances handle mass and heat transport among themselves, eliminating most of the programming overhead required to implement multicomponent chemistry within a flow model.
- **Memory management**. Cutting edge memory management allows a >100,000 ChemPlugin instances to run together on a simple laptop, and many more on a workstation or cluster.
- **Multithreading**. ChemPlugin objects are thread-safe and ready for parallel deployment. For $>$10,000 instances, tests show parallel speedups on a four-core hyperthreaded processor of about $\times$3.7 to $\times$4.
- **Cluster computing**. An MPI version of ChemPlugin is available for parallel implementation on computing clusters. You may as well multithread a cluster program to create a hybrid parallel client.
- **Rapid development**. ChemPlugin consists of 70,000 lines of code pre-packaged as an object. This is code your team need not write, debug, test, perfect, validate, and document from scratch.
- **Ease of coding**. ChemPlugin encapsulates the technical details, so your team can build sophisticated applications without specific expertise in chemical modeling.
- **Completeness**. Coding all the myriad aspects users demand of a chemical modeling code into your application is a daunting task, but with ChemPlugin your app arrives full-featured.

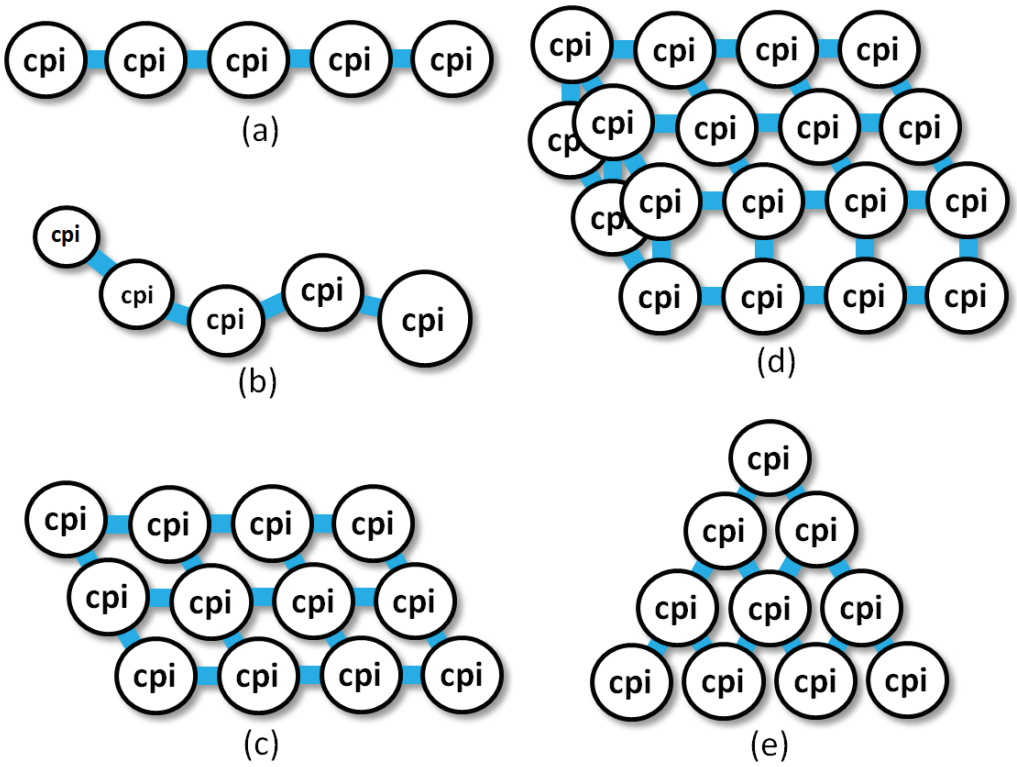


Figure 1.2 Once spawned, ChemPlugin instances can self-link into virtually any geometry. The examples shown include (a) a one-dimensional domain, (b) a curvilinear chain, (c) a two-dimensional domain, (d) a three-dimensional domain, and (e) a bifurcating tree.

- **Code fingerprints**. ChemPlugin instances are controlled by a sleek API that makes incorporating them into a client program a snap. The API's light fingerprints in your source code minimize development and support effort.
- **Memory footprint**. The baseline memory footprints per ChemPlugin instance are just 50 kilobytes (32 bit) and 65 kB (64 bit).
- **Reliability**. ChemPlugin objects are derived directly from The Geochemist's Workbench® package, which is trusted worldwide and used at thousands of installations in 97 countries.
- **Trust**. The objects are subject to the same quality control program, including daily automated testing, as the GWB.

- **Versatility**. ChemPlugin objects can solve the same broad gamut of multicomponent reaction problems as the GWB software package.
- **Flexibility**. ChemPlugin comes with "thin wrappers" that make let appear as a native C++, FORTRAN, or Python object. A single license serves all of these languages.
- **No retraining**. The broadly known interactive scripting employed by the GWB configures ChemPlugin instances; hence, no retraining for engineers and scientists is needed.
- **Common user interface**. Users see a common interface to the geochemical modeling aspects of all of an organization's codes, reducing training and increasing productivity and responsiveness.
- **Painless replication**. Once implemented within one of an organization's codes, ChemPlugin's capabilities can be readily transferred to other codes using the knowledge and experience acquired in the initial deployment.
- **Thermodynamic datasets**. ChemPlugin uses the open-format GWB thermo datasets available for a variety of purposes from sources worldwide; the datasets can be quickly manipulated with the TEdit application, reducing users' time-to-solution.
- **Textbook**. A clearly written, tutorial-based textbook carries the reader through a series of specific examples that show how to set up progressively more powerful client programs.
- **Reference Manual**. The thorough but concise Reference Guide is organized, accurate, and helpful.
- **Support**. ChemPlugin is professionally supported by Aqueous Solutions LLC, maker of the GWB software; there is no need to train and deploy expensive support staff in-house.

1.3 Languages supported

The ChemPlugin software is supplied with a number of "thin wrappers" that provide for its use in client programs written in a range of languages. The object itself is written is C++, but by employing the corresponding wrapper, it can appear to a FORTRAN client program to be written in FORTRAN, or to a Python client as a Python object. A single copy of ChemPlugin, then, serves for a variety of uses — there is no need to license additional versions of the software, just because the language you are using changes.

ChemPlugin is currently distributed with wrappers for:

- C++

- FORTRAN
- Python

In the tutorials in this User's Guide, we will work with client programs written in C++. Significantly, however, a version of the client from each tutorial in any of the languages above can be downloaded from the ChemPlugin.GWB.com website.

As well, you can download from this website multithreaded, cluster computing, and hybrid multithreaded-cluster clients written in C++ and Fortran; Python does not natively support parallel programming.

Overview

In this chapter, we look at a few specifics of how ChemPlugin instances can be deployed within a client program. We conclude the chapter by writing a simple client program that drives a single ChemPlugin instance. Later chapters build on this example, gradually introducing new ways to take advantage of the power of the ChemPlugin object.

2.1 Creating and destroying instances

Creating a ChemPlugin instance is a simple matter of declaring it. For example, the statement

```
ChemPlugin cp;
```

creates a reference "cp" to a ChemPlugin instance. When "cp" is created, it spawns the instance to which it refers. Hence, bringing a variable of type ChemPlugin into scope creates a ChemPlugin instance and sets a reference to it.

A client can similarly set a vector of ChemPlugin instances:

```
ChemPlugin cp[100];
    or
ChemPlugin* cp = new ChemPlugin[100];
    or
vector <ChemPlugin> cp;
```

In these cases, each element in the vector references a separate ChemPlugin instance: "cp[0]" is the first instance, "cp[1]" is the second, and so on.

2.1.1 Deleting instances

In computer languages that feature "garbage collection," you may decide to delete ChemPlugin instances when you are done with them to immediately free memory for other uses, but there is little compelling need to do so. In languages such as C++, however, it is best practice to delete instances once they are no longer needed to avoid the possibility of memory leaks. In either case, a client removes a ChemPlugin instance from memory simply by deleting the reference to it.

When a client sets a reference to a ChemPlugin instance as an automatic variable, the instance is created when the reference comes into scope, and deleted when it goes out of scope. When a client executes a statement

```
ChemPlugin cp;
```

bringing a reference "cp" into scope, a ChemPlugin instance is created. Once "cp" goes out of scope, both the reference and instance itself are deleted automatically.

When a client allocates a reference explicitly, however, it should delete it explicitly, as well. For example, in the code

```
ChemPlugin* cp = new ChemPlugin;
... some code goes here ...
delete cp;
```

we allocate a ChemPlugin reference, then delete it and the associated instance when we are finished using it. Similarly, we might allocate a vector of ChemPlugin instances

```
int nx = 100;
ChemPlugin* cp = new ChemPlugin[nx];
... some code goes here ...
delete[ ] cp;
```

and delete them after their purpose has been served.

2.1.2 Console messages

ChemPlugin instances run silently by default, but can be set to write console messages that trace the instance's actions and report any errors encountered.

To set an instance to begin writing messages as it starts up, a client specifies an output stream as an argument at instantiation time. The outlet stream can be standard output, standard error, or a dataset:

```
ChemPlugin cp("stdout");
    or
ChemPlugin cp("stderr");
    or
ChemPlugin cp("MyMessages.txt");
```

In these cases, console output is directed to the standard output, standard error, or a text dataset.

Significantly, console messages can be turned on or off, or redirected, at any point in the execution of a client program. The procedure for doing so is described in the **Controlling instances** section.

2.1.3 Option flags

You can set option flags in an optional second argument to the constructor, following the optional argument for specifying the output stream. For example:

```
ChemPlugin cp("stdout", "-d mythermo.tdat -s mysurface.sdat");
    or
ChemPlugin cp("", "-d mythermo.tdat");
    or
ChemPlugin cp(NULL, "-d mythermo.tdat");
```

The following option flags are available:

`-cd`	Change the working directory to the directory containing the input script specified with the `-i` option.
`-i` <input_script>	Read initial input commands from the specified file.
`-gtd` <gtdata_dir>	Set directory to search for thermodynamic datasets.
`-cond` <cond_data>	Set the dataset for calculating electrical conductivity.
`-d` <thermo_data>	Set the thermodynamic dataset.
`-s` <surf_data>	Set a dataset of surface sorption reactions.

2.1.4 Environmental variables

You can specify various default settings for ChemPlugin by defining environment variables. In a command line environment, you might, for example, issue the command

```
set CPI_THERMODATA=my_thermo.tdat
```

which would define “my_thermo.tdat” as the default thermodynamic dataset that loads whenever a ChemPlugin object initializes. You set environmental variables globally from the Windows Control Panel, under **System** → **Advanced system settings**.

You can set the following environment variables:

`CPI_GTDATA`	The directory where the apps will look for thermo datasets, if not found in the working directory.
`CPI_THERMODATA`	The default thermodynamic dataset.
`CPI_CONDUCTIVITYDATA`	The dataset of coefficients for calculating electrical conductivity.
`CPI_ISOTOPEDATA`	The dataset of isotope fractionation factors.
`CPI_SURFACEDATA`	Dataset(s) of surface sorption reactions.

2.2 Controlling instances

A client program uses *member functions* to control a ChemPlugin instance. For example, to set pH at an instance to 5, a client program might use the “Config()” member function to pass the configuration command “pH = 5”:

```
ChemPlugin cp
cp.Config("pH = 5");
```

Similarly, member function "Initialize()"

```
cp.Initialize();
```

solves for the initial state of an instance, given its configuration.

Note the form of a member function call: the instance in question, the name of the member function, and any arguments. Suppose a client needs to set pH at two instances to different values:

```
ChemPlugin cp0, cp1;
cp0.Config("pH = 5");
cp1.Config("pH = 9");
```

Now, the fluid at "cp0" is slightly acidic, whereas at "cp1" it is slightly alkaline.

The remainder of this section provides an overview of the ChemPlugin member functions. The **Member Functions** appendix to this User's Guide provides a complete list of the member functions and a full description of how each is used.

2.2.1 Configuring and initializing an instance

As already noted, a client uses the "Config()" member function to configure a ChemPlugin instance, and "Initialize()" to initialize it. Function "Config()" takes a *configuration command* as an argument. ChemPlugin's configuration commands are very similar to those used in The Geochemist's Workbench by program **React**, so if you are familiar with the GWB already, there is little to learn. The **Configuration Commands** appendix to this User's Guide provides a complete reference.

A client can pass several commands with a single call if it separates the commands with semicolons. The statements

```
ChemPlugin cp;
cp.Config("Na+ = 2 mmol/kg; Cl- = 2 mmol/kg; pH = 6");
cp.Initialize();
```

configure instance "cp" to contain a dilute, slightly acidic salt solution, and then initializes the instance by calculating the distribution of mass among the various chemical species: Na^+, Cl^-, NaCl(aq), H^+, OH^-, HCl, and so on.

2.2.2 Linking instances

A client uses the "Link()" member function to link two ChemPlugin instances. For example,

```
ChemPlugin cp0, cp1;
cp0.Link(cp1);
```

links “cp0” to “cp1”. The link is reciprocal, so the client should not then link “cp1” to “cp0”, except to create a second link between the instances. Calling link without an argument

```
cp0.Link();
```

creates a free outlet. The **Linking Instances** chapter in this User’s Guide provides complete details and several examples of linking ChemPlugin instances.

Once a link is in place, a client uses the “FlowRate()” member function to set the rate at which fluid traverses it:

```
ChemPlugin cp0, cp1;
CpiLink link0 = cp0.Link(cp1);
link0.FlowRate(10., "m3/day");
```

Function “FlowRate()”, as we can see, is a member of the CpiLink class, because it applies to a link, rather than a ChemPlugin instance.

Flow is by convention positive when it moves toward the originating ChemPlugin instance. In the example above, then, fluid moves from “cp1” toward “cp0”. The **Flow and Transport** chapter gives specifics on setting flow across links.

A client can also set transmissivities that represent mass transport by diffusion and physical mixing (i.e., hydrodynamic dispersion and turbulent mixing) using the “Transmissivity()” member function. Function “HeatTrans()” defines in a similar fashion heat conduction from one instance to another. Setting the mass transmissivity is described in detail in the **Diffusion and Dispersion** chapter, and the **Heat Transfer** chapter shows how to set the thermal transmissivity across a link.

2.2.3 Time marching

A set of five member functions provide for time marching, once an instance has been initialized. They are:

- “ReportTimeStep()” returns the optimum time step length;
- “AdvanceTimeStep()” moves forward the time level, adds or removes simple reactants, and adjusts sliding buffers;
- “AdvanceTransport()” computes the effects of mass transport by advection, diffusion, and mixing;
- “AdvanceHeatTransport()” calculates the movement of heat by advection and conduction; and
- “AdvanceChemical()” evaluates the equations describing chemical reaction.

These functions, except the first, return a non-zero value when they encounter the end of a simulation, or fail for any reason.

A loop for marching a single ChemPlugin instance forward in time might be coded:

```
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceTransport()) break;
   if (cp.AdvanceHeatTransport()) break;
   if (cp.AdvanceChemical()) break;
}
```

The actual form of the loop differs, of course, depending on the application. Where heat transfer is not considered, the call to "AdvanceHeatTransport()" would be omitted. In a program that spawned multiple ChemPlugin instances, the client would loop over the instances for each of the calls. And in a flow model that already contains a time marching loop, the member function calls would be inserted within the existing loop.

2.2.4 Console messages

Console output contains routine messages tracing the actions of an instance, as well as any warning and error messages generated. Unless enabled at instantiation, as described above, a ChemPlugin instance produces no console output until directed to do so with a call to the "Console()" member function.

"Console()" takes the target for output as its argument. The call

```
cp.Console("stdout");
```

directs console messages to the standard output, whereas

```
cp.Console("MyMessages.txt");
```

sends them to a text file for later inspection. A call with "NULL" as an argument, or no argument at all

```
cp.Console();
```

disables console output. For more information, please reference the **Member Functions** appendix to this guide.

2.2.5 Retrieving results

A client program retrieves individual pieces of information from a ChemPlugin instance with the "Report1()" member function, and arrays of data with "Report()". To query instance "cp" for its pH, alkalinity, and H^+ ion concentation, for example, a client could call:

```
double pH = cp.Report1("pH");
double alk = cp.Report1("alkalinity", "meq_acid/kg");
double conc = cp.Report1("concentration H+", "mmol/kg");
```

The statements

```
double* conc = new double[naqueous];
cp.Report(conc, "concentration aqueous", "mmol/kg");
```

retrieve a vector of the concentrations of the various aqueous species considered by "cp", and store it in "conc".

To retrieve an individual integer or character string, a client uses the "Report1i()" or "Report1c()" member functions, rather than "Report1()". The **Retrieving Results** chapter in this User's Guide provides more information about the "Report()" family of functions.

2.2.6 Output streams

A ChemPlugin instance can write out its calculation results in two ways. Print-format output consists of calculation results formatted to be readable by humans. The output is produced in blocks periodically as a simulation marches forward in time. Calling the "PrintOutput()" member function triggers an instance to write a block of output.

Plot-format output is a data stream designed to be read by the **Gtplot** application distributed with The Geochemist's Workbench. Plot datasets consist of a header, one or more blocks describing the chemical system at given points in a simulation, and a trailer. A client can create plot datasets by calling the "PlotHeader()", "PlotBlock()", and "PlotTrailer()" member functions.

The **Direct Output** chapter in this User's Guide describes how to generate print-format and plot-format output datasets.

2.3 Example program

The use of ChemPlugin is perhaps best demonstrated by writing a simple client program that spawns a single ChemPlugin instance. Our program "Simple.cpp" serves to predict how pH changes as NaOH is titrated into a NaCl solution of pH 3. The code is:

```
#include <iostream>
#include "ChemPlugin.h"

int main(int argc, char** argv)
{
   // Create and configure a ChemPlugin instance.
   ChemPlugin cp;
   cp.Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg; pH = 3");
   cp.Config("react 2 mmol NaOH; delxi = 0.1");
```

```
    // Calculate initial conditions.
    cp.Initialize();
    std::cout << "pH = " << cp.Report1("pH") << std::endl;

    // March forward in time.
    while (true) {
        double deltat = cp.ReportTimeStep();
        if (cp.AdvanceTimeStep(deltat)) break;
        if (cp.AdvanceChemical()) break;
        std::cout << "pH = " << cp.Report1("pH") << std::endl;
    }

    // Any keystroke closes the console.
    std::cin.get();
    return 0;
}
```

At the beginning of the program we've included the header file "ChemPlugin.h", which is installed with the ChemPlugin software. This file allows the compiler to recognize, among other things, ChemPlugin's member functions.

The first three lines of the program create and configure a ChemPlugin instance named "cp". A call to "Config()" sets the initial conditions by stacking three configuration commands, separated by semicolons. A second call sets up the NaOH titration, to be accomplished in ten steps, since delxi is 0.1. The next two lines use member function "Initialize()" to trigger "cp" to calculate the initial conditions, and function "Report1()" to retrieve from "cp" the initial pH, which the program reports to the console window.

Finally, the program enters a time marching loop, cycling over member functions "ReportTimeStep()", "AdvanceTimeStep()", and "AdvanceChemical()". At each pass through the loop, the program again uses "Report1()" to find and report the current pH. When "AdvanceTimeStep()" finds the time marching loop is complete, at the point when the NaOH titrant is exhausted, it returns a non-zero value, breaking the loop. If "AdvanceChemical()" were to be unable to complete its calculations for some reason, it would similarly return true and break the time marching.

Running client program "Simple.cpp" writes the output

```
pH = 3
pH = 3.092
pH = 3.20886
pH = 3.36925
pH = 3.6263
pH = 4.34164
pH = 10.1464
pH = 10.5096
pH = 10.7039
```

```
pH = 10.8373
pH = 10.9388
```

to the console window.

The source code to “Simple.cpp”, as well as other client programs developed in this User’s Guide, can be downloaded from the ChemPlugin.GWB.com website. The procedure for compiling and linking a client program into an executable application is described in the **ChemPlugin Setup** appendix.

2.4 Using this Guide

This User’s Guide consists of a series of chapters, each of which describes an aspect of using the ChemPlugin self-linking software object. As the chapters unfold, we develop progressively more detailed and interesting client programs, each of which serves as an example of how to use ChemPlugin objects in a certain manner. The final example is a full-featured polythermal reactive transport code. We suggest you progress from client to client, running and experimenting with each, before beginning your own project.

Titration Simulator

As a first example of using ChemPlugin, we set out to write a client program that traces pH titrations. Our client program works by spawning and configuring a single ChemPlugin instance. The client then enters a time marching loop that carries out the titration.

3.1 Program structure

Our client program is laid out as a console program. The general structure of the program file is as follows:

```
#include <iostream>
#include <iomanip>
#include "ChemPlugin.h"

int main(int argc, char** argv) {
    ... client program goes here ...
}
```

The first three lines import C++ system header files, as well as the header "ChemPlugin.h", which is installed with the ChemPlugin software. It's a good idea to point your compiler to the copy of "ChemPlugin.h" in the installation directory, rather than a local copy of the file, to make sure it pulls in the latest installed version. The client program itself follows the header lines.

3.2 Client program

The client program has the form:

```
int main(int argc, char** argv) {
    std::cout << "ChemPlugin example -- pH titration" << std::endl << std::endl;
    std::cout << std::fixed << std::setprecision(2);

    // Create a ChemPlugin instance.
    ChemPlugin cp("stdout");
```

```
    // Configure the instance.
    ... configuration step goes here ...

    // Initialize the instance.
    ... initialization step goes here ...

    // Time marching loop.
    ... time marching loop goes here ...

    // Any keystroke closes the console.
    std::cin.get();
    return 0;
}
```

After identifying itself and setting a format for floating point output, the program creates an instance "cp" of type "ChemPlugin"—"cp" is the ChemPlugin instance embedded in the client. As it instantiates "cp", the program instructs the instance to direct its console messages to "stdout", the standard output stream. Console messages consist of information an instance writes as it progresses through a calculation, as well as any error messages it may generate.

The client then configures the instance, initializes it, and enters a time marching loop that traces the titration. These three steps are described in the following subsections. When the time marching loop completes, the client waits for a keystroke from the user and closes the console. The ChemPlugin instance is an automatic variable, so its memory is freed when function "main()" goes out of scope.

3.2.1 Configuration step

To configure "cp", the client program uses member function "Config()" to send the instance a series of commands that define the initial condition, as well as the titration to be undertaken.

```
    // Configure the instance.
    cp.Config("Ca++ = 1 mmol/kg; Na+ = 1 mmol/kg");
    cp.Config("Cl- = 3 mmol/kg; HCO3- = 2 mmol/kg; pH = 4");
    cp.Config("react 3 mmol/kg NaOH; delxi = 0.1");
```

The first two lines set a Ca^{2+}-Na^{+}-Cl^{-}-HCO_3^{-} solution of pH 4. The third line specifies that 3 mmol/kg of NaOH are to be titrated into the fluid, which by default has a solvent mass of 1 kg.

The third line further specifies a reaction step $\Delta\xi$ of 0.1. Reaction progress ξ varies in a simulation from zero at the outset to a final value of one. $\Delta\xi$ is set to 0.1, so the titration will proceed in 10 steps.

Note that several commands can be passed in a single "Config()" call, if they are separated by semicolons. The **Configuration Commands** appendix of this User's Guide describes ChemPlugin's command set.

3.2.2 Initialization step

Once the instance is configured, a call to member function “Initialize()” triggers it to compute the initial state corresponding to its configuration, and to prepare for time marching.

```
// Initialize the instance.
cp.Initialize();
std::cout << " Xi = " << cp.Report1("Xi");
std::cout << " pH = " << cp.Report1("pH") << std::endl;
```

The client then uses member function “Report1()” to query the ChemPlugin instance for the reaction progress variable ξ and the initial pH, the values of which it writes to the console.

3.2.3 Time marching loop

The time marching loop by which the client program traces the titration consists of only a few lines of code:

```
// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   std::cout << " Xi = " << cp.Report1("Xi");
   std::cout << " pH = " << cp.Report1("pH") << std::endl;
}
```

The loop calls three member functions, and executes two lines in which the client fetches and reports ξ and pH, in a cycle.

First, a call to “ReportTimeStep()” triggers the instance to calculate an appropriate time step size. The function returns a value for Δt in units of seconds, and the program stores the value in variable “deltat”. We haven’t set a time span for the simulation, so reflecting the value set for “delxi” at configuration time, “deltat” is one-tenth of an arbitrary end time of 1 day carried by ChemPlugin.

Next, a call to “AdvanceTimeStep()” passes “deltat” to the instance. Upon executing the function, the instance moves forward in reaction progress by “deltat” seconds. The function returns a value of zero, unless the program has reached the end of the simulation. In that case, a non-zero return breaks the loop and time marching ceases.

Finally, executing member function “AdvanceChemical()” causes the instance to evaluate the chemical equations at the new point in time, accounting for equilibrium as well as kinetic reactions. The function returns zero, unless an error occurs. A non-zero return marks an error, again breaking the loop.

Once the time step is complete, the client writes out ξ and pH, and returns to take another step.

3.3 Running the example program

Upon execution, the client program produces the output shown below:

```
ChemPlugin example -- pH titration


Solving for initial system.

 Loaded:   17 aqueous species,
           16 minerals,
            2 gases,
            0 surface species,
            6 elements,
            3 oxides.

 Xi = 0.00 pH = 4.00
 Xi = 0.10 pH = 5.39
 Xi = 0.20 pH = 5.86
 Xi = 0.30 pH = 6.15
 Xi = 0.40 pH = 6.42
 Xi = 0.50 pH = 6.70
 Xi = 0.60 pH = 7.08
 2 supersaturated phases, most = Calcite
 Swapping Calcite in for CO2(aq)
 Xi = 0.70 pH = 7.68
 Xi = 0.80 pH = 7.87
 Xi = 0.90 pH = 8.16
 Xi = 1.00 pH = 8.72

Successful completion of reaction path.
```

The output consists of console messages from the instance interspersed with lines written by the client; the latter output starts with "Xi = ...". Note especially the message reporting that $CaCO_3$ precipitates as the solution trends alkaline.

We might prefer to not intersperse console and client messages in the output stream. In that case, we can call member function "Console()" without an argument

```
// Time marching loop.
cp.Console();
while (true) {
   double deltat = cp.ReportTimeStep();
   ... and so on ...
}
```

at the head of the time marching loop. A call of this form disables console output from the instance.

3.4 Assembled C++ code

The source code for the client program, as assembled from above, is shown below:

```
#include <iostream>
#include <iomanip>
#include "ChemPlugin.h"


int main(int argc, char** argv) {
   std::cout << "ChemPlugin example -- pH titration" << std::endl << std::endl;
   std::cout << std::fixed << std::setprecision(2);

   // Create a ChemPlugin instance.
   ChemPlugin cp("stdout");

   // Configure the instance.
   cp.Config("Ca++ = 1 mmol/kg; Na+ = 1 mmol/kg");
   cp.Config("Cl- = 3 mmol/kg; HCO3- = 2 mmol/kg; pH = 4");
   cp.Config("react 3 mmol/kg NaOH; delxi = 0.1");

   // Initialize the instance.
   cp.Initialize();
   std::cout << " Xi = " << cp.Report1("Xi");
   std::cout << " pH = " << cp.Report1("pH") << std::endl;

   // Time marching loop.
   while (true) {
      double deltat = cp.ReportTimeStep();
      if (cp.AdvanceTimeStep(deltat)) break;
      if (cp.AdvanceChemical()) break;
      std::cout << " Xi = " << cp.Report1("Xi");
      std::cout << " pH = " << cp.Report1("pH") << std::endl;
   }

   // Any keystroke closes the console.
   std::cin.get();
   return 0;
}
```

The source code may be downloaded as file “Titration1.cpp”, available from the ChemPlugin.GWB.com website.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

3.5 Generalization

In closing, we note the code above can be adapted to trace reaction paths of arbitrary nature, simply by altering the configuration step. In the **React Emulator** chapter we do just that, setting up a reactor model that employs a ChemPlugin instance to trace reaction models in the general sense.

Retrieving Results

The client program we developed in the previous chapter spawned a ChemPlugin instance and used it to trace a titration. At each step in the time marching, the client used the “Report1()” member function to query the instance for the current point in the reaction progress, and again to retrieve the pH.

In this chapter, we consider in more detail how a client can retrieve calculation results from a ChemPlugin instance, by calling the “Report()” member function, or its short forms “Report1()”, “Report1i()”, and “Report1c()”, which return a single floating value, integer number, or character string, respectively. The client might use the results in its own calculations, or just write them out for the user.

It is also possible to direct an instance to write calculation results directly to files, for later viewing or plotting. The **Direct Output** chapter of this User’s Guide describes how ChemPlugin instances can generate output in this manner.

4.1 Report() family of member functions

A client program uses the “Report()” member function, or its cousins “Report1()”, “Report1i()”, and “Report1c()”, to gather information about the current state of a ChemPlugin instance. A call to “Report()” has the form:

```
ChemPlugin cp;
void *target;
const char* keywords, unit;
int n = cp.Report(target, keywords, unit);
```

The function locates the information corresponding to “keywords”, casts it in terms of “unit”, if this field is supplied, and copies the data as a vector to memory location “target”. The function returns the number of pieces of information that have been written to “target”.

The related function “Report1()” returns a single double value; “Report1()” is a short form of the “Report()” function

```
double s = cp.Report1(keywords, unit);
```

in which the result is returned directly, rather than copied to a target in memory. The statement above is functionally equivalent to:

```
double s;
cp.Report(&s, keywords, unit);
```

Functions "Report1i()" and "Report1c()" work similarly,

```
int i = cp.Report1i(keywords);
char* c = cp.Report1c(keywords);
```

except they return an integer or pointer to a character string, respectively.

For the "Report()" family of functions, the options for specifying "keywords" are listed in the **Report Function** chapter of this User's Guide. The "unit" keyword is optional, and the choices available are shown in the **Units Recognized** appendix to this Guide.

"Report()" may not be able to fulfill every request—perhaps the client has specified an impossible unit conversion—in which case the function will write to "target" the marker value "ANULL", which is defined in "ChemPlugin.h".

4.1.1 Scalar values

To retrieve an individual value of type double, such as pH or a given species' concentration, a client can use "Report1()", the short form of the "Report()" function. A client program might need to retrieve from a ChemPlugin instance "cp" the current pH, the alkalinity in meq of acid per kg solution, and the concentration of the H^+ ion in mmol/kg, and to write out the values. In this case, you can code

```
double pH     = cp.Report1("pH");
double alk    = cp.Report1("alkalinity", "meq_acid/kg");
double chplus = cp.Report1("concentration H+", "mmol/kg");

cout << "pH = " << pH
     << ", alkalinity = " << alk << " meq/kg acid"
     << ", conc. H+ = " << chplus << " mmol/kg" << endl;
```

within the client program. Alternatively, of course, a client can write values directly

```
cout << "pH = " << cp.Report1("pH") << endl;
```

without storing them.

Member functions "Report1i()" and "Report1c()" work the same way, but query an instance for integer values and character string pointers. The statements

```
int nsp = cp.Report1i("naqueous");
char* spec = cp.Report1c("species 4");
```

store the number of aqueous species considered in “nsp”, and the name of the fifth aqueous species in “spec” (vector positions are numbered by offset, so the fifth species is indexed 4).

4.1.2 Vector quantities

To retrieve vectors of data, such as the names and concentrations of the aqueous species considered by a ChemPlugin instance, a client allocates target memory to hold the results and pass the target’s address to “Report()”.

Suppose a client program requires the names and free concentrations in mmol kg^{-1} of the various aqueous species considered. In this case, you could include:

```
// Find number of aqueous species.
int nsp = cp.Report1i("naqueous");

// Get species names.
char **spec = new char*[nsp];
cp.Report(spec, "species");

// Retrieve species concentrations.
double *conc = new double[nsp];
cp.Report(conc, "concentration aqueous", "mmol/kg");

// Output results.
for (int i; i<nsp; i++)
   cout << spec[i] << " = " << conc[i] << " mmol/kg" << endl;

// Free up memory.
delete[ ] spec;
delete[ ] conc;
```

within the client.

Rather than gathering a complete vector, a client can retrieve vector elements by index. The following code

```
// Find number of aqueous species.
int nsp = cp.Report1i("naqueous");

for (int i; i<nsp; i++) {
   std::string spec_keywords = "species " + std::to_string(i);
   std::string conc_keywords = "concentration aqueous " + std::to_string(i);

   // Find and output species names and concentrations.
   char *spec = cp.Report1c(spec_keywords);
   double conc = cp.Report1(conc_keywords, "mmol/kg");
```

```
    cout << spec << " = " << conc << " mmol/kg" << endl;
}
```

serves the same purpose as the previous example.

Finally, a client can retrieve elements of a vector by name, as shown in this example:

```
// Get names of aqueous species.
int nsp = cp.Report1i("naqueous");
char **spec = new char*[nsp];
cp.Report(spec, "species");

for (int i; i<nsp; i++) {
    // Output concentration of each species.
    std::string conc_keywords = "concentration aqueous " + std::string(spec[i]);
    double conc = cp.Report1(conc_keywords, "mmol/kg");
    cout << spec[i] << " = " << conc << " mmol/kg" << endl;
}
```

Again, the results are the same as the previous examples in this section.

4.1.3 NULL target

If the client passes NULL for "target", the "Report()" function determines the number of pieces of information to be returned, without copying information to the client program. This feature can help avoid memory overwrites.

In the example above, we might replace the sequence

```
int nsp = cp.Report1i("naqueous");
```

with

```
int nsp = cp.Report(NULL, "concentration aqueous", "mmol/kg");
```

Now, we can be more directly certain that

```
double *conc = new double[nsp];
cp.Report(conc, "concentration aqueous", "mmol/kg");
```

will not overwrite memory, since we have let "Report()" determine the number of values a specific call to the function will copy to "conc".

4.2 An example

In the previous chapter, we developed a program for reporting how pH varies over the course of a titration. Suppose as an alternative example we would like the client

to report not pH, but the concentration of each aqueous species present at levels of 10 μmol/kg or greater.

To this end, we replace the time marching loop in the original client program with the code:

```
// Initialize the instance.
cp.Initialize();

int nsp = cp.Report(NULL, "species");
char **spec = new char*[nsp];
double *conc = new double[nsp];
cp.Report(spec, "species");

// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   cp.Report(conc, "concentration aqueous", "umol/kg");
   for (int i=0; i<nsp; i++)
      if (conc[i] >= 10.0)
         std::cout << spec[i] << " = " << conc[i] << " umol/kg" << std::endl;
   std::cout << std::endl;
}

delete[ ] spec;
delete[ ] conc;
```

The strategy is to gather pointers to the species' names and store the vector in "spec", and to write the species' concentrations to vector "conc".

At the onset of time marching, the client program determines the number of aqueous species "nsp" and, knowing that, allocates vectors "spec" and "conc". It then calls "Report()" to fill "spec" with pointers to the species' names. Note the species' names are available only after "Initialize()" has been called.

Upon completing each time step, the client queries the ChemPlugin instance for the species' concentrations. For each species at the requisite level, the program writes out its name and concentration. Once the time marching is complete, the client frees the memory allocated to "spec" and "conc".

Console output from running the revised code is:

```
ChemPlugin example -- pH titration

Solving for initial system.

 Loaded:   17 aqueous species,
           16 minerals,
            2 gases,
```

```
            0 surface species,
            6 elements,
            3 oxides.

CO2(aq) = 1792.81 umol/kg
Ca++ = 985.82 umol/kg
CaCl+ = 11.61 umol/kg
Cl- = 3085.40 umol/kg
HCO3- = 204.29 umol/kg
Na+ = 1299.59 umol/kg

CO2(aq) = 1495.66 umol/kg
Ca++ = 982.33 umol/kg
CaCl+ = 11.48 umol/kg
Cl- = 3085.48 umol/kg
HCO3- = 497.17 umol/kg
Na+ = 1598.93 umol/kg

 ... and so on ...
```

As an alternative, we can code the time marching loop to retrieve species' names and concentrations individually, rather than in vector form:

```
// Initialize the instance.
cp.Initialize();

int nsp = cp.Report1i("naqueous");

// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;

   for (int i=0; i<nsp; i++) {
      std::string spec_keywords = "species " + std::to_string(i);
      std::string conc_keywords = "concentration aqueous " + std::to_string(i);

      char *spec = cp.Report1c(spec_keywords);
      double conc = cp.Report1(conc_keywords, "umol/kg");
      if (conc > 10.0)
         std::cout << spec << " = " << conc << " umol/kg" << std::endl;
   }
   std::cout << std::endl;
}
```

The scalar coding gives the same results as the vector coding, but requires additional calls to “Report1()” and “Report1c()”, and hence might be expected to execute less quickly.

4.3 Source code

The complete source code for the client programs developed in this chapter are contained in files “Titration2.cpp” and “Titration3.cpp”, available for download from the ChemPlugin.GWB.com website.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

Direct Output

A client program, as we showed in the previous chapter, can retrieve calculation results from a ChemPlugin instance and write out those results for later viewing or plotting. This scenario is perhaps the most common mode of rendering the results of ChemPlugin simulations.

The client program can also prompt a ChemPlugin instance to write out results directly. ChemPlugin can do this in two ways:

- A print-format dataset. Such datasets are composed of data blocks within a simple text file. Each block represents the state of a ChemPlugin instance at a specific point in a simulation, set out in terms of human-readable tables.
- A plot-format dataset. Plot-format datasets are meant to be read by the **Gtplot** program supplied with the GWB. Plot datasets provide for a simple method for graphically rendering the results of a ChemPlugin simulation for an individual ChemPlugin instance.

Direct output, then, provides a method for communicating results to the software user with a minimum of effort on the part of the client.

5.1 Scheduling output

There are two ways to set a ChemPlugin instance to write out results directly:

- Self-scheduled output. In this case, the client program uses the “Config()” member function to turn on print output, plot output, or both. Optionally, the client may call “Config()” to set variables controlling the frequency of output.
- On-demand output. Here, the client program calls a member function whenever it wants the instance to write print- or plot-format output.

The two options are described in detail in the following sections.

5.2 Self-scheduled output

A client can allow a ChemPlugin instance to schedule print-format and plot-format output on its own, just as the GWB program **React** does.

5.2.1 Print output

A client turns on self-scheduled output to a print-format dataset with the "print" configuration command. The call

```
cp.Config("print = on");
```

causes print output to be scheduled.

A ChemPlugin instance, when using self scheduling, writes output blocks after initializing the system, at set intervals in reaction progress, as well as whenever the phase assemblage in the chemical system changes. The "dxprint" configuration command controls the interval between output points. For example, the call

```
cp.Config("dxprint = 0.1");
```

sets the instance to write an output block at the onset of the simulation, ten times over the course of the simulation, at ξ = 0, 0.1, 0.2, and so on, and whenever the phase assemblage changes.

5.2.2 Plot output

A client turns on self-scheduled output to the plot dataset with the "plot" command:

```
cp.Config("plot = on");
```

The "dxplot" configuration command

```
cp.Config("dxplot = 0.001");
```

sets the minimum spacing in reaction progress between plot output points, in this case to one-thousandth of the reaction interval. The command

```
cp.Config("dxplot = 0");
```

specifies that each step in the reaction simulation be represented in the plot dataset.

5.3 On-demand output

A client program can trigger output to a print-format dataset using the "PrintOutput()" member function, and to a plot-format dataset with functions "PlotHeader()", "PlotBlock()", and "PlotTrailer()".

5.3.1 Print output

A call to member function “PrintOutput()” triggers a ChemPlugin instance to write a block of data representing the instance’s current state to a print-format dataset. When passed a file name

```
cp.PrintOutput("myPrint.txt");
```

the instance appends a data block to that file, if it is open for output. If not, the instance opens and writes to a new file of that name.

Calling the function without an argument

```
cp.PrintOutput();
```

causes the block to be appended to whatever dataset is currently open for print output. If none is open, the instance opens and writes to a new dataset “ChemPlugin_output.txt”.

In either case, the instance will append any output suffix that may have been set to the file name. For example, the statements

```
cp.Config("suffix _mysuffix");
cp.PrintOutput("myPrint.txt");
```

directs output to a dataset named “myPrint_mysuffix.txt”.

As an example, if we were to add two calls to “PrintOutput()” to the time marching loop in our reactor

```
// Initialize the instance.
cp.Initialize();
cp.PrintOutput("myPrint.txt");

// Time marching loop.
while (true) {
    double deltat = cp.ReportTimeStep();
    if (cp.AdvanceTimeStep(deltat)) break;
    if (cp.AdvanceChemical()) break;
    cp.PrintOutput();
}
```

the program would write into “myPrint.txt” a block representing the initial condition, followed by blocks representing the result after completing each time step.

In a variation on the coding

```
// Initialize the instance.
cp.Initialize();
cp.PrintOutput("Initial.txt");

// Time marching loop.
```

```
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   cp.PrintOutput("Path.txt");
}
```

the initial condition would be written to "Initial.txt" and results of the time stepping to "Path.txt".

An optional second argument to "PrintOutput()" causes the instance to post an identifying label at the head of the data block being written; a third argument rewinds the output dataset before writing to it, if the argument evaluates to true. In the time marching loop

```
// Initialize the instance.
cp.Initialize();

// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   cp.PrintOutput("myPrint.txt", "Latest result", true);
   std::cin.get();
}
```

the program pauses after each step in the time marching so the user can inspect the results for that step only in file "myPrint.txt".

5.3.2 Plot output

A plot-format dataset consists of three parts: a header, a series of data blocks, and a trailer. All three are needed if **Gtplot** is to render the data graphically. Member function "PlotHeader()" writes the dataset header, "PlotBlock()" adds a data block, and "PlotTrailer()" writes the trailer.

Function "PlotHeader()" works like "PrintOutput()". If the client specifies a file name, the function writes a header to the named file; otherwise it writes to whatever file is open for plot-format output, or to "ChemPlugin_plot.gtp" if none is open. The function honors any suffix that has been set, so

```
myCpi.Config("suffix _mysuffix");
myCpi.PlotHeader("myPlot.gtp");
```

writes a plot header to dataset "myPlot_mysuffix.gtp".

To create a plot dataset, a client calls "PlotHeader()" once and "PlotBlock()" each time it wants to add the results for a time step. When function "ReportTimeStep()"

detects the end of a simulation, it automatically appends the dataset trailer. For this reason, it is commonly not necessary to call “PlotTrailer()”, although doing so simply overwrites any existing trailer and hence is harmless.

As an example, the time marching loop

```
// Initialize the instance.
cp.Initialize();
cp.PlotHeader("myPlot.gtp");
cp.PlotBlock();

// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   cp.PlotBlock();
}
cp.PlotTrailer();  // Not necessary.
```

creates a valid plot-format dataset “myPlot.gtp” that contains the initial condition and the instance’s state after each step in the time marching loop.

Significantly, a client can extend a plot dataset, even after the plot trailer has been written. To do so, simply call “PlotBlock()” again as needed, then append the trailer once again, if necessary. In the time marching loop

```
// Initialize the instance.
cp.Initialize();
cp.PlotHeader("myPlot.gtp");
cp.PlotBlock();

// Time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   cp.PlotBlock();
   cp.PlotTrailer();
   std::cout << "Pausing at Xi = " << cp.Report1("Xi") << std::endl;
   std::cin.get();
}
cp.PlotTrailer();  // Not necessary.
```

the client program after completing each time step adds a data block and trailer to the plot dataset. The user can open the plot dataset with **Gtplot** to inspect results to that point in the calculation, before undertaking a new time step.

5.4 Contents of print-format output

A client controls the content and arrangment of information in a print-format dataset with the "print" configuration command. For example, the calls

```
myCpi.Config("print species = long");
myCpi.Config("print = alphabetic");
```

instruct the instance to write out information about all aqueous species, regardless of concentration, and to arrange the output alphabetically, rather than in decreasing numeric order. The **Configuration Commands** appendix of this Guide describes the "print" command in detail.

5.5 Source code

Source code for examples demonstrating direct output by a ChemPlugin instance are available as files "Titration4.cpp", "Titration5.cpp", and "Titration6.cpp" from the ChemPlugin.GWB.com website.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

Extending Runs

Once a time marching loop is complete, a client can extend the simulation to continue marching through time. The client can, furthermore, reconfigure the reaction parameters between time marching loops so that each loop traces a different reaction path. You can use ChemPlugin objects, then, to daisy-chain time marching loops into complex simulations.

6.1 Extending a titration

As an example of extending a simulation, we modify the client program we developed in the **Titration Simulator** chapter to simulate a second titration into result of the first titration.

To do so, we replace the initializiation step and time marching loop in file Titration1.cpp with the code fragment:

```
// Initialize the instance.
cp.Initialize(1.0, "hour");
std::cout << " Xi = " << cp.Report1("Xi");
std::cout << " pH = " << cp.Report1("pH") << std::endl;

// First time marching loop.
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   std::cout << " Xi = " << cp.Report1("Xi");
   std::cout << " pH = " << cp.Report1("pH") << std::endl;
}

// Reconfigure and extend the run.
std::cout << std::endl << " Extending run to Xi = 2" << std::endl;
cp.Config("remove reactant NaOH");
cp.Config("react 3 mmol/kg HCl");
cp.ExtendRun(1.0, "hour");

// Second time marching loop.
```

```
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceChemical()) break;
   std::cout << " Xi = " << cp.Report1("Xi");
   std::cout << " pH = " << cp.Report1("pH") << std::endl;
}
```

The client now contains two time marching loops, one following the other.

Where the client calls member function "Initialize()" to initialize instance "cp", it uses the function's optional arguments to set a time span of 1 hour; this end time applies to the first loop. After that loop completes at $\xi = 1$, the client replaces the NaOH titrant with HCl and calls member function "ExtendRun()" to add an hour to the simulation. The client then enters a second time marching loop, coded identically to the first, stepping from $\xi = 1$ to $\xi = 2$.

Output from simulating the dual titration looks like:

```
ChemPlugin example -- extending a pH titration


Solving for initial system.

 Loaded:   17 aqueous species,
           16 minerals,
            2 gases,
            0 surface species,
            6 elements,
            3 oxides.

 Xi = 0.00 pH = 4.00
 Xi = 0.10 pH = 5.39
 Xi = 0.20 pH = 5.86
 Xi = 0.30 pH = 6.15
 Xi = 0.40 pH = 6.42
 Xi = 0.50 pH = 6.70
 Xi = 0.60 pH = 7.08
 2 supersaturated phases, most = Calcite
 Swapping Calcite in for CO2(aq)
 Xi = 0.70 pH = 7.68
 Xi = 0.80 pH = 7.87
 Xi = 0.90 pH = 8.16
 Xi = 1.00 pH = 8.72

Successful completion of reaction path.


 Extending run to Xi = 2
```

```
Xi = 1.10 pH = 8.17
Xi = 1.20 pH = 7.88
Xi = 1.30 pH = 7.69
Xi = 1.33 pH = 7.64
Xi = 1.33 pH = 7.64
Calcite is undersaturated
Swapping CO2(aq) in for Calcite
Xi = 1.33 pH = 7.64
Xi = 1.33 pH = 7.64
... and so on ...
Xi = 1.97 pH = 4.51
Xi = 2.00 pH = 4.01

Successful completion of reaction path.
```

Note the instance takes small steps near the point at which $CaCO_3$ dissolves away completely, in order to predict the point at which the phase disappears precisely.

6.2 C++ source code

The full C++ source code for the client program above is available for download from the ChemPlugin.GWB.com website as file “Extend1.cpp”.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

React Emulator

In this chapter, we set out to generalize the titration model “Titration.cpp” we developed in previous chapters. To this end, we embed an instance of the ChemPlugin object within a client program in order to create a general-purpose reaction simulator.

Our program—we call it **mReact**—uses ChemPlugin to emulate the **React** application in The Geochemist’s Workbench. ChemPlugin and **React** are configured in terms of similar sets of commands, as described in the **Configuration Commands** appendix.

Program **mReact** works by configuring a ChemPlugin instance with commands taken from an input file. Whenever **mReact** encounters the command “go” in the input stream, it triggers the instance to trace a reaction model, as prescribed by its current configuration. When finished, the program returns to processing commands.

We examine here the program piece by piece, but you may wish to reference the complete C++ code for **mReact**, given at the end of this chapter, as you work.

7.1 Program structure

The **mReact** program is laid out as a console program in file “mReact.cpp”. The general structure is as follows:

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

void open_input(std::ifstream& input, int argc, char** argv) {
    ... function goes here ...
}

int main(int argc, char** argv) {
    ... main program goes here ...
}
```

The first four lines import system headers and the ChemPlugin header “ChemPlugin.h”, which is installed with the ChemPlugin software.

The program then sets out function "open_input", which identifies the input file and opens a data stream from it. We won't discuss this function, since it is not related to ChemPlugin, but its coding is shown at the end of the chapter. The remainder of the file sets out the coding for the main program.

7.2 Main program

The main **mReact** program has the form:

```
int main(int argc, char** argv) {
   std::cout << "mReact -- Use ChemPlugin to emulate React"
             << std::endl << std::endl;

   // Create a ChemPlugin instance and capture output messages.
   ChemPlugin cp("stdout");

   // Use React's default settings; write print- and plot-format output.
   cp.Config("delxi = 0.01; step_increase = 1.5; pluses = banner");
   cp.Config("print = on; plot = on");

   // Process input line-by-line while watching for "go" statements.
   std::ifstream input;
   open_input(input, argc, argv);
      ... input loop goes here ...

   // Any keystroke closes the console.
   std::cin.get();
   return 0;
}
```

After identifying itself, the program creates a ChemPlugin instance "cp", which will direct its console messages to the standard output stream.

mReact next uses member function "Config()" to send a series of commands to the ChemPlugin instance. The default settings for a small number of variables in ChemPlugin differ from those in **React**, as described in the **Configuration Commands** appendix. Here we set those values to the defaults carried by **React**.

A second call to "Config()" turns on generation of print-format and plot-format output files. As it traces a simulation, **mReact** will now write the calculation results to files "ChemPlugin_output.txt" and "ChemPlugin_plot.gtp".

Next, function "open_input()" opens the input stream and **mReact** enters into a loop in which it reads and processes the input file, line by line. The input loop is described in the next subsection. When the loop terminates, **mReact** waits for a keystroke from the user and closes the console.

7.2.1 Input loop

The input loop proceeds by fetching lines from the input file until reaching the end:

```
while (!input.eof()) {
    std::string line;
    std::getline(input, line);
    if (line != "go" ) {
        cp.Config(line);
    }
    else {
        ... time marching loop goes here ...
    }
}
```

Each line of input is passed to the ChemPlugin instance using member function “Config()”, unless the line contains the command “go”. In that case, **mReact** enters the time marching loop to trace the simulation.

7.2.2 Time marching loop

The time marching loop by which **mReact** carries out the reactor simulation consists of only a few lines of code:

```
cp.Initialize();
while (true) {
    double deltat = cp.ReportTimeStep();
    if (cp.AdvanceTimeStep(deltat)) break;
    if (cp.AdvanceHeatTransport()) break;
    if (cp.AdvanceChemical()) break;
}
```

The call to member function “Initialize()” triggers the ChemPlugin instance to calculate its initial state, as defined by the input commands processed to this point, and prepare to enter the time marching loop.

The time marching loop consists of four member function calls executed in a cycle. A call to “ReportTime()” triggers the instance to calculate an appropriate time step size. The function returns a value for Δt in units of seconds, and the program stores the value in variable “deltat”.

A call to “AdvanceTimeStep()” passes the time step size to the instance. Upon executing the function, the instance moves forward in reaction progress. The function returns a value of zero, unless **mReact** has reached the end of the simulation. In that case, a non-zero return breaks the loop and time marching ceases.

Calling “AdvanceHeatTransport()” updates temperature in a polythermal path. Finally, executing member function “AdvanceChemical()” causes the instance to evaluate the chemical equations at the new point in time, accounting for equilibrium as well as kinetic reactions. The function returns zero, unless an error occurs. A non-zero return marks an error, breaking the loop.

7.3 Running the example program

To test out our program, we prepare an input file "Acidity.rea"

```
Ca++ = 1 mmol/kg
Na+ = 1 mmol/kg
Cl- = 3 mmol/kg
HCO3- = 2 mmol/kg
pH = 4
react 3 mmol/kg NaOH
go
```

that reacts NaOH into an initially acidic fluid. Executing the client program using this file as input writes the following to the console:

```
mReact -- Use ChemPlugin to emulate React

Enter React input script: Acidity.rea
Reading from file Acidity.rea

Solving for initial system.

 Loaded:   17 aqueous species,
           16 minerals,
            2 gases,
            0 surface species,
            6 elements,
            3 oxides.

Step   0, Xi = 0     (32 iterations)
 Charge balance: Cl- molality adjusted from .003 to .003098
Step   1, Xi = .01   (9 iterations)
Step   2, Xi = .02   (7 iterations)
Step   3, Xi = .03   (7 iterations)
Step   4, Xi = .04   (8 iterations)
Step   5, Xi = .05   (8 iterations)

  ... and so on ...

Step  95, Xi = .95   (9 iterations)
Step  96, Xi = .96   (7 iterations)
Step  97, Xi = .97   (9 iterations)
Step  98, Xi = .98   (9 iterations)
Step  99, Xi = .99   (9 iterations)
Step 100, Xi = 1     (9 iterations)

Successful completion of reaction path.
```

Upon completion, the calculation results are found in files “ChemPlugin_output.txt” and “ChemPlugin_plot.gtp”.

7.4 mReact C++ code

The full source code for **mReact** is given in file “mReact.cpp”, available for download from the ChemPlugin.GWB.com website.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

The code is also reproduced below:

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

void open_input(std::ifstream& input, int argc, char** argv) {
   while (!input.is_open()) {
      std::string filename;
      if (argc < 2) {
         std::cout << "Enter React input script: ";
         std::cin >> filename;
         std::cin.ignore();
      }
      else {
         filename = argv[1];
      }

      input.open(filename);

      if (!input.is_open())
         std::cerr << "The input file does not exist" << std::endl;
   }
}

int main(int argc, char** argv) {
   std::cout << "mReact -- Use ChemPlugin to emulate React"
             << std::endl << std::endl;

   // Create a ChemPlugin instance and capture output messages.
   ChemPlugin cp("stdout");

   // Use React's default settings; write print- and plot-format output.
   cp.Config("delxi = 0.01; step_increase = 1.5; pluses = banner");
   cp.Config("print = on; plot = on");

   // Process input line-by-line while watching for "go" statements.
```

```
    std::ifstream input;
    open_input(input, argc, argv);
    while (!input.eof()) {
        std::string line;
        std::getline(input, line);
        if (line != "go" ) {
            cp.Config(line);
        }
        else {
            cp.Initialize();
            while (true) {          // Time marching loop.
                double deltat = cp.ReportTimeStep();
                if (cp.AdvanceTimeStep(deltat)) break;
                if (cp.AdvanceHeatTransport()) break;
                if (cp.AdvanceChemical()) break;
            }
        }
    }

    // Any keystroke closes the console.
    std::cin.get();
    return 0;
}
```

Linking Instances

A link is a connection between two ChemPlugin instances, across which chemical mass and heat energy may pass. This chapter shows how a client program sets and removes links.

Note that a client can create any number of links between two instances. It might set a link to carry flow from one instance to another, then a second to handle the possibility of back-flow.

You should know as well that links are reciprocal: when a client connects one instance to another, it should not then connect the second instance to the first, except to form a second link.

8.1 Linking instances

A client program connects two instances with member function “Link()”, which returns a reference of type “CpiLink” to the resulting link. For example, the code

```
ChemPlugin cp0, cp1;
CpiLink link0 = cp0.Link(cp1);
```

connects “cp0” and “cp1”, storing a reference to the link in variable “link0”. Once “link0” is created, if the client were to then execute

```
CpiLink link1 = cp1.Link(cp0);
```

the call would set a second connection between the instances, as referenced by “link1”. Most commonly, a single connection between any two nodes is all that is required.

Once two ChemPlugin instances are linked, a client might wish to hold onto the link’s reference. For example, the code

```
link0.FlowRate(2.0, "m3/day");
```

sets flow rate across “link0”, as discussed in the next chapter. References to links, nonetheless, can be recovered easily. If two links to “cp0” have been set, for example, references to the links are returned

```
CpiLink link0 = cp0.Link(0);
CpiLink link1 = cp0.Link(1);
```

by calling the "Link()" member function with an integer argument.

It is also easy to determine the number of links to an instance, by using member function "nLinks()". The code

```
int nlinks = cp0.nLinks();
int nlink1 = cp0.nLinks(cp1);
```

stores the total number of links to "cp0" in variable "nlinks", and the number of links between "cp0" and "cp1" in "nlink1".

8.2 Free outlets

Free outlets are open links to a ChemPlugin instance. Fluid may flow across a free outlet away from the instance, but not toward it. A free outlet, furthermore, cannot carry diffusive transport, or conduct heat.

A client can set a free outlet with the "Outlet()" member function,

```
CpiLink outlet0 = cp0.Outlet();
```

or, equivalent, by calling "Link()"

```
CpiLink outlet0 = cp0.Link();
```

without an argument.

Member function "nOutlets()"

```
int noutlet0 = cp0.nOutlets();
```

reports the number of free outlets connected to an instance.

8.3 Removing links

A client program is free at any time to reconfigure the arrangement of ChemPlugin instances by removing and creating links. Function "Unlink()", which serves to remove links, is a member of both the "ChemPlugin" and "CpiLink" classes.

Once a client has linked two "ChemPlugin" instances

```
ChemPlugin cp0, cp1;
CpiLink link0 = cp0.Link(cp1);
```

it may remove that link by its reference

```
link0.Unlink();
```

by reference to the linked instance

```
cp0.Unlink(cp1);
```

or by index

```
cp0.Unlink(0);
```

Each case is equivalent in function, and in each case a return value of zero signals success.

Member function “ClearLinks()”

```
cp0.ClearLinks();
```

removes all of the links to an instance; again, success is signaled by a return value of zero.

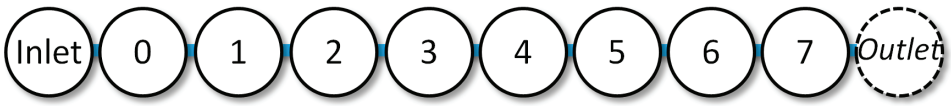
8.4 Example programs

The examples below show how to link ChemPlugin instances in varying geometries. The header lines in the second and third examples are not shown.

8.4.1 Linear chain

First, we consider a client program that arranges eight ChemPlugin instances in a linear chain. The chain is connected to an inlet boundary on the left side, and a free outlet boundary on the right.

The instances are referenced left-to-right as “cp[0]” through “cp[7]”, a boundary instance “cp_inlet” represents the inlet condition, and the free outlet is an open link:



The client program is given:

```
#include <iostream>
#include "ChemPlugin.h"

int main(int argc, char** argv) {
   std::cout << "Link ChemPlugin instances into a one-dimensional chain"
             << std::endl << std::endl;
```

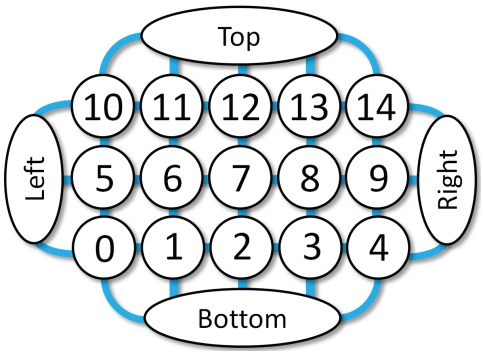
```
    // Create the ChemPlugin instances.
    int nchain = 8;
    ChemPlugin cp_inlet;
    ChemPlugin *cp = new ChemPlugin[nchain];

    // Link the instances into a chain.
    cp[0].Link(cp_inlet);
    for (int i=1; i<nchain; i++)
       cp[i].Link(cp[i-1]);
    cp[nchain-1].Outlet();

    // Report the number of links to each instance.
    std::cout << "Inlet is linked to " << cp_inlet.nLinks()
              << " instance" << std::endl;
    for (int i=0; i<nchain; i++)
       std::cout << "Instance " << i << " is linked to " << cp[i].nLinks()
                 << " instances" << std::endl;

    // Any keystroke closes the console.
    std::cin.get();
    delete[ ] cp;
    return 0;
}
```

Running the program produces the console output

```
Link ChemPlugin instances into a one-dimensional chain

Inlet is linked to 1 instance
Instance 0 is linked to 2 instances
Instance 1 is linked to 2 instances
Instance 2 is linked to 2 instances
Instance 3 is linked to 2 instances
Instance 4 is linked to 2 instances
Instance 5 is linked to 2 instances
Instance 6 is linked to 2 instances
Instance 7 is linked to 2 instances
```

8.4.2 Grid

Second, we consider a 5×3 grid of instances, with the left, right, bottom, and top of the grid linked to boundary instances.



Running the client program

```
int main(int argc, char** argv) {
   int nx = 5, ny = 3;
   std::cout << "Link ChemPlugin instances into a " << nx << " by " << ny
             << " grid" << std::endl << std::endl;

   // Create the ChemPlugin instances.
   ChemPlugin cp_left, cp_right, cp_bottom, cp_top;
   ChemPlugin *cp = new ChemPlugin[nx*ny];

   // Link the instances into a tree.
   for (int j=0; j<ny; j++) {
      for (int i=0; i<nx; i++) {
         int ij = i + j*nx;
         cp[ij].Link(i == 0? cp_left : cp[ij-1]);
         cp[ij].Link(j == 0? cp_bottom : cp[ij-nx]);
         if (j == ny-1) cp_top.Link(cp[ij]);
      }
      cp_right.Link(cp[(j+1)*nx-1]);
   }

   // Report the number of links to each instance.
   std::cout << "Left boundary is linked to " << cp_left.nLinks()
             << " instances" << std::endl;
   std::cout << "Bottom boundary is linked to " << cp_bottom.nLinks()
             << " instances" << std::endl;
   for (int i=0; i<nx*ny; i++)
      std::cout << "Instance " << i << " is linked to " << cp[i].nLinks()
                << " instances" << std::endl;
   std::cout << "Top boundary is linked to " << cp_top.nLinks()
             << " instances" << std::endl;
   std::cout << "Right boundary is linked to " << cp_right.nLinks()
             << " instances" << std::endl;
```

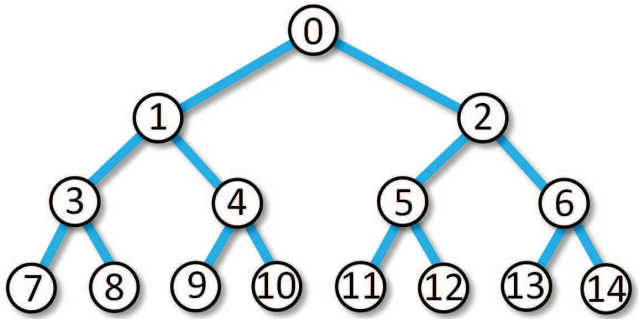
```
    // Any keystroke closes the console.
    std::cin.get();
    delete[] cp;
    return 0;
}
```

produces the console output

```
Link ChemPlugin instances into a 5 by 3 grid

Left boundary is linked to 3 instances
Bottom boundary is linked to 5 instances
Instance 0 is linked to 4 instances
Instance 1 is linked to 4 instances
Instance 2 is linked to 4 instances
Instance 3 is linked to 4 instances
Instance 4 is linked to 4 instances
Instance 5 is linked to 4 instances
Instance 6 is linked to 4 instances
Instance 7 is linked to 4 instances
Instance 8 is linked to 4 instances
Instance 9 is linked to 4 instances
Instance 10 is linked to 4 instances
Instance 11 is linked to 4 instances
Instance 12 is linked to 4 instances
Instance 13 is linked to 4 instances
Instance 14 is linked to 4 instances
Top boundary is linked to 5 instances
Right boundary is linked to 3 instances
```

8.4.3 Bifurcating tree

Finally, we look at a client program that builds a bifurcating tree of 4 levels.



Since there are $2^N - 1$ nodes in a bifuncating tree of N levels, there will be 15 ChemPlugin instances in our linked domain.

The client program is given:

```
int main(int argc, char** argv) {
   int nlevel = 4;
   std::cout << "Link ChemPlugin instances into a " << nlevel
             << " bifurcating tree" << std::endl << std::endl;

   // Create the ChemPlugin instances.
   int ninst = pow(2, nlevel) - 1;
   ChemPlugin *cp = new ChemPlugin[ninst];

   // Link the instances into a tree.
   int inst = 0, linked_inst = 1;
   for (int level=0; level<nlevel-1; level++) {
      for (int i=0; i<pow(2, level); i++) {
         cp[linked_inst++].Link(cp[inst]);
         cp[linked_inst++].Link(cp[inst]);
         inst++;
      }
   }

   // Report the number of links to each instance.
   for (int i=0; i<ninst; i++)
      std::cout << "Instance " << i << " is linked to  " << cp[i].nLinks()
                << " instance(s)" << std::endl;

   // Any keystroke closes the console.
   std::cin.get();
   delete[ ] cp;
   return 0;
}
```

Running the client produces the following output:

```
Link ChemPlugin instances into a 4 bifurcating tree

Instance 0 is linked to 2 instance(s)
Instance 1 is linked to 3 instance(s)
Instance 2 is linked to 3 instance(s)
Instance 3 is linked to 3 instance(s)
Instance 4 is linked to 3 instance(s)
Instance 5 is linked to 3 instance(s)
Instance 6 is linked to 3 instance(s)
Instance 7 is linked to 1 instance(s)
Instance 8 is linked to 1 instance(s)
Instance 9 is linked to 1 instance(s)
Instance 10 is linked to 1 instance(s)
Instance 11 is linked to 1 instance(s)
```

```
Instance 12 is linked to 1 instance(s)
Instance 13 is linked to 1 instance(s)
Instance 14 is linked to 1 instance(s)
```

8.4.4 C++ source code

Source code for the examples above are given in files "Links1.cpp", "Links2.cpp", and "Links3.cpp", which can be downloaded from the ChemPlugin.GWB.com website.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

Flow and Transport

Advective transport is the movement of chemical components among ChemPlugin instances, as the result of fluid flowing across links. To model advective transport, a client program must set the rate at which fluid crosses each link in a simulation, in terms of volume per unit time. The ChemPlugin instances on either side of a link, then, use that flow rate to figure fluxes across the link, in moles per unit time, of the chemical components considered in the simulation.

Specifically, if Q is the flow rate across a link, in units of $m^3\ s^{-1}$, then the advective flux J_i^o of chemical component i, in mol s^{-1}, is given

$$J_i^o = Q C_i \tag{9.1}$$

where C_i is the concentration of i, in mol m^{-3}.

This transport law is set out in terms of the concentration itself, rather than a derivative. As such, advective transport is commonly referred to as a zero-order process; hence, the notation J_i^o. The topic of first-order transport, in which the transport law is written in terms of the gradient (i.e., the first-order derivative) of concentration, is treated in the next chapter.

9.1 Flow rate

A client program uses the "FlowRate()" member function to specify the flow rate across a link, or to retrieve such a value, as previously set.

9.1.1 Setting the flow rate

To set the rate at which fluid moves across a link, the client passes "FlowRate()" the fluid volume crossing the link per unit time. Flow is by convention positive when it moves toward the instance that created the link, and negative in the opposite direction. For example, in the code

```
ChemPlugin cp0, cp1;
cp0.Link(cp1);
```

flow from "cp1" toward "cp0" is positive in sign, whereas flow away from "cp0" is negative.

The "FlowRate()" function is a member of the "CpiLink" class, so it is used as follows:

```
ChemPlugin cp0, cp1;
CpiLink link = cp0.Link(cp1);
link.FlowRate(.52e6, "cm3/s");
```

The function takes two arguments: the value of the flow rate and, optionally, the unit in which the value is cast. Units for the flow rate include "cm3/s", "m3/yr", "gal/day", and so on, as set out in the **Units Recognized** appendix. If the client does not specify a second argument,

```
link.FlowRate(.52);
```

the value is taken to be in $m^3\ s^{-1}$.

9.1.2 Retrieving the flow rate

The client program can also use the "FlowRate()" member function to determine the rate of flow across a link, once it has been set. To do so, the client calls the member function without specifying a value. Following from above, the statement

```
double flow = link.FlowRate("cm3/s");
```

stores in variable "flow" the current flow rate across "link", in units of $cm^3\ s^{-1}$. Alternatively, the statement

```
double flow = link.FlowRate();
```

stores the flow rate, cast in the default units, $m^3\ s^{-1}$.

9.1.3 Steady and transient flow

A client program may set the flow rate across each link once, at the onset of the simulation. The flow field in this case is invariant in time, or steady. Alternatively, the client may specify flow repeatedly, upon commencing each time step, in order to construct a transient flow field.

9.2 Stability

In solving for flow and transport using a finite-volume code like ChemPlugin, numerical stability of the time stepping is limited by the Courant condition. The Courant condition requires that a time step not exceed the time required to displace all the fluid from a ChemPlugin instance.

The fraction of an instance's fluid displaced over a time step is the Courant number C_o. If V_f is the volume of fluid contained in an instance, in m^3, and Q_ℓ is the flow rate

across a link ℓ, in $m^3\ s^{-1}$, then the Courant condition can be expressed

$$C_o = \left(\sum_{\ell: Q_\ell > 0} Q_\ell \right) \Delta t \Big/ V_f \leq 1 \tag{9.2}$$

Here, we find the Courant number by adding together the flow rates of each link with a positive flow rate—that is, each link transporting fluid into the instance—multiplying by the time step Δt, and dividing by V_f.

Each ChemPlugin instance carries a limiting Courant number $C_o^{\lim}$ that it uses to constrain the time step returned by "ReportTimeStep()", according to the inequality above. As such,

$$\Delta t = C_o^{\lim} V_f \Big/ \left(\sum_{\ell: Q_\ell > 0} Q_\ell \right) \tag{9.3}$$

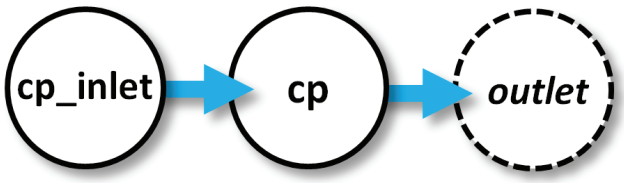
The value of $C_o^{\lim}$ is carried by default as 1.0, but a client program may set it to any value in the range $0 < C_o^{\lim} \leq 1$ with the "Courant" configuration command:

```
cp.Config("Courant = 0.5");
```

In this example, no more than half the fluid in instance "cp" will be displaced over a time step.

9.3 Flow-through reactor

As an example of how a client program can use linked ChemPlugin instances to model reaction processes in open systems, we construct here a client program that simulates a well-stirred flow-through reactor.



In our program, a ChemPlugin instance "cp_inlet" represents the inlet fluid that passes into the well-stirred reactor, which is represented by instance "cp". Fluid from "cp_inlet" flows across a link into "cp", displacing fluid from it. The displaced fluid follows an open link to the free outlet, where it is lost.

9.3.1 Program structure

The client program works by setting up a ChemPlugin instance for the inlet fluid and another for the stirred reactor. The client then links the instances and sets the flow rate across the links.

The general structure of the client is:

```
#include <iostream>
#include "ChemPlugin.h"

int main(int argc, char** argv) {
   std::cout << "Model a flow-through reactor" << std::endl << std::endl;

   // Configure and initialize the inlet fluid.
   ... set up the inlet fluid ...

   // Configure and initialize the stirred reactor.
   ... set up the stirred reactor ...

   // Link reactor to inlet and free outlet; set rate of flow.
   ... link the instances and set flow rates ...

   // Time marching loop.
   ... time marching loop goes here ...

   // Any keystroke closes the console.
   std::cin.get();
   return 0;
}
```

We will discuss each part of the client in the sections below.

9.3.2 Inlet fluid

To begin, the client sets a ChemPlugin instance "cp_inlet" to represent the inlet fluid, which is a dominantly HCl solution of pH 1.

```
   // Configure and initialize the inlet fluid.
   ChemPlugin cp_inlet("stdout");
   const char *cmds = "Ca++ = 1 mmol/kg; HCO3- = 1 mmol/kg;"
                      "pH = 1; balance on Cl-";
   cp_inlet.Config(cmds);
   cp_inlet.Initialize();
```

The client creates the instance, setting it to write out console messages, configures it according to the character string pointed to by "cmds", and initializes it.

The inlet instance serves in the program as a static fluid of known composition. There is no need to specify a time span for the instance, nor does the instance's

extent (i.e., its volume or mass) need to be known; by the zero-order equation above, the concentrations but not the masses of the fluid's chemical components enter into the transport calculation.

9.3.3 Stirred reactor

Next, the client sets a ChemPlugin instance "cp" to act in the simulation as a stirred reactor.

```
// Configure and initialize the stirred reactor.
ChemPlugin cp("stdout");
cmds = "swap Calcite for Ca++; Calcite = 0.03 free m3; Cl- = 1 mmol/kg;"
       "swap CO2(g) for H+; fugacity CO2(g) = 1; balance on HCO3-;"
       "volume = 1 m3; fix f CO2(g); delxi = 0.01; pluses = banner";
cp.Config(cmds);
cp.Initialize(1.0, "day");
```

The client configures the instance to contain a fluid in equilibrium with $CaCO_3$ and a CO_2 reservoir of known fugacity, which is held constant over the simulation. The instance volume is set to 1 m^3, of which 0.03 m^3 consists of $CaCO_3$.

Setting "delxi" to 0.01 prescribes the simulation traverse 100 times steps over the course of the simulation, which is set to span 1 day. The command "pluses = banner" sets the instance to write a banner-style output at each reaction step.

9.3.4 Links and flow rates

We next link instance "cp" to instance "cp_inlet", create an open link from "cp", and set a flow rate across each link.

```
// Link reactor to inlet and free outlet; set rate of flow.
CpiLink link1 = cp.Link(cp_inlet);
CpiLink link2 = cp.Outlet();

link1.FlowRate(10.0, "m3/day");
link2.FlowRate(-10.0, "m3/day");
```

Since the flow rate is 10 m^3 day^{-1} and the simulation spans 1 day, 10 m^3 of fluid will pass into "cp", and an equivalent volume will be displaced across the free outlet.

Note especially that the flow rate across the first link, from "cp" to "cp_inlet", is positive, since fluid is flowing toward "cp". The rate at which fluid is displaced across the second link, the free outlet, on the other hand, is negative, since flow is in this case away from "cp".

9.3.5 Time marching loop

The time marching loop is similar to the loop we constructed in the first program we wrote, in the **Titration Simulator** chapter, except we have added a call to member function "AdvanceTransport()":

```
// Time marching loop.
cp.PlotHeader("FlowThrough.gtp", "char");
cp.PlotBlock();
while (true) {
   double deltat = cp.ReportTimeStep();
   if (cp.AdvanceTimeStep(deltat)) break;
   if (cp.AdvanceTransport()) break;
   if (cp.AdvanceChemical()) break;
   cp.PlotBlock();
}
```

By calling "AdvanceTransport()", we trigger instance "cp" to account for how flow from "cp_inlet" into "cp", as well as flow from "cp" into the free outlet, affect the chemical composition of the reactor. Note that we've used calls to member functions "PlotHeader()" and "PlotBlock()" to trigger "cp" to create a plot output file, "FlowThrough.gtp".

9.3.6 Program output

Running our client program produces the following output:

```
Model a flow-through reactor


Solving for initial system.

 Loaded:   12 aqueous species,
           13 minerals,
            2 gases,
            0 surface species,
            5 elements,
            2 oxides.


Solving for initial system.

 Loaded:   12 aqueous species,
           13 minerals,
            2 gases,
            0 surface species,
            5 elements,
            2 oxides.

Step   0, Xi = 0     (85 iterations)
 Charge balance: HCO3- molality adjusted from 1.02e-6 to -.001003
Step   1, Xi = .01   (31 iterations)
Step   2, Xi = .02   (29 iterations)
Step   3, Xi = .03   (28 iterations)
```

```
Step   4, Xi = .04   (28 iterations)
Step   5, Xi = .05   (28 iterations)

... and so on ...

Step 100, Xi = 1     (16 iterations)

Successful completion of reaction path.
```

Once the run completes, we can open dataset “FlowThrough.gtp” with program **Gtplot** to render the calculation results graphically.

9.3.7 C++ source code

The complete C++ source code to our flow-through simulator can be downloaded from the ChemPlugin.GWB.com website as file “FlowThrough1.cpp”, and is listed below.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

```
#include <iostream>
#include "ChemPlugin.h"

int main(int argc, char** argv) {
   std::cout << "Model a flow-through reactor" << std::endl << std::endl;

   // Configure and initialize the inlet fluid.
   ChemPlugin cp_inlet("stdout");
   const char *cmds = "Ca++ = 1 mmol/kg; HCO3- = 1 mmol/kg;"
                      "pH = 1; balance on Cl-";
   cp_inlet.Config(cmds);
   cp_inlet.Initialize();

   // Configure and initialize the stirred reactor.
   ChemPlugin cp("stdout");
   cmds = "swap Calcite for Ca++; Calcite = 0.03 free m3; Cl- = 1 mmol/kg;"
          "swap CO2(g) for H+; fugacity CO2(g) = 1; balance on HCO3-;"
          "volume = 1 m3; fix f CO2(g); delxi = 0.01; pluses = banner";
   cp.Config(cmds);
   cp.Initialize(1.0, "day");

   // Link reactor to inlet and free outlet; set rate of flow.
   CpiLink link1 = cp.Link(cp_inlet);
   CpiLink link2 = cp.Outlet();

   link1.FlowRate(10.0, "m3/day");
   link2.FlowRate(-10.0, "m3/day");

   // Time marching loop.
```

```
    cp.PlotHeader("FlowThrough.gtp", "char");
    cp.PlotBlock();
    while (true) {
        double deltat = cp.ReportTimeStep();
        if (cp.AdvanceTimeStep(deltat)) break;
        if (cp.AdvanceTransport()) break;
        if (cp.AdvanceChemical()) break;
        cp.PlotBlock();
    }

    // Any keystroke closes the console.
    std::cin.get();
    return 0;
}
```

Diffusion and Dispersion

Diffusive transport is the movement of chemical components in response to gradients in concentration. The transport arises due to chemical and physical processes, such as molecular diffusion, hydrodynamic dispersion, and turbulent mixing.

Transport of this nature is commonly described by Fick's first law

$$J_i^1 = -AD\frac{dC_i}{dx} \tag{10.1}$$

In this equation, J_i^1 is the flux of chemical component i, in mol s^{-1}; A is the cross-sectional area across which transport occurs, in m^2; D is a Fickian coefficient, in m^2 s^{-1}; C_i is the concentration of component i, in mol m^{-3}; and x is the spatial coordinate between two ChemPlugin instances, in m, positive displacement being toward the originating instance. The notation J_i^1 reflects the first-order derivative in the transport law.

The precise form of D depends on the process being represented. To model molecular diffusion in porous media, $D = nD^*$, where n is porosity and D^* is the diffusion coefficient for the medium, accounting for its tortuosity. In the case of hydrodynamic dispersion, $D = n(\alpha v_x + D^*)$, where α is dispersivity in m, v_x is fluid velocity along x in m s^{-1}, and n and D^* are as before. For turbulent mixing, D is the eddy diffusivity K in m^2 s^{-1}.

To model diffusive transport across a link, a client program supplies a transmissivity that describes the rate of first-order transport, per unit concentration difference between ChemPlugin instances. The following section describes the transmissivity coefficient.

10.1 Transmissivity

ChemPlugin instances employ a transmissivity τ, in units of m^3 s^{-1}, to calculate the first-order mass fluxes, according to the equation

$$J_i^1 = -\tau\left(C_i^j - C_i^{\text{linked}}\right) \approx -AD\frac{dC_i}{dx} \tag{10.2}$$

Here, C_i^j and C_i^{linked} are the concentrations of component i at the originating and linked instances, respectively. The transmissivity, then, is defined

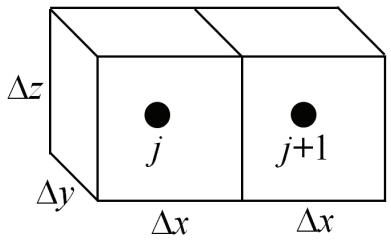
$$\tau = \frac{AD}{\Delta x} \approx \frac{AD}{dx} \tag{10.3}$$

by the cross-sectional area A, Fickian coefficient D, and separation Δx between instances.

10.1.1 Determining transmissivity

To model first-order mass transport among ChemPlugin instances, the client program must specify a value for the transmissivity of each link. A link's transmissivity reflects the geometry of the linked instances, as well as the Fickian coefficient for each.

Consider two linked instances j and $j+1$, each of which is a rectangular prism



Here Δx is the x dimension of the prisms, and A is the product $\Delta y \Delta z$ of the dimensions along y and z.

If the prisms are equally sized, so $\Delta x^j = \Delta x^{j+1} = \Delta x$ and $A^j = A^{j+1} = A$, and if the Fickian coefficient representing each is the same, so $D^j = D^{j+1} = D$, then the transmissivity τ

$$\tau = \frac{AD}{\Delta x} \tag{10.4}$$

is given directly.

In a heterogeneous case in which $D^j \neq D^{j+1}$, the transmissivity τ is the harmonic mean

$$\tau = \frac{2}{1/\tau^j + 1/\tau^{j+1}} \tag{10.5}$$

of the transmissivity τ^j at instance j and the value τ^{j+1} at instance $j+1$

$$\tau^j = \frac{AD^j}{\Delta x} \qquad \tau^{j+1} = \frac{AD^{j+1}}{\Delta x} \tag{10.6}$$

Substituting gives the transmissivity

$$\tau = \left(\frac{2A}{\Delta x}\right) \frac{D^j D^{j+1}}{D^j + D^{j+1}} \tag{10.7}$$

appropriate for describing first-order transport across the link between j and $j+1$.

More generally, the instances may be cast in an adaptive grid, or in non-Cartesian coordinates, such as radial, spherical, and curvilinear. The instances, then, may be of varying size, such that Δx^j depends on position j, as does the cross-sectional area A^j. In this case,

$$\tau^j = \frac{A^j D^j}{\Delta x^j} = \left(\frac{AD}{\Delta x}\right)^j \qquad \tau^{j+1} = \frac{A^{j+1} D^{j+1}}{\Delta x^{j+1}} = \left(\frac{AD}{\Delta x}\right)^{j+1} \tag{10.8}$$

Taking once again the harmonic mean gives the transmissivity

$$\tau = \frac{2\left(\frac{AD}{\Delta x}\right)^j \left(\frac{AD}{\Delta x}\right)^{j+1}}{\left(\frac{AD}{\Delta x}\right)^j + \left(\frac{AD}{\Delta x}\right)^{j+1}} \tag{10.9}$$

appropriate for the heterogeneous, arbitrarily gridded case.

10.1.2 Setting transmissivity

A client uses member function “Transmissivity()” to set transmissivity across a link. Like “FlowRate()”, “Transmissivity()” is a member of the “CpiLink” class; the function is used as follows:

```
ChemPlugin cp0, cp1;
CpiLink link = cp0.Link(cp1);
link.Transmissivity(2.e-3, "m3/s");
```

The unit field may be any unit of flow rate as set out in the **Units Recognized** appendix. If the client does not specify a second argument,

```
link.Transmissivity(2.e-3);
```

the value is taken to be in $m^3\ s^{-1}$.

10.1.3 Retrieving the transmissivity

When the “Transmissivity()” member function is called without an argument, it returns the transmissivity currently set for a link. For example, the statement

```
double trans = link.Transmissivity("cm3/s");
```

stores the current transmissivity value across "link" in variable "trans", in units of cm^3 s^{-1}. The statement

```
double trans = link.Transmissivity();
```

returns the value cast in the default units, m^3 s^{-1}.

10.2 Numerical stability

If D_x, D_y, and D_z are the Fickian coefficient D along the principal coordinates, von Neumann's criterion for numerical stability of a finite difference procedure requires that the time step Δt satisfy

$$2\left(\frac{D_x}{n\Delta x^2}+\frac{D_y}{n\Delta y^2}+\frac{D_z}{n\Delta z^2}\right)\Delta t \le 1 \tag{10.10}$$

where n is the porosity of a porous medium, or unity when considering transport in an open channel, and Δx, Δy, and Δz are the dimensions of the nodal block. Multiplying both sides of the equation by the fluid volume

$$V_f = n\Delta x\Delta y\Delta z \tag{10.11}$$

and substituting the transmissivities

$$\tau_x = \frac{A_x D_x}{\Delta x} \qquad \tau_y = \frac{A_y D_y}{\Delta y} \qquad \tau_z = \frac{A_z D_z}{\Delta z} \tag{10.12}$$

gives

$$\Delta t \le V_f \Big/ \left(\tau_x^{j-1/2}+\tau_x^{j+1/2}+\tau_y^{k-1/2}+\tau_y^{k+1/2}+\tau_z^{l-1/2}+\tau_z^{l+1/2}\right) \tag{10.13}$$

Here, $\tau_x^{j-1/2}$ represents the transmissivity between node j and $j-1$, $\tau_x^{j+1/2}$ is transmissivity between j and $j+1$, and so on.

Generalizing to a finite volume linked to an arbitrary number of other volumes, the equation becomes

$$\Delta t \le V_f \Big/ \left(\sum_{\ell}\tau_\ell\right) \tag{10.14}$$

where ℓ indexes the links, each of which has an associated transmissivity τ_ℓ. Recasting this expression, Δt is given uniquely by

$$\Delta t = X_{\text{stable}} V_f \Big/ \left(\sum_\ell \tau_\ell \right) \tag{10.15}$$

where X_{stable} is a value ≤ 1 provided by the user.

The value in ChemPlugin of X_{stable} is by default 1.0, but that is the boundary between stability and instability. As such, the user may wish to set a somewhat smaller value, which is accomplished by issuing within the client program the "Xstable" configuration command:

```
cp.Config("Xstable = 0.9");
```

In this case, the value of X_{stable} carried by instance "cp" is set to 0.9. If the client program then executed

```
double deltat = cp.ReportTimeStep();
```

the value returned to "deltat" would account for this setting.

10.3 Model of diffusion

As a demonstration of how a client program can use ChemPlugin to model diffusive transport, we construct here a one-dimensional model of diffusion within a porous medium. In our model, the domain is 100 cm long and contains a NaCl solution of concentration 1 mmol/kg where $0 \leq x < 50$ cm, and 0.001 mmol/kg where $50 < x \leq 100$ cm. At t = 0, the salt begins diffusing from left to right, toward large x.

10.3.1 Program structure

The client program is laid out as follows:

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
```

```
{
   ... output function goes here ...
}

int main(int argc, char** argv) {
   std::cout << "Model diffusion in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   ... simulation parameters are set out here ...

   // Open output file and write instance positions on the first line.
   ... output file is opened and initialized here ...

   // Configure and initialize the instances.
   ... instances are created, configured, and initialized here ...

   // Link the instances.
   ... links among the instances are created and defined here ...

   // Time marching loop.
   ... time marching loop goes here ...

   // Never gets here.
   return 0;
}
```

Two functions and the client program immediately follow the header lines at the top of the file. Function "exit_client()" provides a convenient way to ensure the console window does not close immediately when the client program completes, whether the client reaches the end of the simulation normally or encounters an error. Function "write_line()" writes out the calculation results at time levels separated by "gap" years, rather than writing results at each step, as set out in the next section. Finally, function "main()" is the client program itself, laid out here in subsections described below.

10.3.2 Output function

The purpose of function "write_line()" is to write the simulation results to a file at specific points in time, instead of writing output after each time step, which would be unwieldy. The code is:

```
void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      f << now;
      for (int i=0; i<nx; i++)
```

```
            f << "\ t" << cp[i].Report1("concentration Na+", "mmol/kg");
        f << std::endl;
        then += gap;
    }
}
```

Here, “f” is a reference to the output stream, “cp” is the origin of a vector of “nx” references to ChemPlugin instances, “gap” is the separation in years between output points, and “then” is a reference to a memory location in the calling program. The memory location holds the time level, in years, at which the next output event is to occur.

10.3.3 Simulation parameters

The simulation parameters are the numerical values that define the transport model.

```
// Simulation parameters.
int nx = 100; // number of instances along x
double length = 100; // cm
double deltax = length / nx; // cm
double deltay = 1.0, deltaz = 1.0; // cm
double porosity = 0.25;  // volume fraction

double diffcoef = 1e-6;  // cm2/s
double trans = deltay * deltaz * porosity * diffcoef / deltax;  // cm3/s

double xstable = 0.9;

double time_end = 15.0;  // years
double delta_years = time_end / 3; // years
double next_output = 0.0;  // years
```

Here, we note the length “deltax” of the instances is the domain size “length” divided by the number “nx” of ChemPlugin instances carried. The transmissivity “trans” is the product of the cross-sectional area “deltay * deltaz”, the porosity, and the diffusion coefficient “diffcoef”, divided by “deltax”.

Variable “xstable” holds the stability factor X_{stable}, and “time_end” is the duration of the simulation. The value set for “delta_years” is the gap between the output events, and “next_output” is the time level at which the next event is to be triggered.

10.3.4 Output file

The next block of code opens an output stream to a disk file and writes a line identifying the position along x at which each ChemPlugin instance will be positioned.

```
// Open output file and write instance positions on the first line.
std::ofstream f;
f.open("Diffusion.txt");
```

```
if (f.is_open()) {
   f << "years";
   for (int i=0; i<nx; i++)
      f << "\t" << (i+0.5) * deltax;
   f << std::endl;
}
else {
   std::cout << "Failed to open output file" << std::endl;
   return exit_client(-1);
}
```

10.3.5 Configuring and initializing instances

Next, the client program sets out to create, configure, and initialize each ChemPlugin instance making up the domain.

```
// Configure and initialize the instances.
std::string cmd0 = "volume = " + std::to_string(deltax * deltay * deltaz) +
                   " cm3; porosity = " + std::to_string(porosity) +
                   "; time end = " + std::to_string(time_end) +
                   " years; Xstable = " + std::to_string(xstable);
std::string cmd1 = "Na+ = 1 mmol/kg; Cl- = 1 mmol/kg";
std::string cmd2 = "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";

ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

for (int i=0; i<nx; i++) {
   cp[i].Config(cmd0);
   if (i < nx/2)
      cp[i].Config(cmd1);
   else
      cp[i].Config(cmd2);

   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
      return exit_client(-1);
   }
}
write_line(f, cp, nx, delta_years, next_output);
```

The first lines of the code set three character strings to hold ChemPlugin configuration commands: “cmd0”, “cmd1”, and “cmd2”. The former, “cmd0”, is to be applied to each of the instances, whereas “cmd1” pertains to instances on the left side of the domain, and “cmd2” to only the right-side instances.

The client next instantiates a vector of “nx” instances, but sets only the first to write console out messages, using the “banner” format to trace time stepping. If each of the instances had been set to produce console output, the result would be overwhelming and largely redundant. Finally, the client enters a loop in which it configures each ChemPlugin instance in the domain, and initializes it. If any of the instances does not initialize, the client exits with a non-zero status code.

10.3.6 Linking instances

The loop for linking the instances into a one-dimensional chain and setting transmissivity for each link is:

```
// Link the instances.
for (int i=1; i<nx; i++) {
   CpiLink link = cp[i].Link(cp[i-1]);
   link.Transmissivity(trans, "cm3/s");
}
```

The loop starts by linking the second instance (index 1) to the first (index 0), then links the third to the second, and so on. For each link, the client sets the corresponding transmissivity, as previously stored in “trans”.

Note the best practice of linking an instance to the instance behind it, rather than in front of it. Flow across a link in ChemPlugin is positive when it moves toward the instance that originated the link, away from the linked instance. By the convention of linking backward, then, flow is positive toward instances of increasing index.

10.3.7 Time marching loop

The time marching loop begins each pass by querying the instances for the preferred time step, as determined honoring the stability considerations outlined above. Then, taking the minimum of the steps reported, it steps forward in time, advances the transport equations, and advances the chemical equations. Finally, it passes the results to “write_line()” and returns to make another step.

The time marching code is:

```
// Time marching loop.
while (true) {
   double deltat = 1e99;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return exit_client(-1);
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceChemical()) return exit_client(-1);
```

```
    write_line(f, cp, nx, delta_years, next_output);
}
```

If any step in time stepping—advancing the time level, the transport equations, or the chemical equations—yields a non-zero return at any instance, the client program exits by issuing:

```
return exit_client(...);
```

In this way, the console remains open until the user touches the return key. The argument "..." is zero following a call to "AdvanceTimeStep()", since a non-zero return status commonly indicates the time marching is complete, rather than an error condition. Following calls to "AdvanceTransport()" or "AdvanceChemical()", on the other hand, the argument is -1, since a non-zero return from these functions indicates an error has occurred.

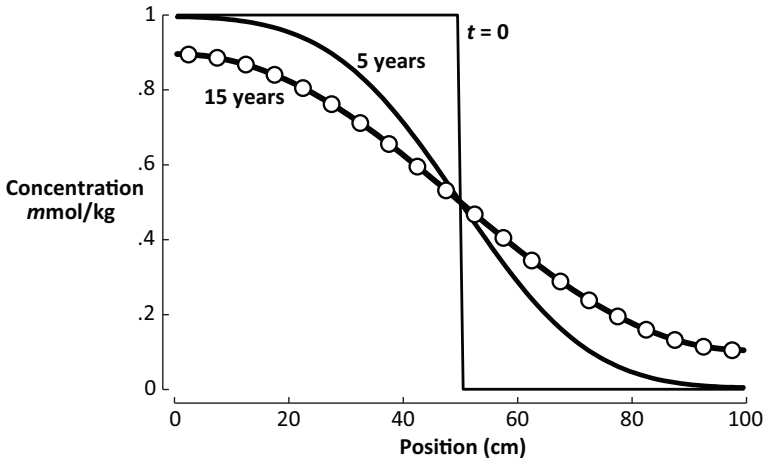
10.3.8 Running the client

Running the client produces the following

```
Model diffusion in one dimension

Solving for initial system.

 Loaded:    3 aqueous species,
            1 minerals,
            1 gases,
            0 surface species,
            4 elements,
            0 oxides.

Step   0, Xi = 0     (19 iterations)
 Charge balance: Cl- molality adjusted from 3.95 to .001

Step   1, Xi = .000845 (2 iterations)
Step   2, Xi = .00169 (2 iterations)
Step   3, Xi = .002535 (2 iterations)
Step   4, Xi = .00338 (2 iterations)
Step   5, Xi = .004225 (2 iterations)
... and so on ...
Step 1183, Xi = .9997 (3 iterations)
Step 1184, Xi = 1     (3 iterations)

Successful completion of reaction path.
```

on the console window.

Once the run is complete, opening file “Diffusion.txt” in a graphing program such as **Excel** lets you quickly plot the salinity profile across the domain at various points in the simulation. The plot below shows the calculation results after 5 years and 15 years.



The circles in the diagram correspond to the analytic solution to the diffusion equation at t = 15 years, from Carslaw and Jaeger’s 1959 text, demonstrating correctness of the client’s results.

10.3.9 C++ source code

The complete C++ source code may be downloaded as file “Diffusion1.cpp” from the ChemPlugin.GWB.com website, and is listed below.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      f << now;
      for (int i=0; i<nx; i++)
```

```
            f << "\t" << cp[i].Report1("concentration Na+", "mmol/kg");
        f << std::endl;
        then += gap;
    }
}

int main(int argc, char** argv) {
    std::cout << "Model diffusion in one dimension"
              << std::endl << std::endl;

    // Simulation parameters.
    int nx = 100; // number of instances along x
    double length = 100; // cm
    double deltax = length / nx; // cm
    double deltay = 1.0, deltaz = 1.0; // cm
    double porosity = 0.25;  // volume fraction

    double diffcoef = 1e-6;  // cm2/s
    double trans = deltay * deltaz * porosity * diffcoef / deltax;  // cm3/s

    double xstable = 0.9;

    double time_end = 15.0;  // years
    double delta_years = time_end / 3; // years
    double next_output = 0.0;  // years

    // Open output file and write instance positions on the first line.
    std::ofstream f;
    f.open("Diffusion.txt");
    if (f.is_open()) {
        f << "years";
        for (int i=0; i<nx; i++)
            f << "\t" << (i+0.5) * deltax;
        f << std::endl;
    }
    else {
        std::cout << "Failed to open output file" << std::endl;
        return exit_client(-1);
    }

    // Configure and initialize the instances.
    std::string cmd0 = "volume = " + std::to_string(deltax * deltay * deltaz) +
                       " cm3; porosity = " + std::to_string(porosity) +
                       "; time end = " + std::to_string(time_end) +
                       " years; Xstable = " + std::to_string(xstable);
    std::string cmd1 = "Na+ = 1 mmol/kg; Cl- = 1 mmol/kg";
    std::string cmd2 = "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";
```

```
    ChemPlugin *cp = new ChemPlugin[nx];
    cp[0].Console("stdout");
    cp[0].Config("pluses = banner");

    for (int i=0; i<nx; i++) {
        cp[i].Config(cmd0);
        if (i < nx/2)
            cp[i].Config(cmd1);
        else
            cp[i].Config(cmd2);

        if (cp[i].Initialize()) {
            std::cout << "Instance " << i << " failed to initialize" << std::endl;
            return exit_client(-1);
        }
    }
    write_line(f, cp, nx, delta_years, next_output);

    // Link the instances.
    for (int i=1; i<nx; i++) {
        CpiLink link = cp[i].Link(cp[i-1]);
        link.Transmissivity(trans, "cm3/s");
    }

    // Time marching loop.
    while (true) {
        double deltat = 1e99;
        for (int i=0; i<nx; i++)
            deltat = std::min(deltat, cp[i].ReportTimeStep());
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTransport()) return exit_client(-1);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) return exit_client(-1);

        write_line(f, cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Advection-Dispersion Model

In this chapter, we build on the ideas in the previous two chapters to create a model of simultaneous advective and diffusive transport. We begin by considering issues of numerical stability and then lay out a client program that uses ChemPlugin to solve the advection-dispersion equation in one dimension.

11.1 Numerical stability

The stability criterion for solving the advection-dispersion equation using a finite volume scheme follows from the Courant condition and von Neumann's analysis, as presented in the previous two chapters. In a three-dimensional Cartesian domain, the time step Δt must satisfy the inequality

$$\left(\frac{|v_x|}{\Delta x} + \frac{|v_y|}{\Delta y} + \frac{|v_z|}{\Delta z} + \frac{2D_x}{n\Delta x^2} + \frac{2D_y}{n\Delta y^2} + \frac{2D_z}{n\Delta z^2} \right) \Delta t \leq 1 \tag{11.1}$$

to ensure stability. Here, v_x, v_y, and v_z are the fluid velocities along the principal coordinates; D_x, D_y, and D_z are the Fickian coefficients; Δx, Δy, and Δz are the dimensions of the finite volume; and n is porosity, which is one for open-channel flow.

Multiplying this equation as before by $V_f = n\Delta x \Delta y \Delta z$, we note that the terms $A_x D_x/\Delta x$, $A_y D_y/\Delta y$, and $A_z D_z/\Delta z$ correspond to transmissivities τ_ℓ across the links ℓ in the associated direction. As well, the terms $A_x v_x/n$, $A_y v_y/n$, and $A_z v_z/n$ are the fluxes Q_ℓ across the links. The limiting time step, which we denote Δt_1, then, can be calculated according to the equation

$$\Delta t_1 = V_f \Big/ \left(\sum_{\ell : Q_\ell > 0} Q_\ell + \sum_\ell \tau_\ell \right) \tag{11.2}$$

which combines the results in the previous two chapters into a single equation.

ChemPlugin calculates a second limiting time step Δt_2

$$\Delta t_2 = C_{\mathrm{o}}^{\lim} V_f \Big/ \left(\sum_{\ell : Q_\ell > 0} Q_\ell \right) \tag{11.3}$$

to account for the possibility that a value of $C_{\mathrm{o}}^{\mathrm{lim}} \neq 1$ has been set by the client program, as well as a value Δt_3

$$\Delta t_3 = X_{\mathrm{stable}} V_f \Big/ \left(\sum_{\ell} \tau_{\ell} \right) \tag{11.4}$$

to account for non-default settings for X_{stable}. The limiting step reported by "ReportTimeStep()"

$$\Delta t = \min(\Delta t_1, \Delta t_2, \Delta t_3) \tag{11.5}$$

is the least of the three values.

11.2 Advection-dispersion model

In this section, we lay out a client program that traces advection and dispersion over time in a single dimension.

11.2.1 Program structure

The client structure is the same as that for the program we used to trace diffusion:

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                  double gap, double& then)
{
   ... output function goes here ...
}

int main(int argc, char** argv) {
   std::cout << "Model advection-dispersion in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   ... simulation parameters are set out here ...
```

```
    // Open output file and write instance positions on the first line.
    ... output file is opened and initialized here ...

    // Configure and initialize the inlet and interior instances.
    ... instances are created, configured, and initialized here ...

    // Link the instances.
    ... links among the instances are created and defined here ...

    // Time marching loop.
    ... time marching loop goes here ...

    // Never gets here.
    return 0;
}
```

We explain here only those sections of the program that differ from program "Diffusion1.cpp", the client we developed in the previous chapter.

11.2.2 Simulation parameters

The simulation parameters carried at the top of the program parallel the parameters in the diffusion model, with the addition of terms reflecting the passage of fluid through the domain:

```
    // Simulation parameters.
    int nx = 400; // number of instances along x
    double length = 100; // m
    double deltax = length / nx; // m
    double deltay = 1.0, deltaz = 1.0; // m
    double porosity = 0.25;  // volume fraction

    double veloc_in; // m/yr
    std::cout << "Please enter fluid velocity in m/yr: ";
    std::cin >> veloc_in;
    std::cin.ignore();

    double velocity = veloc_in / 31557600.;  // m/s
    double flow = deltay * deltaz * porosity * velocity; // m3/s

    double diffcoef = 1e-10;  // m2/s
    double dispersivity = 1.0; // m
    double dispcoef = velocity * dispersivity + diffcoef; // m2/s
    double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

    double time_end = 10.0;  // years
    double delta_years = time_end / 5; // years
    double next_output = 0.0;  // years
```

Here, the client queries the user for the fluid velocity v_x, which it uses to figure the flow rate Q_x

$$Q_x = A_x \, n \, v_x = \Delta y \, \Delta z \, n \, v_x \tag{11.6}$$

The transmissivity τ_x, in turn, is given by

$$\tau_x = \Delta y \, \Delta z \, n \, \left(\alpha v_x + D^*\right) \tag{11.7}$$

from the dispersivity α and diffusion coefficient D^*.

11.2.3 Configure and initialize instances

The client creates a ChemPlugin instance "cp_inlet" to represent the composition of the inlet fluid, as well as an array "cp" of "nx" ChemPlugin instances to represent discrete segments of the domain. The client configures each instance with member function "Config()" and initializes it with function "Initialize()", trapping any return status indicating failure.

The code is:

```
// Configure and initialize the inlet and interior instances.
ChemPlugin cp_inlet;
cp_inlet.Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg");
if (cp_inlet.Initialize()) {
   std::cout << "Inlet failed to initialize" << std::endl;
   return exit_client(-1);
}

ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg; "
                  "volume = " + std::to_string(deltax * deltay * deltaz) +
                  " m3; porosity = " + std::to_string(porosity) +
                  "; time end = " + std::to_string(time_end) + " years";

for (int i=0; i<nx; i++) {
   cp[i].Config(cmd);
   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
      return exit_client(-1);
   }
}
write_line(f, cp, nx, delta_years, next_output);
```

Here, we have set a 1 mmol kg^{-1} inlet fluid, and a dilute fluid within the domain.

In the case of the domain, we are careful to set the system extent with the “volume” and “porosity” configuration commands. As well, we prescribe the end point for the time marching, using the “time end” command. We could do the same for the inlet, but our efforts would be wasted, since the inlet serves only as a marker for the chemistry of the inflow and hence remains static over the simulation.

11.2.4 Link the instances

The client links the ChemPlugin instances in three steps. It links the left-side instance (index 0) to the inlet “cp_inlet”. Then, it links each remaining instance (index 1, 2, . . .) to the instance at its immediate left. Finally, it sets a free outlet for the right-side instance (index nx-1). For each link, the client sets a flow rate and transmissivity.

The code is:

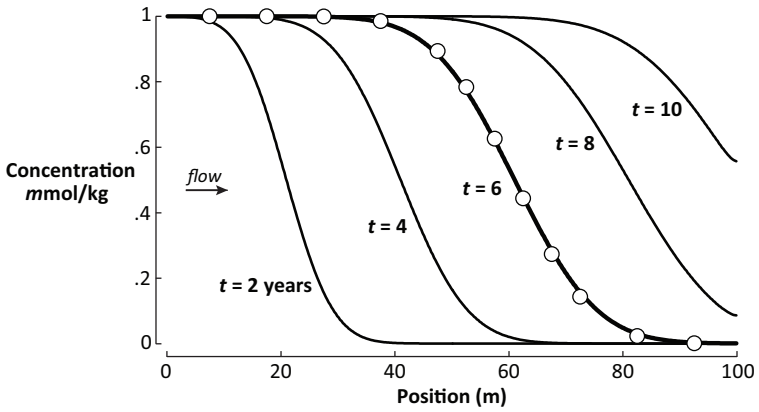
```
// Link the instances.
CpiLink link = cp[0].Link(cp_inlet);
link.Transmissivity(trans, "m3/s");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
   link = cp[i].Link(cp[i-1]);
   link.Transmissivity(trans, "m3/s");
   link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");
```

Note that, except for the free outlet, we have followed the best practice of originating each link from the instance on the right side of the pair. Flow across a link is positive along increasing x, then, and variable “flow” gives the flow rate directly. If we had linked the instances in the opposite sense, we would have needed to negate “flow” in our calls to “FlowRate()”.

The free outlet on the right of the domain is a special case, since the open link is, of necessity, created by the instance at lesser x. Now, we must set the flow rate to “-flow”. Note also that we set no transmissivity for the open link, since first-order transport across a free outlet is not possible.

11.3 Running the model

Running the model yields console output similar to that produced by the diffusion model in the previous chapter, and generates an output file “Advection.txt” that you can open in **Excel** or another graphing application. The plot below shows the salinity profile across the domain predicted at two year increments



corresponding to a fluid velocity v_x of 10 m yr^{-1}. The circles in the diagram correspond to the analytic solution to the advection-dispersion equation at t = 6 years, from page 373 of Domenico and Schwartz's 1998 text.

The small differences between the exact solution and that predicted by our client program are attributable in the most part to numerical dispersion. The effect of numerical dispersion can be demonstrated by decreasing or increasing the number "nx" of ChemPlugin instances, to coarsen or refine the discretization. Raising the number of instances lowers the dispersion and hence improves the accuracy of the calculated results.

11.4 C++ source code

The complete C++ source code may be downloaded as file "Advection1.cpp" from the ChemPlugin.GWB.com website, and is listed below.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
```

```
        f << now;
        for (int i=0; i<nx; i++)
            f << "\t" << cp[i].Report1("concentration Na+", "mmol/kg");
        f << std::endl;
        then += gap;
    }
}

int main(int argc, char** argv) {
    std::cout << "Model advection-dispersion in one dimension"
              << std::endl << std::endl;

    // Simulation parameters.
    int nx = 400; // number of instances along x
    double length = 100; // m
    double deltax = length / nx; // m
    double deltay = 1.0, deltaz = 1.0; // m
    double porosity = 0.25;  // volume fraction

    double veloc_in; // m/yr
    std::cout << "Please enter fluid velocity in m/yr: ";
    std::cin >> veloc_in;
    std::cin.ignore();

    double velocity = veloc_in / 31557600.;  // m/s
    double flow = deltay * deltaz * porosity * velocity; // m3/s

    double diffcoef = 1e-10;  // m2/s
    double dispersivity = 1.0; // m
    double dispcoef = velocity * dispersivity + diffcoef; // m2/s
    double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

    double time_end = 10.0;  // years
    double delta_years = time_end / 5; // years
    double next_output = 0.0;  // years

    // Open output file and write instance positions on the first line.
    std::ofstream f;
    f.open("Advection.txt");
    if (f.is_open()) {
        f << "years";
        for (int i=0; i<nx; i++)
            f << "\t" << (i+0.5) * deltax;
        f << std::endl;
    }
    else {
        std::cout << "Failed to open output file" << std::endl;
        return exit_client(-1);
```

```
}

// Configure and initialize the inlet and interior instances.
ChemPlugin cp_inlet;
cp_inlet.Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg");
if (cp_inlet.Initialize()) {
   std::cout << "Inlet failed to initialize" << std::endl;
   return exit_client(-1);
}

ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg; "
                  "volume = " + std::to_string(deltax * deltay * deltaz) +
                  " m3; porosity = " + std::to_string(porosity) +
                  "; time end = " + std::to_string(time_end) + " years";

for (int i=0; i<nx; i++) {
   cp[i].Config(cmd);
   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
      return exit_client(-1);
   }
}
write_line(f, cp, nx, delta_years, next_output);

// Link the instances.
CpiLink link = cp[0].Link(cp_inlet);
link.Transmissivity(trans, "m3/s");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
   link = cp[i].Link(cp[i-1]);
   link.Transmissivity(trans, "m3/s");
   link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");

// Time marching loop.
while (true) {
   double deltat = 1e99;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   for (int i=0; i<nx; i++)
```

```
            if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTransport()) return exit_client(-1);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) return exit_client(-1);

        write_line(f, cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Heat Transfer

A ChemPlugin instance can trace how its temperature varies over the course of a simulation by keeping track of the net accumulation or loss of heat energy during the time marching procedure. The calculation accounts for the transfer of thermal energy across links by advecting fluids and by heat conduction, as well as the effects of any internal heat sources or sinks the client may have set.

To trace heat transfer over the course of a simulation, the client program sets the flow rate across each link using the “FlowRate()” member function. If heat conduction is to be considered, the client further uses the “HeatTrans()” member function to set thermal transmissivity across each link; the thermal transmissivity is described below. Then, at each time step within the time marching loop, the client calls member function “AdvanceHeatTransport()” to trigger the temperature change calculation.

A client may, as an alternative, prescribe an instance’s temperature evolution explicitly. Temperature at an instance can be set to hold steady over the simulation, or to slide linearly from an initial to a final value. Or, the client program can directly update temperature at each step in the time marching loop, using values it determines independently.

12.1 Initial temperature

A client program uses the “Config()” member function to set an instance’s initial temperature. For example, the statement

```
cp.Config("temperature = 25 C");
```

or

```
cp.Config("T = 298 K");
```

prescribes room temperature as the initial state for ChemPlugin instance “cp”.

The client can further use the “Config()” member function to specify that the instance hold steady

```
cp.Config("temperature = 60 C isothermal");
```

at a certain temperature, using the "isothermal" keyword, or to slide

```
cp.Config("T initial = 25 C, final = 60 C");
```

from one temperature to another over the course of the simulation, using the "initial" and "final" keywords.

To trace a polythermal simulation in which conduction and advection among linked instances transports heat energy into and out of an instance, you specify the instance's initial temperature as above, and use the "span" configuration command to prescribe a temperature range for the calculation. The instance will load only chemical species whose stabilities are known across the prescribed range, and "AdvanceHeatTransport()" will report a failure condition if the updated temperature deviates outside the range.

As an example, the commands

```
cp.Config("T = 25 C; span 25 C to 100 C");
```

initiates the instance at 25°C and sets an allowed temperature range up to 100°C. Aqueous species, minerals, and so on for which log Ks in the thermo dataset do not bracket the temperatures will not be considered. As well, if a call to "AdvanceHeatTransport()" drives temperature more than a few degrees below 25°C or above 100°C, the function will report failure.

12.2 Temperature calculation

When a client program calls member function "AdvanceHeatTransport()" within a time marching loop, the ChemPlugin instance evaluates the change in temperature over the time step due to advective transport, heat conduction, and internal heat sources.

Specifically, the function enters into the heat transfer calculation when, at the instance in question, each of the following conditions are met:

- The client has not set the isothermal option, using the "isothermal" keyword of the "temperature" configuration command.
- The client has not specified a sliding temperature path with the "initial" and "final" keywords of the "temperature" command.
- Temperature at the instance differs from that at any instance linked to it, or a heat source has been specified, or both.

If any of the conditions are not met, "AdvanceHeatTransport()" returns with temperature unaltered, or adjusted according to the sliding temperature feature.

12.2.1 Advective transfer

If Q is the flow rate in $m^3\ s^{-1}$ across a link from one ChemPlugin instance to another, the rate J_T^o of advective heat transfer between the instances is given by

$$J_T^o = \rho_w C_w Q T \tag{12.1}$$

in units of J s^{-1}. Here, ρ_w is the fluid density in kg m^{-3}, C_w is fluid heat capacity in J kg^{-1} K^{-1}, and T is temperature in K.

A client program uses the "FlowRate()" member function to set the flow rate Q across a link, as described in previous chapters of this User's Guide. Once Q is specified, the instance computes the effects of advective transport whenever the client program calls member function "AdvanceHeatTransport()", assuming the conditions listed above are met.

12.2.2 Conductive transfer

Fourier's law gives the conductive heat flux J_T^1 in J s^{-1} across an arbitrary plane as

$$J_T^1 = -A K_T \frac{dT}{dx} \tag{12.2}$$

In this equation, A is cross-sectional area, in m^2; K_T is thermal conductivity, in W m^{-1} K^{-1} (or, equivalently, J m^{-1} s^{-1} K^{-1}, since 1 W = 1 J s^{-1}); and dT/dx is the temperature gradient across the plane, in K m^{-1}.

ChemPlugin calculates J_T^1 according to the approximate equation

$$J_T^1 \approx -\tau_T \left(T^j - T^{\text{linked}}\right) \tag{12.3}$$

where τ_T is the thermal transmissivity in units of W K^{-1}, and T^j and T^{linked} are temperatures at the originating and linked instances, respectively, in K.

Calculation of the thermal transmissivity τ_T closely parallels determination of the transmissivities τ for mass transport, as described in the **Diffusion and Dispersion** chapter. Where the ChemPlugin instances represent equally-sized prisms of equivalent thermal conductivities, the thermal transmissivity is given simply as

$$\tau_T = \frac{A K_T}{\Delta x} \tag{12.4}$$

For the case of equal instance sizes but heterogeneous thermal conductivity, the equation takes the form

$$\tau_T = \left(\frac{2A}{\Delta x}\right) \frac{K_T^j K_T^{\text{linked}}}{K_T^j + K_T^{\text{linked}}} \tag{12.5}$$

where K_T^j and K_T^{linked} are conductivity at the originating and linked instances. Finally, the thermal transmissivity is given

$$\tau_T = \frac{2\left(\frac{AK_T}{\Delta x}\right)^j \left(\frac{AK_T}{\Delta x}\right)^{\text{linked}}}{\left(\frac{AK_T}{\Delta x}\right)^j + \left(\frac{AK_T}{\Delta x}\right)^{\text{linked}}} \tag{12.6}$$

for the general case of arbitrary geometry and heterogeneous thermal conductivity.

12.2.3 Heat sources

The client can set within any ChemPlugin instance a heat source or sink, using the "heat_source" configuration command. For example:

```
cp.Config("heat_source = 5e-6 W/m3");
```

The source is expressed as a rate of heat supply per unit bulk volume of the instance. Setting a positive value creates a source, whereas a negative value serves as a sink.

12.2.4 Stability

In tracing heat transfer, as was the case for mass transport, the size of the time steps that can be taken during the simulation procedure are limited by the need to maintain numerical stability. Following the logic in the previous chapter, the equation

$$\Delta t = V_f \Big/ \left(\sum_{\ell: Q_\ell > 0} Q_\ell + \frac{1}{C_b} \sum_\ell \tau_{T_\ell} \right) \tag{12.7}$$

gives the largest time step at which a ChemPlugin instance honors the stability constraints on tracing heat conduction and advection. Here, Q_ℓ are the flow rates across each of the instance's links ℓ, in $m^3\ s^{-1}$; C_b is the instance's bulk heat capacity, in $J\ m^{-3}\ K^{-1}$; and τ_{T_ℓ} are the links' thermal transmissivities, in $W\ K^{-1}$.

12.2.5 Time marching loop

Member function "AdvanceHeatTransport()" is commonly called within the time marching loop

```
// Time marching loop.
while (true) {
   double deltat = 1e99;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat)) return -1;
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return -1;
```

```
    for (int i=0; i<nx; i++)
      if (cp[i].AdvanceHeatTransport()) return -1;
    for (int i=0; i<nx; i++)
      if (cp[i].AdvanceChemical()) return -1;
  }
```

after advancing the mass transport equations, but before evaluating the chemical equations.

12.3 Externally prescribed temperature

A notable option for tracing polythermal simulations is for the client program to prescribe how the temperature of each ChemPlugin instance varies, according to its own logic. To do so, once an instance has been initialized, a client program makes use of member function "SlideTemperature()" to adjust the instance's temperature.

Suppose within a client program a function

```
double my_temperature(int i) { ... };
```

is coded to return temperature for ChemPlugin instance "i" at any point in a simulation. In this case, the time marching loop above could be cast as:

```
  // Time marching loop.
  while (true) {
    double deltat = 1e99;
    for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
    for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat)) return -1;
    for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return -1;
    for (int i=0; i<nx; i++)
      if (cp[i].SlideTemperature(my_temperature(i))) return -1;
    for (int i=0; i<nx; i++)
      if (cp[i].AdvanceChemical()) return -1;
  }
```

Here, in the second to last loop, where we had previously made a call to member function "AdvanceHeatTransport()", we instead call "SlideTemperature()" to set temperature directly.

12.4 Model of heat conduction

In light of the direct analog between Fourier's law of heat conduction and Fick's law of diffusion, our model of heat conduction is quite similar to the diffusion model we developed previously, in the **Diffusion and Dispersion** chapter.

Here, we will emphasize the differences between the two client programs. The full C++ code is listed in the final section of this chapter.

12.4.1 Simulation parameters

There are two primary differences between the simulation parameters for the diffusion and heat conduction models. First, in the heat conduction model, the domain is 100 m long, rather than 100 cm.

```
// Simulation parameters.
int nx = 100;  // number of instances along x
double length = 100;  // m
double deltax = length / nx;  // m
double deltay = 1.0, deltaz = 1.0;  // m
```

The difference in domain length reflects the fact that in a given length of time, heat is conducted through rock farther than solute diffuses.

Second, the transmissivity variable "trans" represents the thermal rather than mass transmissivity. As such, it is defined in terms of the thermal conductivity "tcon"

```
double tcond = 2.0;  // W/m/K
double trans = deltay * deltaz * tcond / deltax;  // W/K
```

rather than a diffusion coefficent.

12.4.2 Configuring and initializing instances

In the diffusion model, we set solute concentration on the left half of the domain larger than on the right side. In the heat conduction model, in contrast, we set temperature to the left higher than to the right. The code is:

```
// Configure and initialize the instances.
ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "span 20 C to 100 C; "
                  "volume = " + std::to_string(deltax * deltay * deltaz) + " m3; "
                  "porosity = " + std::to_string(porosity) + "; "
                  "time end = " + std::to_string(time_end) + " years; "
                  "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";

for (int i=0; i<nx; i++) {
   cp[i].Config(cmd);
   if (i < nx/2)
      cp[i].Config("T = 100 C");
   else
      cp[i].Config("T = 20 C");
```

```
    if (cp[i].Initialize()) {
        std::cout << "Instance " << i << " failed to initialize" << std::endl;
        return exit_client(-1);
    }
}
```

12.4.3 Linking instances

In linking the ChemPlugin instances that make up the domain, the heat conduction model differs from the diffusion model in that we use the “HeatTrans()” member function to set thermal transmissivity:

```
// Link the instances.
for (int i=1; i<nx; i++) {
    CpiLink link = cp[i].Link(cp[i-1]);
    link.HeatTrans(trans, "W/K");
}
```

In the diffusion model, we instead specified the mass transmissivity with the “Transmissivity()” member function.

12.4.4 Time marching loop

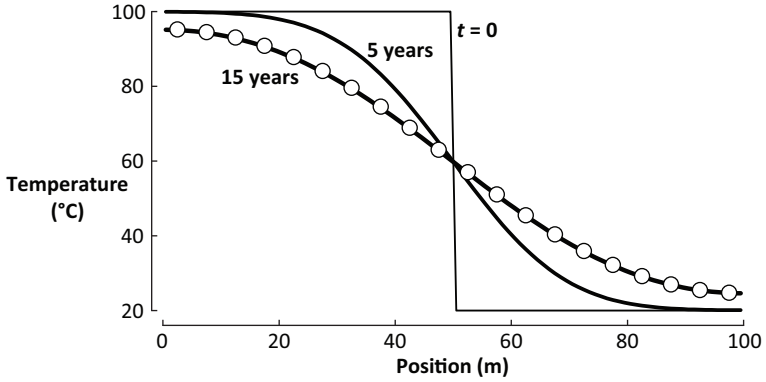
The time marching loop in the heat conduction example includes a call to member function “AdvanceHeatTransport()” for each ChemPlugin instance:

```
// Time marching loop.
while (true) {
    double deltat = 1e99;
    for (int i=0; i<nx; i++)
        deltat = std::min(deltat, cp[i].ReportTimeStep());
    for (int i=0; i<nx; i++)
        if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
    for (int i=0; i<nx; i++)
        if (cp[i].AdvanceHeatTransport()) return exit_client(-1);
    for (int i=0; i<nx; i++)
        if (cp[i].AdvanceChemical()) return exit_client(-1);

    write_line(f, cp, nx, delta_years, next_output);
}
```

In the diffusion model, we instead called member function “AdvanceTransport()” to evaluate the equations describing mass transport, rather than heat transport.

12.4.5 Running the client

Running the heat conduction model produces a file "HeatConduction.txt" containing temperature profiles across the domain at discrete points in time. Plotting the results for 5 years and 15 years gives:



The circles in the diagram correspond to the analytic solution to the problem at t = 15 years, from Carslaw and Jaeger's 1959 textbook.

12.5 Model of advective heat transfer

In this section, we develop a model of the simultaneous advection and conduction of heat. The model closely parallels the model of advective mass transport presented in the previous chapter, **Advection-Dispersion Model**. As such, we will limit our discussion to differences between the two models; the full source code for our new model is listed in the final section of this chapter.

12.5.1 Simulation parameters

Just as in the heat conduction model above, the simulation parameters differ in that here we set a thermal transmissivity in terms of the thermal conductivity:

```
double tcond = 2.0;  // W/m/K
double trans = deltay * deltaz * tcond / deltax;  // W/K
```

In the mass transport model from the previous chapter, we set instead a mass transmissivity representing the processes dispersion and diffusion.

12.5.2 Configuring and initializing instances

For the current model, we set temperature at the inlet to 100 °C, and within the initial domain to 20 °C. The fluid composition everywhere is the same.

```
ChemPlugin cp_inlet;
cp_inlet.Config("T = 100 C; Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg");
if (cp_inlet.Initialize()) {
```

```
    std::cout << "Inlet failed to initialize" << std::endl;
    return exit_client(-1);
}

ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "T = 20 C; span 20 C to 100 C; volume = " +
                  std::to_string(deltax * deltay * deltaz) + " m3; "
                  "porosity = " + std::to_string(porosity) + "; "
                  "time end = " + std::to_string(time_end) + " years; "
                  "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";

for (int i=0; i<nx; i++) {
    cp[i].Config(cmd);
    if (cp[i].Initialize()) {
        std::cout << "Instance " << i << " failed to initialize" << std::endl;
        return exit_client(-1);
    }
}
```

12.5.3 Linking instances

In linking the instances, we use member function “HeatTrans()” to define conduction among the instances.

```
CpiLink link = cp[0].Link(cp_inlet);
link.HeatTrans(trans, "W/K");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
    link = cp[i].Link(cp[i-1]);
    link.HeatTrans(trans, "W/K");
    link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");
```

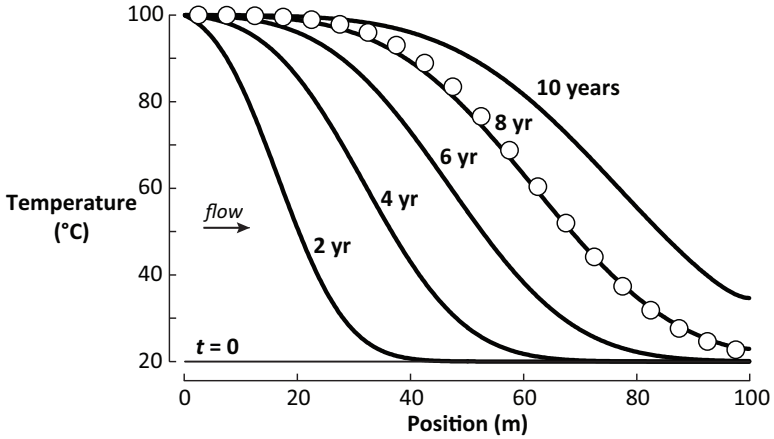
We are not considering mass transport, so we do not set mass transmissivities, as we did in the previous chapter’s model.

12.5.4 Time marching loop

The time marching loop is the same as coded in the heat conduction model earlier in this chapter.

12.5.5 Running the client

Running the model generates an output file "HeatTransfer.txt" containing temperature profiles across the domain at discrete points in time. The plot below shows temperature predicted by the model at two year increments



calculated assuming a fluid velocity v_x of 20 m yr^{-1}. The circles in the diagram correspond to the analytic solution at t = 8 years. The small discrepancies between the numerical and analytic results reflect in large part the fact that ChemPlugin accounts for how fluid properties such as density vary with temperature, whereas the closed-form mathematical solution cannot.

12.6 C++ source code

The C++ source codes for the two example client programs in this chapter are given in this section.

12.6.1 Heat conduction code

The source code for the heat conduction example can be downloaded from the ChemPlugin.GWB.com website as file "HeatConduction1.cpp", and is listed below.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}
```

```
void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      f << now;
      for (int i=0; i<nx; i++)
         f << "\t" << cp[i].Report1("temperature", "C");
      f << std::endl;
      then += gap;
   }
}

int main(int argc, char** argv) {
   std::cout << "Model heat conduction in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   int nx = 100;  // number of instances along x
   double length = 100;  // m
   double deltax = length / nx;  // m
   double deltay = 1.0, deltaz = 1.0;  // m
   double porosity = 0.25;  // volume fraction

   double tcond = 2.0;  // W/m/K
   double trans = deltay * deltaz * tcond / deltax;  // W/K

   double time_end = 15.0;  // years
   double delta_years = time_end / 3;  // years
   double next_output = 0.0;  // years

   // Open output file and write instance positions on the first line.
   std::ofstream f;
   f.open("HeatConduction.txt");
   if (f.is_open()) {
      f << "years";
      for (int i=0; i<nx; i++)
         f << "\t" << (i+0.5) * deltax;
      f << std::endl;
   }
   else {
      std::cout << "Failed to open output file" << std::endl;
      return exit_client(-1);
   }

   // Configure and initialize the instances.
   ChemPlugin *cp = new ChemPlugin[nx];
   cp[0].Console("stdout");
```

```
    cp[0].Config("pluses = banner");

    std::string cmd = "span 20 C to 100 C; "
                      "volume = " + std::to_string(deltax * deltay * deltaz) + " m3; "
                      "porosity = " + std::to_string(porosity) + "; "
                      "time end = " + std::to_string(time_end) + " years; "
                      "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";

    for (int i=0; i<nx; i++) {
       cp[i].Config(cmd);
       if (i < nx/2)
          cp[i].Config("T = 100 C");
       else
          cp[i].Config("T = 20 C");

       if (cp[i].Initialize()) {
          std::cout << "Instance " << i << " failed to initialize" << std::endl;
          return exit_client(-1);
       }
    }
    write_line(f, cp, nx, delta_years, next_output);

    // Link the instances.
    for (int i=1; i<nx; i++) {
       CpiLink link = cp[i].Link(cp[i-1]);
       link.HeatTrans(trans, "W/K");
    }

    // Time marching loop.
    while (true) {
       double deltat = 1e99;
       for (int i=0; i<nx; i++)
          deltat = std::min(deltat, cp[i].ReportTimeStep());
       for (int i=0; i<nx; i++)
          if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
       for (int i=0; i<nx; i++)
          if (cp[i].AdvanceHeatTransport()) return exit_client(-1);
       for (int i=0; i<nx; i++)
          if (cp[i].AdvanceChemical()) return exit_client(-1);

       write_line(f, cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

12.6.2 Advective heat transfer code

The source code for the heat advection example can be downloaded from the ChemPlugin.GWB.com website as file “HeatTransfer1.cpp”, and is listed below.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void write_line(std::ofstream& f, ChemPlugin *cp, int nx,
                double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      f << now;
      for (int i=0; i<nx; i++)
         f << "\t" << cp[i].Report1("temperature", "C");
      f << std::endl;
      then += gap;
   }
}

int main(int argc, char** argv) {
   std::cout << "Model heat transfer in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   int nx = 400; // number of instances along x
   double length = 100; // m
   double deltax = length / nx; // m
   double deltay = 1.0, deltaz = 1.0; // m
   double porosity = 0.25;  // volume fraction

   double tcond = 2.0;  // W/m/K
   double trans = deltay * deltaz * tcond / deltax;  // W/K

   double veloc_in; // m/yr
   std::cout << "Please enter fluid velocity in m/yr: ";
   std::cin >> veloc_in;
```

```
std::cin.ignore();

double velocity = veloc_in / 31557600.;  // m/s
double flow = deltay * deltaz * porosity * velocity;  // m3/s

double time_end = 10.0;  // years
double delta_years = time_end / 5;  // years
double next_output = 0.0;  // years

// Open output file and write instance positions on the first line.
std::ofstream f;
f.open("HeatTransfer.txt");
if (f.is_open()) {
   f << "years";
   for (int i=0; i<nx; i++)
      f << "\t" << (i+0.5) * deltax;
   f << std::endl;
}
else {
   std::cout << "Failed to open output file" << std::endl;
   return exit_client(-1);
}

// Configure and initialize the inlet and interior instances.
ChemPlugin cp_inlet;
cp_inlet.Config("T = 100 C; Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg");
if (cp_inlet.Initialize()) {
   std::cout << "Inlet failed to initialize" << std::endl;
   return exit_client(-1);
}

ChemPlugin *cp = new ChemPlugin[nx];
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "T = 20 C; span 20 C to 100 C; volume = " +
                  std::to_string(deltax * deltay * deltaz) + " m3; "
                  "porosity = " + std::to_string(porosity) + "; "
                  "time end = " + std::to_string(time_end) + " years; "
                  "Na+ = 0.001 mmol/kg; Cl- = 0.001 mmol/kg";

for (int i=0; i<nx; i++) {
   cp[i].Config(cmd);
   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
      return exit_client(-1);
   }
}
```

```
    write_line(f, cp, nx, delta_years, next_output);

    // Link the instances.
    CpiLink link = cp[0].Link(cp_inlet);
    link.HeatTrans(trans, "W/K");
    link.FlowRate(flow, "m3/s");

    for (int i=1; i<nx; i++) {
        link = cp[i].Link(cp[i-1]);
        link.HeatTrans(trans, "W/K");
        link.FlowRate(flow, "m3/s");
    }

    link = cp[nx-1].Link();
    link.FlowRate(-flow, "m3/s");

    // Time marching loop.
    while (true) {
        double deltat = 1e99;
        for (int i=0; i<nx; i++)
            deltat = std::min(deltat, cp[i].ReportTimeStep());
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceHeatTransport()) return exit_client(-1);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) return exit_client(-1);

        write_line(f, cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Reactive Transport Model

In this final chapter, we culminate our modeling exercises by creating a one-dimensional model of polythermal reactive transport out of ChemPlugin instances. Our model is similar to the client program "Advection1.cpp" that we wrote in the **Advection-Dispersion Model** chapter. We set a domain of the same medium geometry as in that code, and we again query the user for the fluid velocity.

Unlike "Advection1.cpp", however, we ask the user to point to an input file of configuration commands. The commands constrain the chemistry of the inlet fluid, as well as the domain's initial chemistry. The input file has the structure:

```
scope inlet
   ... configuration commands applied to the inlet ...
scope initial
   ... configuration commands applied to the initial domain ...
```

Configuration commands following a "scope inlet" statement apply to the inlet fluid, whereas those following "scope initial" are used to constrain the chemistry of the domain.

13.1 Program structure

The structure of the client program is similar to our previous clients:

```
#include <iostream>
#include <fstream>
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void open_input(std::ifstream& input, int argc, char** argv) {
   ... function to open input file goes here ...
```

```
}

void write_results(ChemPlugin *cp, int nx, double gap, double& then)
{
   ... function to write modeling results goes here ...
}

int main(int argc, char** argv) {
   std::cout << "Model reactive transport in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   ... simulation parameters are set out here ...

   // Create the inlet and interior instances.
   ... set out instances representing inlet and domain here ...

   // Query user for input file and configure inlet, initial domain.
   ... read the input file and configure the instances here ...

   // Initialize the inlet and interior instances; write out initial conditions.
   ... instances are initialized here ...

   // Query user for velocity; calculate flow rate and transmissivities.
   ... read in the velocity and calculate transport parameters here ...

   // Link the instances.
   ... links among the instances are created and defined here ...

   // Time marching loop.
   ... time marching loop goes here ...

   // Never gets here.
   return 0;
}
```

Function "open_input()" is the same as previous clients; the remainder of the code is explained below.

13.2 Output function

The client's output strategy is to write blocks of the calculation results in "print format" to a file "RTM.txt". Initially, the client writes blocks describing the inlet fluid and the initial state of each of the ChemPlugin instances comprising the domain. Then, every so often over the course of the time marching, the client scans across the domain, writing a block of output for each of the ChemPlugin instances.

Function “write_results()” writes out the calculation results for each instance in the domain, once every “gap” years:

```
void write_results(ChemPlugin *cp, int nx, double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      for (int i=0; i<nx; i++) {
         std::string label = "Instance " + std::to_string(i);
         cp[i].PrintOutput("RTM.txt", label);
      }
      then += gap;
   }
}
```

The function is similar to “write_line()” in previous examples, except it employs the “PrintOutput()” member function to trigger output events, rather than writing a specific value, such as the pH.

13.3 Simulation parameters

The simulation parameters define a domain 100 m $\times$ 1 m $\times$ 1 m, composed of 100 ChemPlugin instances of 1 m^3 each. Rather than setting porosity explicitly, we will query the first ChemPlugin instance in the domain for the value, once it is configured and initialized.

```
   // Simulation parameters.
   int nx = 100; // number of instances along x
   double length = 100; // m
   double deltax = length / nx; // m
   double deltay = 1.0, deltaz = 1.0; // m

   double time_end = 10.0;  // years
   double delta_years = time_end / 5; // years
   double next_output = 0.0;  // years
```

The simulation is set to span 10 years, writing output at 0 years, and then every 2 years.

13.4 Create instances

To create the ChemPlugin instances that will comprise the model, we, as in previous models, instantiate an instance “cp_inlet” to represent the inlet fluid, and “nx” instances to make up the interior of the domain.

```
// Create the inlet and interior instances.
ChemPlugin cp_inlet;
ChemPlugin *cp = new ChemPlugin[nx];
cp_inlet.Console("stdout");
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "volume = " + std::to_string(deltax * deltay * deltaz) +
                  " m3; time end = " + std::to_string(time_end) + " years";
for (int i=0; i<nx; i++)
   cp[i].Config(cmd);
```

For each of the interior instances, we pass a set of configuration commands that set the instance's bulk volume and the end time of the simulation.

13.5 Configure instances

Next, we configure the chemical state of each ChemPlugin instance. The strategy is to call "open_input()", which queries the user for the name of an input file and opens an input stream from that file.

```
// Query user for input file and configure inlet, initial domain.
std::ifstream input;
open_input(input, argc, argv);
int scope = 0;
while (!input.eof()) {
   std::string line;
   std::getline(input, line);
   if (line == "go")
      break;
   else if (line == "scope inlet")
      scope = 1;
   else if (line == "scope initial")
      scope = 2;
   else if (scope == 1)
      cp_inlet.Config(line);
   else if (scope == 2)
      for (int i=0; i<nx; i++)
         cp[i].Config(line);
}
```

The client scans through the input file, sending lines following an occurrence of "scope inlet" to configure "cp_inlet", and lines after "scope initial" to each of the interior instances.

13.6 Initialize instances

Once the ChemPlugin instances are configured, the client initializes them with member function “Initialize()”, and writes the boundary and initial conditions to output file “RTM.txt”.

```
if (cp_inlet.Initialize()) {
    std::cout << "Inlet failed to initialize" << std::endl;
    return exit_client(-1);
}
cp_inlet.PrintOutput("RTM.txt", "Inlet fluid");

for (int i=0; i<nx; i++) {
    if (cp[i].Initialize()) {
        std::cout << "Instance " << i << " failed to initialize" << std::endl;
        return exit_client(-1);
    }
}
write_results(cp, nx, delta_years, next_output);
```

13.7 Set transport parameters

To describe mass transport and heat transfer among the instances, the client calculates the flow rate Q, in m^3 s^{-1}; the mass transmissivity τ, in the same units; and the thermal transmissivity τ_T, in W/K.

```
// Query user for velocity; calculate flow rate and transmissivities.
double veloc_in; // m/yr
std::cout << "Please enter fluid velocity in m/yr: ";
std::cin >> veloc_in;
std::cin.ignore();
std::cout << std::endl;

double velocity = veloc_in / 31557600.;  // m/s
double porosity = cp[0].Report1("porosity");  // volume fraction
double flow = deltay * deltaz * porosity * velocity;  // m3/s

double diffcoef = 1e-10;  // m2/s
double dispersivity = 1.0; // m
double dispcoef = velocity * dispersivity + diffcoef; // m2/s
double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

double tcond = 2.0;  // W/m/K
double ttrans = deltay * deltaz * tcond / deltax;  // W/K
```

The first two values depend on the flow velocity v_x, which the client prompts the user to provide, and the porosity n, determined by querying the first instance in the domain.

13.8 Link the instances

The code for linking the instances into a flow domain

```
// Link the instances.
CpiLink link = cp[0].Link(cp_inlet);
link.Transmissivity(trans, "m3/s");
link.HeatTrans(ttrans, "W/K");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
   link = cp[i].Link(cp[i-1]);
   link.Transmissivity(trans, "m3/s");
   link.HeatTrans(ttrans, "W/K");
   link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");
```

parallels the coding in "Advection1.cpp" and "HeatTransfer1.cpp". In this client, however, we specify transmissivities for both mass transport and heat transfer.

13.9 Time marching loop

The time marching loop

```
// Time marching loop.
while (true) {
   double deltat = 1e99;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return exit_client(-1);
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceHeatTransport()) return exit_client(-1);
   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceChemical()) return exit_client(-1);

   write_results(cp, nx, delta_years, next_output);
}
```

sets out the sequential computation of mass transport, heat transfer, and chemical reaction.

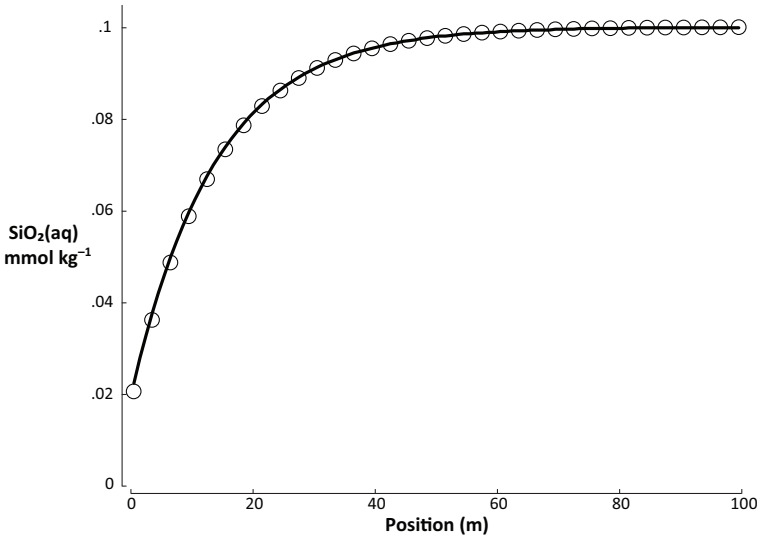
13.10 Running the model

As an example of running the reactive transport model we've constructed, we read in a file "Infilter.cpi":

```
scope inlet
   SiO2(aq) = 1 mg/kg
scope initial
   swap Quartz for SiO2(aq)
   Quartz = 70 vol%
   kinetic Quartz rate_con = 4.2e-18 surface = 1000
```

The file describes the reaction of dilute water infiltering into a quartz aquifer, where the reaction proceeds according to a kinetic rate law. We then specify a flow velocity of 100 m yr^{-1}.

The plot below shows the concentration of dissolved silica as a function of position along the aquifer, at the end of the simulation.



The circles represent the result of modeling the same scenario with program **X1t**.

13.11 C++ source code

The full C++ code for the client program is available on the ChemPlugin.GWB.com website as file "RTM1.cpp", and is listed below.

Note: This code is also available in FORTRAN and Python from ChemPlugin.GWB.com.

```
#include <iostream>
#include <fstream>
```

```
#include <string>
#include "ChemPlugin.h"

int exit_client(int status)
{
   std::cin.get();
   return status;
}

void open_input(std::ifstream &input, int argc, char** argv) {
   while (!input.is_open()) {
      std::string filename;
      if (argc < 2) {
         std::cout << "Enter RTM input script: ";
         std::cin >> filename;
         std::cin.ignore();
      }
      else {
         filename = argv[1];
      }

      input.open(filename);

      if (!input.is_open())
         std::cerr << "The input file does not exist" << std::endl;
   }
}

void write_results(ChemPlugin *cp, int nx, double gap, double& then)
{
   double now = cp[0].Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      for (int i=0; i<nx; i++) {
         std::string label = "Instance " + std::to_string(i);
         cp[i].PrintOutput("RTM.txt", label);
      }
      then += gap;
   }
}

int main(int argc, char** argv) {
   std::cout << "Model reactive transport in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   int nx = 100; // number of instances along x
   double length = 100; // m
   double deltax = length / nx; // m
```

```
double deltay = 1.0, deltaz = 1.0; // m

double time_end = 10.0;  // years
double delta_years = time_end / 5; // years
double next_output = 0.0;  // years

// Create the inlet and interior instances.
ChemPlugin cp_inlet;
ChemPlugin *cp = new ChemPlugin[nx];
cp_inlet.Console("stdout");
cp[0].Console("stdout");
cp[0].Config("pluses = banner");

std::string cmd = "volume = " + std::to_string(deltax * deltay * deltaz) +
                  " m3; time end = " + std::to_string(time_end) + " years";
for (int i=0; i<nx; i++)
   cp[i].Config(cmd);

// Query user for input file and configure inlet, initial domain.
std::ifstream input;
open_input(input, argc, argv);
int scope = 0;
while (!input.eof()) {
   std::string line;
   std::getline(input, line);
   if (line == "go")
      break;
   else if (line == "scope inlet")
      scope = 1;
   else if (line == "scope initial")
      scope = 2;
   else if (scope == 1)
      cp_inlet.Config(line);
   else if (scope == 2)
      for (int i=0; i<nx; i++)
         cp[i].Config(line);
}

// Initialize the inlet and interior instances; write out initial conditions.
if (cp_inlet.Initialize()) {
   std::cout << "Inlet failed to initialize" << std::endl;
   return exit_client(-1);
}
cp_inlet.PrintOutput("RTM.txt", "Inlet fluid");

for (int i=0; i<nx; i++) {
   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
```

```
            return exit_client(-1);
        }
    }
    write_results(cp, nx, delta_years, next_output);

    // Query user for velocity; calculate flow rate and transmissivities.
    double veloc_in; // m/yr
    std::cout << "Please enter fluid velocity in m/yr: ";
    std::cin >> veloc_in;
    std::cin.ignore();
    std::cout << std::endl;

    double velocity = veloc_in / 31557600.;  // m/s
    double porosity = cp[0].Report1("porosity");  // volume fraction
    double flow = deltay * deltaz * porosity * velocity;  // m3/s

    double diffcoef = 1e-10;  // m2/s
    double dispersivity = 1.0; // m
    double dispcoef = velocity * dispersivity + diffcoef; // m2/s
    double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

    double tcond = 2.0;  // W/m/K
    double ttrans = deltay * deltaz * tcond / deltax;  // W/K

    // Link the instances.
    CpiLink link = cp[0].Link(cp_inlet);
    link.Transmissivity(trans, "m3/s");
    link.HeatTrans(ttrans, "W/K");
    link.FlowRate(flow, "m3/s");

    for (int i=1; i<nx; i++) {
        link = cp[i].Link(cp[i-1]);
        link.Transmissivity(trans, "m3/s");
        link.HeatTrans(ttrans, "W/K");
        link.FlowRate(flow, "m3/s");
    }

    link = cp[nx-1].Link();
    link.FlowRate(-flow, "m3/s");

    // Time marching loop.
    while (true) {
        double deltat = 1e99;
        for (int i=0; i<nx; i++)
            deltat = std::min(deltat, cp[i].ReportTimeStep());
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTimeStep(deltat)) return exit_client(0);
        for (int i=0; i<nx; i++)
```

```
            if (cp[i].AdvanceTransport()) return exit_client(-1);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceHeatTransport()) return exit_client(-1);
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) return exit_client(-1);

        write_results(cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Multithreading

We consider in this chapter how to multithread a client program, so the ChemPlugin objects it spawns work in parallel. A multithreaded client, instead of running on a single computing core, executes across all the cores present in a multicore processor, such as those implemented in modern laptops, personal computers, and workstations.

Making use of a number of cores at once, a multithreaded client can run considerably more quickly than a monothreaded version, without taking up appreciably more memory. You can multithread ChemPlugin clients in C++ and Fortran.

In this chapter, we use the OpenMP application programming interface (API) to multithread the "RTM1.cpp" client program that we developed in the previous chapter, **Reactive Transport Model**. We will save the multithreaded version of the code we develop under the name "RTM2.cpp".

In contrast to parallelizing a client by multithreading, you can run it in parallel on a computing cluster. In this case, a number of copies of the client run at the same time across a group of tightly networked computers. The copies work together by sharing work among themselves, as described in the next chapter, **Cluster Computing**.

You can further blend the two methods by running a multithreaded client across the computers in a cluster; this hybrid technique is described in the following chapter, **Hybrid Parallelization**.

14.1 Code changes

Multithreading the "RTM1.cpp" application requires relatively minor changes to the original source code, as described below. Before reviewing these changes, you may wish to visit a tutorial to introduce yourself to the fundamental concepts of OpenMP programming.

14.1.1 Header files

We begin by appending

```
#include <omp.h>
```

to the system header lines at the top of the client program. File "omp.h" is the C++ header for OpenMP.

14.1.2 Number of instances

In "RTM1.cpp", the statement

```
nx = 100;
```

sets the client to instantiate 100 ChemPlugin instances. This is, however, too few to take good advantage of multithreading the client.

In general, creating a parallel region in a client program requires the system to expend overhead. If there's too little work to share, the client may spend as much time administering the the parallel region as it gains from the work sharing. As such, we'll recast the statement above as

```
nx = 4000;
```

so as to create a larger number of ChemPlugin instances.

14.1.3 Instantiation

Instantiating a ChemPlugin instance involves a non-trivial amount of work. Each instance, upon being created, lays out memory for itself, reads in a thermodynamic dataset, and prepares itself to accept configuration commands. By multithreading the instantiation step, you can significantly reduce the time required for a client to start up.

The original instantiation

```
ChemPlugin *cp = new ChemPlugin[nx];
```

is strictly serial—a single thread lays out one instance after another until "nx" instances have been created.

One way to multithread the initialization is to cast "cp" as a vector of pointers to ChemPlugin instances, instead of a vector of the instances themselves. In this way, we can instantiate in parallel:

```
   std::vector<ChemPlugin*> cp(nx);
#pragma omp parallel for
   for (int i=0; i<nx; i++)
      cp[i] = new ChemPlugin();
```

Here, the statements following the "#pragma" directive split creation of the ChemPlugin instances across a work-sharing loop.

Each vector element "cp[i]" is a pointer to a ChemPlugin instance now, rather than a reference to an instance itself. Hence, we need to change constructions like

```
cp[i].Config("pH = 5");
```

throughout "RTM1.cpp" to

```
cp[i]->Config("pH = 5");
```

Note the instances no longer occupy a contiguous block of memory.

If instantiating ChemPlugin instances in continguous memory is a programming objective, we might instead use a concurrent vector class to store the instances. The “concurrent_vector” object in “tbb.h”, a set of threaded building blocks for Intel programming environments, serves this purpose:

```
#include <tbb\tbb.h>
#undef min

   ... some code ...

   tbb::concurrent_vector<ChemPlugin> cp;
#pragma omp parallel for
   for (int i=0; i<nx; i++)
      cp.push_back( ChemPlugin() );
```

Here “ChemPlugin()” is a direct call to the object’s constructor function. As we see, “concurrent_vector” behaves like the familiar C++ “vector” object, except that it is thread-safe. #undef’ing “min” is a precaution to prevent a macro definition in “tbb.h” from interfering with our use of the standard “std::min” function.

Since there is no standard implementation of a concurrent vector class across operating systems, we work in this chapter with the first alternative, the vector of pointers to the ChemPlugin instances.

14.1.4 Configuration

Where “RTM1.cpp” configures the ChemPlugin instances with calls to the “Config()” member function, two loops over the instances can be parallelized

```
#pragma omp parallel for
   for (int i=0; i<nx; i++)
      cp[i]->Config(cmd);

   ... some code ...

      else if (scope == 2) {
#pragma omp parallel for
         for (int i=0; i<nx; i++)
            cp[i]->Config(line);
      }
```

by inserting “#pragma” directives, as shown above.

14.1.5 Initialization

The loop where client "RTM1.cpp" initializes the ChemPlugin instances

```
   for (int i=0; i<nx; i++) {
      if (cp[i].Initialize()) {
         std::cout << "Instance " << i << " failed to initialize" << std::endl;
         return exit_client(-1);
      }
   }
```

requires a little extra care to multithread, because work-sharing loops in OpenMP cannot be broken within the parallel region.

To parallelize the loop, we use a reduction variable "nerror" to count the number of times the loop encounters a condition that would cause it to break:

```
   int nerror = 0;
#pragma omp parallel for reduction(+ : nerror)
   for (int i=0; i<nx; i++) {
      if (cp[i]->Initialize()) {
         std::cout << "Instance " << i << " failed to initialize" << std::endl;
         nerror++;
      }
   }
   if (nerror) return exit_client(-1);
```

If "nerror" is non-zero after the loop completes, the client exits.

14.1.6 Linking

The loop over which client "RTM1.cpp" links the ChemPlugin instances

```
   CpiLink link ...

   ... some code ...

   for (int i=1; i<nx; i++) {
      link = cp[i].Link(cp[i-1]);
      link.Transmissivity(trans, "m3/s");
      link.HeatTrans(ttrans, "W/K");
      link.FlowRate(flow, "m3/s");
   }
```

cannot work correctly as listed, because within the parallel region the various threads would write to and read from "link" simultaneously.

Instead, "link" must be re-declared within the scope of the loop

```
CpiLink link ...

... some code ...

for (int i=1; i<nx; i++) {
    CpiLink link = cp[i]->Link(cp[i-1]);
    link.Transmissivity(trans, "m3/s");
    link.HeatTrans(ttrans, "W/K");
    link.FlowRate(flow, "m3/s");
}
```

Inside the parallel region, now, “link” is private to each thread.

14.1.7 Loop scheduling

OpenMP offers a variety of options for divvying up passes through a loop among the available computing cores, a process known as scheduling. At its simplest, OpenMP splits the passes into contiguous chunks, sending each chunk to run on one of the cores. This strategy is known as static scheduling.

Take as an example a client considering an “nx” of 3200 ChemPlugin instances, running on a computer with eight cores. In executing the loop

```
#pragma omp parallel for schedule(static)
   for (int i=0; i<nx; i++)
      cp[i]->Config(cmd);
```

OpenMP divides the instances into chunks of 400. Core zero will be responsible the first 400, core one for the next 400, and so on. The qualifier “schedule(static)” may be omitted, because static scheduling is the *de facto* default among OpenMP installations.

A static strategy works best for member functions that involve roughly the same amount of work each time they are called. A dynamic scheduling strategy is generally advantageous when calling “AdvanceChemical()”, however, because the work required by that function varies from instance to instance, depending on how many iterations are needed to converge to a solution.

In dynamic scheduling, an alternative strategy, OpenMP assigns successive passes through the loop as computer cores become available. To call “AdvanceChemical()” in a dynamically scheduled loop

```
#pragma omp parallel for schedule(dynamic)
      for (int i=0; i<nx; i++)
         cp[i]->AdvanceChemical();
```

you use the “schedule(dynamic)” qualifier in the compiler directive.

Scheduling the loop dynamically can significantly reduce latency, the state in which some cores sit idle waiting for others to finish. In a static schedule, one of the chunks

might include a number of instances where "AdvanceChemical()" converges slowly, such as in the vicinity of a reaction front. A dynamically scheduled loop, on the other hand, is flexible in how it assigns passes to cores, so some cores may end up working on a relatively few instances that converge slowly, whereas others take care of a larger number of quickly-converging instances.

14.1.8 Time marching loop

Within the time marching loop, there are five loops over the ChemPlugin instances that can be multithreaded:

```
   // Time marching loop.
   while (true) {
      double deltat = 1e99;
#pragma omp parallel for reduction(min : deltat)
      for (int i=0; i<nx; i++)
         deltat = std::min(deltat, cp[i]->ReportTimeStep());

      int nerror = 0;
#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceTimeStep(deltat)) nerror++;
      if (nerror) return exit_client(0);

#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceTransport()) nerror++;
      if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceHeatTransport()) nerror++;
      if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror) schedule(dynamic)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceChemical()) nerror++;
      if (nerror) return exit_client(-1);

      write_results(cp, nx, delta_years, next_output);
   }
```

The first work-sharing loop requires reduction over "deltat", whereas the remaining loops reduce over "nerror". Each loop is statically scheduled, except the last, for which scheduling is dynamic.

14.2 Speedup

To test the extent to which multithreading the client program "RTM2.cpp" sped its execution relative to the single-threaded version "RTM1.cpp", we timed the solution of three problems using varying numbers "nx" of ChemPlugin instances. We compiled the clients into 64-bit apps and ran the tests on a Windows 8.1 computer with a hyperthreaded quad core Intel Core i7 processor and 12 GB of memory. In each case, we report speedup as the clock time required to solve the problem using the single-threaded relative to the multithreaded client.

The three problems are:

- **Complexation**: 7 chemical components; 1 kinetic equation describing an aqueous complexation reaction.
- **Weathering**: 8 chemical components; 3 kinetic equations describing mineral dissolution.
- **Dual porosity**: 6 chemical components; solute diffusion into and out of stagnant zones.

The speedups observed on a quad core processor for the multithreaded client on runs made using 4 000, 10 000, and 40 000 ChemPlugin instances are:

	nx = 4 000	10 000	40 000
Complexation	×3.65	×3.90	×4.08
Weathering	×4.07	×4.04	×3.92
Dual porosity	×3.22	×3.72	×4.02

The nominal maximum speedup on a quad core processor is ×4. It is possible, as we see in this table, to exceed this limit somewhat by hyperthreading—i.e., scheduling two threads on each core.

14.3 C++ source code

The full C++ code for the client program is listed below; versions in C++ and Fortran are available to download from the ChemPlugin.GWB.com website, respectively, as files "RTM2.cpp" and "RTM2.f90".

```
#include <iostream>
#include <fstream>
#include <string>
#include <omp.h>
#include <vector>
#undef min
#include "ChemPlugin.h"

int exit_client(int status)
```

```
{
   std::cin.get();
   return status;
}

void open_input(std::ifstream &input, int argc, char** argv) {
   while (!input.is_open()) {
      std::string filename;
      if (argc < 2) {
         std::cout << "Enter RTM input script: ";
         std::cin >> filename;
         std::cin.ignore();
      }
      else {
         filename = argv[1];
      }

      input.open(filename);

      if (!input.is_open())
         std::cerr << "The input file does not exist" << std::endl;
   }
}

void write_results(std::vector<ChemPlugin*>& cp, int nx, double gap, double& then)
{
   double now = cp[0]->Report1("Time", "years");
   if ((then - now) < gap / 1e4) {
      for (int i=0; i<nx; i++) {
         std::string label = "Instance " + std::to_string(i);
         cp[i]->PrintOutput("RTM.txt", label);
      }
      then += gap;
   }
}

int main(int argc, char** argv) {
   std::cout << "Model reactive transport in one dimension"
             << std::endl << std::endl;

   // Simulation parameters.
   int nx = 1000; // number of instances along x
   double length = 100; // m
   double deltax = length / nx; // m
   double deltay = 1.0, deltaz = 1.0; // m

   double time_end = 10.0;  // years
   double delta_years = time_end / 5; // years
```

```
   double next_output = 0.0;  // years

   // Create the inlet and interior instances.
   ChemPlugin cp_inlet;

   std::vector<ChemPlugin*> cp(nx);
#pragma omp parallel for
   for (int i=0; i<nx; i++)
      cp[i] = new ChemPlugin();

   cp_inlet.Console("stdout");
   cp[0]->Console("stdout");
   cp[0]->Config("pluses = banner");

   std::string cmd = "volume = " + std::to_string(deltax * deltay * deltaz) +
                     " m3; time end = " + std::to_string(time_end) + " years";
#pragma omp parallel for
   for (int i=0; i<nx; i++)
      cp[i]->Config(cmd);

   // Query user for input file and configure inlet, initial domain.
   std::ifstream input;
   open_input(input, argc, argv);
   int scope = 0;
   while (!input.eof()) {
      std::string line;
      std::getline(input, line);
      if (line == "go")
         break;
      else if (line == "scope inlet")
         scope = 1;
      else if (line == "scope initial")
         scope = 2;
      else if (scope == 1)
         cp_inlet.Config(line);
      else if (scope == 2) {
#pragma omp parallel for
         for (int i=0; i<nx; i++)
            cp[i]->Config(line);
      }
   }

   // Initialize the inlet and interior instances; write out initial conditions.
   if (cp_inlet.Initialize()) {
      std::cout << "Inlet failed to initialize" << std::endl;
      return exit_client(-1);
   }
   cp_inlet.PrintOutput("RTM.txt", "Inlet fluid");
```

```
   int nerror = 0;
#pragma omp parallel for reduction(+ : nerror)
   for (int i=0; i<nx; i++) {
      if (cp[i]->Initialize()) {
         std::cout << "Instance " << i << " failed to initialize" << std::endl;
         nerror++;
      }
   }
   if (nerror) return exit_client(-1);

   write_results(cp, nx, delta_years, next_output);

   // Query user for velocity; calculate flow rate and transmissivities.
   double veloc_in; // m/yr
   std::cout << "Please enter fluid velocity in m/yr: ";
   std::cin >> veloc_in;
   std::cin.ignore();
   std::cout << std::endl;

   double velocity = veloc_in / 31557600.;  // m/s
   double porosity = cp[0]->Report1("porosity");  // volume fraction
   double flow = deltay * deltaz * porosity * velocity; //  m3/s

   double diffcoef = 1e-10;  // m2/s
   double dispersivity = 1.0; // m
   double dispcoef = velocity * dispersivity + diffcoef; // m2/s
   double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

   double tcond = 2.0;  // W/m/K
   double ttrans = deltay * deltaz * tcond / deltax;  // W/K

   // Link the instances.
   CpiLink link = cp[0]->Link(cp_inlet);
   link.Transmissivity(trans, "m3/s");
   link.HeatTrans(ttrans, "W/K");
   link.FlowRate(flow, "m3/s");

#pragma omp parallel for
   for (int i=1; i<nx; i++) {
      CpiLink link = cp[i]->Link(cp[i-1]);
      link.Transmissivity(trans, "m3/s");
      link.HeatTrans(ttrans, "W/K");
      link.FlowRate(flow, "m3/s");
   }

   link = cp[nx-1]->Link();
   link.FlowRate(-flow, "m3/s");
```

```
   // Time marching loop.
   while (true) {
      double deltat = 1e99;
#pragma omp parallel for reduction(min : deltat)
      for (int i=0; i<nx; i++)
         deltat = std::min(deltat, cp[i]->ReportTimeStep());

      int nerror = 0;
#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceTimeStep(deltat)) nerror++;
      if (nerror) return exit_client(0);

#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceTransport()) nerror++;
      if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceHeatTransport()) nerror++;
      if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror) schedule(dynamic)
      for (int i=0; i<nx; i++)
         if (cp[i]->AdvanceChemical()) nerror++;
      if (nerror) return exit_client(-1);

      write_results(cp, nx, delta_years, next_output);
   }

   // Never gets here.
   return 0;
}
```

Cluster Computing

To parallelize a client program on a computing cluster, a group of tightly networked computers, you run duplicate copies of the client distributed across the cluster. Each client copy is responsible for the calculations at only a portion of the ChemPlugin instances considered in a run. Since the copies share the workload, computing clusters offer the potential for faster turnaround, larger simulations, or both.

You might, for example, run a simulation making use of 32,000 ChemPlugin instances on a cluster of 32 computers. In this case, you could launch 32 copies of the client program, each running on its own computer, and each responsible for 1,000 ChemPlugin instances. Or, you could spawn 128 copies, four per computer, each working on 250 instances. You can even work on a "cluster" of a single computer, which can be helpful as you debug a client.

A cluster solution, as you can see, differs from a multithreaded program in that rather than a single copy of the client program running across the cores on one computer, multiple program copies run across a group of computers. It is possible to implement a hybrid solution in which each client copy on a computing cluster is multithreaded, thereby taking advantage of both parallelization strategies at once. Such hybrid solutions are considered in the following chapter, **Hybrid Parallelization**.

15.1 MPI protocol

Computing clusters typically use a protocol called MPI, which stands for "Message Passing Interface," to manage copies of the client program and allow them to pass information among themselves. Native MPI support is available in C++ and Fortran, and we recommend you use one of those languages to code ChemPlugin client programs under MPI.

When you launch a program, MPI sets up duplicate program copies in a global communication group called "MPI_COMM_WORLD". Each copy in a group of N has a worker number or rank ranging from zero to $N - 1$. The first copy, with rank zero, is sometimes referred to as the "master worker."

The client copies as they run commonly need to share data among themselves, so that one copy has access to information known to the others. In the case of a ChemPlugin application, when a copy is creating a link, or considering transport across one, the client needs to know conditions within the ChemPlugin instance at the

opposite end. The opposing instance might be known to the client copy in question, another copy on the same computer, or a copy of the program running elsewhere in the cluster.

The onus in cluster computing is on the programmer to arrange data sharing when necessary, and preferably only when necessary, since the clients must pause their work to transmit and gather information, and much of the data may need to pass among computers over the network, a relatively slow process. Fortunately, the ChemPlugin library wraps the gathering and scattering of data into a single function call, greatly reducing the burden of cluster computing on the programmer.

15.2 ChemPlugin under MPI

To run on a computing cluster, a ChemPlugin client program needs to pull in from file "chemplugin_mpi.dll" (or "chemplugin_mpi.so" under Linux) a special MPI version of the software object. The client creates a family of these special ChemPlugin instances and controls them, as before, with member function calls.

In this section, we explore a few special considerations needed to launch a client program in MPI, spawn ChemPlugin instances, and control them across a distributed environment.

15.2.1 Initializing MPI

A client program needs to take a few extra steps as it starts up in an MPI environment. The user runs a cluster simulation by telling the system to launch a group of program copies. A copy, however, is born knowing only about itself; to work as part of the group, it needs to determine the group's size and its position or rank within the group.

To this end, a client begins by initializing the MPI library and querying it to learn group size and the copy's rank

```
// Initialize MPI
MPI_Init(NULL, NULL);
// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
// Get the rank of this process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

The size and rank are held in variables "world_size" and "world_rank".

15.2.2 Instantiation

You instantiate a ChemPlugin instance under MPI in two steps. First, use ChemPlugin's constructor

```
ChemPlugin cp;
```

to create a stub for a ChemPlugin object. The stub contains empty member functions and will hold a few critical details about the corresponding ChemPlugin object, once it has been created. Significantly, each client copy instantiates stubs for the entire family of ChemPlugin objects to be considered by the communication group, not just the instances for which it will be responsible.

You then use the "MpiAssign()" member function

```
cp.MpiAssign(i);
```

to associate the stub with a specific copy of the client program. In this case, instance "cp" is assigned to the client copy of rank "i". That client copy expands the stub for "cp" into a local ChemPlugin instance, whereas the other copies simply take note of the stub's assignment as a foreign instance.

A given ChemPlugin instance exists on only one client copy, then; the remaining copies contain only a stub associated with the instance. The first client program in the group is responsible for doing work associated with each ChemPlugin instance assigned a rank of zero, the second program copy handles instances with a rank of one, and so on.

You can alternatively create a ChemPlugin stub and assign it to a program copy in one step. ChemPlugin's constructor normally allows two arguments, as described in the **Overview** chapter, one for setting the output stream and a second that lets you specify various program options. For the cluster version of ChemPlugin, you can set as a third argument the rank to be assigned to the instance being created.

By passing rank to the constructor, you conflate the construction and assignment steps onto a single line of code. For example, the line

```
ChemPlugin cp(NULL, NULL, i);
```

creates a stub for an instance "cp"; assigns it a rank of "i"; and, on client copy "i" alone, expands it into a local ChemPlugin object. We did not make use of the first two arguments in this case, so "NULL" serves as a placeholder.

15.2.3 Assigning rank

It is important that each client copy assign a rank to every ChemPlugin instance to be considered by the group, whether the instance is local or foreign. The code fragment, for example

```
ChemPlugin cp;
// Do not do this!
if (world_rank == i) cp.MpiAssign(i);
```

is incorrect, because "cp" is assigned a rank by only client copy "i", rather than by each copy in the communication group. Instead, the code

```
ChemPlugin cp;
cp.MpiAssign(i);
```

works correctly.

For a vector of instances, similarly, every client copy constructs a stub for each of the group's instances and then associates the stub with a single client copy. Consider the code

```
std::vector<ChemPlugin> cp(nx);
for (int i=0; i<nx; i++)
   cp[i].MpiAssign(i * world_size / nx);
```

noting especially the "MpiAssign()" call. Here, the integer expression "i * world_size / nx" serves to divide the instances evenly among the copies. If the number of copies "world_size" is 32 and the number of instances "nx" is 32,000, then the first 1000 instances are assigned to the first client copy, the next 1000 to the second copy, and so on.

You can significantly speed execution of a client programs by minimizing the number of links between ChemPlugin instances of different ranks. When a link connects two instances of the same rank, the program does not need to obtain any data from other client copies to evaluate transport, since the instances are assigned to the same copy. When the instances differ in rank, in contrast, accounting for transport across the link inevitably involves data transfer, either between copies on the same computer, or more commonly across the network between copies running elsewhere in the cluster, a comparatively slow process.

As an example, consider a case in which the instances "cp" are to be linked in a linear chain. In the code fragment

```
// Probably a bad idea!
std::vector<ChemPlugin> cp(nx);
for (int i=0; i<nx; i++)
   cp[i].MpiAssign(i % world_size);
```

where "%" is the modulus operator, the expression "i % world_size" assigns rank in rotation. Linking the instances in a line, each link now connects instances of differing rank, and hence assigned to different client copies. The transfer overhead in this case will be sharply higher than in the first case, where the expression "i * world_size / nx" assigned rank in chunks.

15.2.4 Calling member functions

A client can call the member functions of any ChemPlugin instance under MPI, not just the ones assigned to it. If the ChemPlugin instance is local, that is to say of the same rank as the calling program, the function call proceeds as usual. Whenever the

instance is foreign and hence assigned to a different client copy, on the other hand, the function call returns without doing work.

A client, then, doesn't need to keep track of which ChemPlugin instances belong to it, which simplifies coding. As an example, consider the loop

```
std::vector<ChemPlugin> cp(nx);
... some code ...
for (int i=0; i<nx; i++)
   cp[i].Initialize();
```

We might be tempted to recode the loop as

```
std::vector<ChemPlugin> cp(nx);
... some code ...
for (int i=0; i<nx; i++)
   // This step is not necessary!
   if (cp[i].MpiOnRank())
      cp[i].Initialize();
```

where the "MpiOnRank()" member function returns non-zero if "cp[i]" is local.

The "if" statement here, while certainly acceptable, is unnecessary, since calls for "Initialize()" for foreign instances "cp[i]" return without doing work. Nesting "if" checks when calling member functions adds to a code's complexity without providing significant benefit.

15.2.5 Transferring data

When a ChemPlugin instance forms a link with another instance, and again when it transfers mass or heat across the link, it needs to know current information about the opposing instance. For a conventional client program, retrieving this data is a simple matter, because both instances share the same memory space.

In cluster computing, the task is more complex, because the opposing instance may be found on a different client copy, perhaps on a different computer. ChemPlugin's strategy is to cache data from the opposing instance at each link to a foreign instance, within the CpiLink object inself. In this way, current information is at hand when needed, as long as the programmer periodically updates the cache.

You use the "MpiUpdateLink()" member function from the CpiLink object to update a link's cache. In the code fragment

```
for (int i=0; i<nx; i++)
   for (int l=0; l<cp[i].nLinks(); l++)
      cp[i].Link(l).MpiUpdateLink();
```

the client scans across the links on every ChemPlugin instance.

On a pass in which instance "i" is assigned locally, but the opposing instance across link "l" is not, the client needs to cache information about the opposing instance. In

this case, the call to "MpiUpdateLink()" waits to receive this information from another client copy.

In the converse case, when instance i is foreign, but the opposing instance is local, the client realizes that another copy must be, or soon will be waiting for information. In this case, the "MpiUpdateLink()" call transmits the state of the opposing instance. Once the data is received, the copies are released to continue execution.

Passing data in this way is known in cluster computing as synchronous communication, which means each client copy must process attempts at data transfer in the same order. If the copies were to update links in different permutations, the group might easily fall into a gridlock in which one client copy pauses to wait for a transmission from another copy, which itself is stopped waiting for data from the first copy.

15.2.6 Retrieving results

The "Report()" family of functions, described in **Retrieving Results**, works like other member functions, depending on whether it is called for a local or a foreign instance. For a local instance, the functions behave normally, but called for a foreign instance, they return a placeholder value, without doing any work. The placeholder value is zero for "Report()", "ANULL" for "Report1()", "ANULL" cast as an integer for "Report1i()", and a NULL pointer for "Report1c()".

Consider as an example a client program that needs to know partial pressure in MPa of the various gases considered in the simulation in order to carry out a calculation. In the code fragment

```
double *Pgas = new double[ngas];
for (int i=0; i<nx; i++) {
   int len = cp[i].Report(Pgas, "gas_pressure", "MPa");
   if (len) {
      ... some code ...
   }
}
```

variable "len" is non-zero when "i" points to a local instance, and zero otherwise. The code, then, uses the gas pressures at local instances for its purposes, leaving work for foreign instances to the other client copies.

For a client needing the partial pressure of only CO_2 gas, the code might instead be posed

```
for (int i=0; i<nx; i++) {
   double pCO2 = cp[i].Report1("gas_pressure CO2(g)", "MPa");
   if (pCO2 != ANULL) {
      ... some code ...
   }
}
```

Again, the loop would carry out the calculations for local, but not foreign ChemPlugin instances.

In cases in which the client needs to retrieve information from any ChemPlugin instance, whether local or foreign, you use the "MpiReport()" family of functions, which includes "MpiReport()", "MpiReport1()", "MpiReport1i()", and "MpiReport1c()". The functions work like their counterparts "Report()" and so on, except they can retrieve results from instances assigned to any client copy.

Suppose, for example, each client copy needs a list of pH values at a number of ChemPlugin instances, even though some of the instances may be associated with other copies. The code fragment

```
double *pH_list = new double[nx];
for (int i=0; i<nx; i++)
   pH_list[i] = cp[i].MpiReport1("pH");
```

creates a vector "pH_list" of pH values at instances zero through "nx", regardless of where the instances are assigned.

The "MpiReport()" family of functions, like the "MpiUpdate()" function, is synchronous, meaning each client copy needs to call the function in the same sequence. The need for parallel ordering arises because at each pass through the loop, one client copy broadcasts information while the others wait to receive it, and they must pause until the transmission is complete.

As an example, in the code

```
std::vector<double> pH_list(nx);
// Do not do this!
if (world_rank == 0)
   for (int i=0; i<nx; i++)
      pH_list[i] = cp[i].MpiReport1("pH");
```

only the first client copy in the group, the one of rank zero, calls "MpiReport1()". The copy in this case would be left waiting indefinitely the first time it tries to receive the pH from a foreign instance, since no copy will be sending that information.

There is a way, however, to use the "MpiReport()" family of functions to allow a single client copy to collect information from the other copies. To do so, you append the rank of the collector copy to the function arguments. The function in this case executes normally on the collector copy, as well as the client copy associated with the ChemPlugin instance in question. On the remaining copies, the function call returns a placeholder value, without doing work.

Consider a case in which the first client copy needs to assemble inorganic lead concentrations in μmol kg^{-1} from a group of instances, some of which are assigned to other copies. The fragment

```
int collector = 0;
double *conc = NULL;
```

```
if (world_rank == collector) conc = new double[nx];

for (int i=0; i<nx; i++) {
   double ci = cp[i].MpiReport1("concentration Pb++", "umol/kg", collector);
   if (world_rank == collector) conc[i] = ci;
}
```

accomplishes this task, as long as the loop runs on copy "collector" as well as the client copies responsible for the instances "cp[i]". If we had coded the loop

```
for (int i=0; i<nx; i++)
   // Do not do this, either!
   if (world_rank == collector)
      conc[i] = cp[i].MpiReport1("concentration Pb++", "umol/kg", collector);
```

execution would fail, because "MpiReport1()" is called only by the collector copy.

As a second example, the collector copy needs the concentration of all the species considered by an instance "cp", which might have been assigned to any client copy. We could code this case as

```
double *conc = NULL;
int nsp = cp.MpiReport(NULL, "concentration aqueous", "", collector);
if (nsp > 0) conc = new double[nsp];
cp.MpiReport(conc, "concentation aqueous", "umol/kg", collector);
```

Running the code on all the client copies, the first call to "MpiReport()" would return a non-zero "nsp" on only "collector" and the copy to which "cp" is assigned. On the remaining copies, there is no need to allocate memory for "conc", because the second "MpiReport()" call returns without doing work.

As an alternative, we could recode the fragment above as

```
double *conc = NULL;
int nsp = 0;
if (world_rank == collector || cp.OnRank()) {
   int nsp = cp.MpiReport(NULL, "concentration aqueous", "", collector);
   conc = new double[nsp];
   cp.MpiReport(conc, "concentation aqueous", "umol/kg", collector);
}
```

with equal validity. In either case, we see that only the collector copy and the copy responsible for "cp" are involved in the data transfer.

If you do not specify a unit when calling a function in the "Report()" family, you need to include a placeholder argument to set a collector. Since the pH has no unit, for example, the code

```
for (int i=0; i<nx; i++) {
   double pH = cp[i].MpiReport1("pH", NULL, collector);
   ... some code ...
}
```

needs to set the second parameter to “NULL” or an empty string ‘ “” ’. Finally, specifying a trailing parameter of “GLOBAL”

```
double pH = cp.MpiReport1("alkalinity", "meq_acid/kg", GLOBAL);
```

signifies the entire group is to serve as collector, and hence reverts to the behavior in the absence of the parameter, as described previously.

15.3 Code changes

To see how all this works, we adapt in this chapter the “RTM1.cpp” client program from the **Reactive Transport Model** chapter and its multithreaded counterpart “RTM2.cpp” from the **Multithreading** chapter to make an MPI program “RTM3.cpp” that runs in parallel on computing clusters.

As we build our MPI client, keep in mind that a number of copies of the client program will run together on the cluster, and that each ChemPlugin instance spawned will be associated with only one of those copies. Recall as well that a client can call the member functions of any ChemPlugin instance, whether assigned to itself or another program copy: if the instance is assigned elsewhere, the function call simply returns without doing work.

The discussion below lays out the changes required to turn “RTM1.cpp” and “RTM2.cpp” into the cluster computing application “RTM3.cpp”. A full listing of the resulting code appears at the end of this chapter. You may find it useful to consult a text or tutorial on MPI programming, either before beginning or as you encounter new concepts.

15.3.1 Header files

In the header section of “RTM2.cpp’, we replace the OpenMP header with the line

```
#include <mpi.h>
```

which pulls in the header for the MPI library’s SDK, or software development kit. An MPI SDK appropriate for your cluster, of course, needs to be installed in your environment.

15.3.2 Ancillary functions

Three functions at the top of the “RTM2.cpp” code provide for terminating program execution, opening an input file, and writing simulation results. Adapting the functions for cluster computing demonstrates useful principles for developing MPI code.

Function "exit_client()" terminates the program normally when called with parameter "status" set to zero, at the end of time marching, and abnormally when "status" is non-zero, if the client has encountered an error condition. The function's MPI version

```
int exit_client(int status)
{
   if (status != 0)
      MPI_Abort(MPI_COMM_WORLD, status);
   MPI_Finalize();
   return status;
}
```

calls "MPI_finalize()" in the former case to gracefully wrap up execution of the client copy. In the latter case, however, the client has encountered an unrecoverable error, in which case the function calls "MPI_Abort()" to force each copy in the group to terminate, not just the copy in question.

The "open_input()" function differs from "RTM1.cpp" in that, while not strictly necessary, MPI clients generally run as batch jobs, rather than interactively. As such, the function

```
void open_input(std::ifstream& input, int argc, char** argv) {
   std::string filename;
   if (argc < 2) {
      std::cerr << "You must specify an input file as the first program argument."
                << std::endl;
      exit_client(-1);
   }
   else {
      filename = argv[1];
   }

   input.open(filename);

   if (!input.is_open()) {
      std::cerr << "The input file does not exist" << std::endl;
      exit_client(-1);
   }
}
```

looks for the name of the input file on the command line used to launch the program, instead of querying the user for it. In C++, "argc" is the number of arguments on the command line, including the command itself, and "argv[1]" points to the first argument.

The MPI version of function "write_results()"

```
void write_results(std::vector<ChemPlugin>& cp, int nx, double gap, double& then)
{
   double now = cp[0].MpiReport1("Time", "years");
```

```
    int step = nx / 100;
    if ((then - now) < gap / 1e4) {
        int last_rank = cp[0].MpiRank();
        for (int i=0; i<nx; i+=step) {
            if (cp[i].MpiRank() != last_rank) {
                last_rank = cp[i].MpiRank();
                MPI_Barrier(MPI_COMM_WORLD);
            }
            std::string label = "Instance " + std::to_string(i);
            cp[i].PrintOutput("RTM.txt", label);
        }
        then += gap;
    }
}
```

works by scanning across the ChemPlugin instances making up the domain, as it does in “RTM2.cpp”. On a given pass through the loop, only the client copy identified by “cp[i].MpiRank()” is doing work when it calls “PrintOutput()”; calls by the other copies return immediately.

As it scans, the function takes note of when rank changes from one instance to the next. A change in rank triggers a call to “MPI_Barrier()”, which forces all client copies in “MPI_COMM_WORLD” to pause execution until each reaches that point in the loop. Pausing in this way gives the copy doing work a chance to catch up with the other copies. Absent the call to “MPI_Barrier()”, the client copies would write into the output file at the same time, leaving a jumbled mess.

15.3.3 Client startup

At the head of the client program itself is code to initialize the MPI library and determine group size and the copy’s rank

```
    // Initialize MPI
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

as previously described. The first client copy, the one of rank zero, proceeds to write a header message

```
    if (world_rank == 0)
        std::cout << std::endl
```

```
                << "Model reactive transport in one dimension using MPI"
                << std::endl << std::endl;
```

If we had not limited this step to a single client, the message would be directed into the standard output stream "world_size" times, probably in an interspersed mishmash.

15.3.4 Instantiation

You instantiate ChemPlugin instances, as discussed previously in this chapter, in two steps under MPI. Each client copy uses the ChemPlugin constructor

```
ChemPlugin cp_inlet;
std::vector<ChemPlugin> cp(nx);
```

to create a stub for all of the ChemPlugin instances to be considered by the communication group. You then associate each stub with a single client copy

```
cp_inlet.MpiAssign(0);
for (int i=0; i<nx; i++)
   cp[i].MpiAssign(i * world_size / nx);
```

by assigning it a rank. When the assigned rank matches that of the client copy, the stub expands into a full ChemPlugin instance.

Here, instance "cp_inlet" is associated with the first client copy, or rank zero, and responsibility for instances "cp" is distributed in chunks among the client copies, using the method discussed previously.

15.3.5 Work sharing loops

Coding ChemPlugin work sharing loops is more straightforward in MPI than in OpenMP. Specifically, compiler directives are not required and the need for special reductions is reduced.

As an example, the OpenMP code

```
   int nerror = 0;
#pragma omp parallel for reduction(+ : nerror)
   for (int i=0; i<nx; i++) {
      if (cp[i].Initialize()) {
         std::cout << "Instance " << i << " failed to initialize" << std::endl;
         nerror++;
      }
   }
   if (nerror) return exit_client(-1);
```

might appear in an MPI client as

```
for (int i=0; i<nx; i++) {
    if (cp[i].Initialize()) {
        std::cout << "Instance " << i << " failed to initialize" << std::endl;
        return exit_client(-1);
    }
}
```

The “#pragma” directive is eliminated, as is the reduction on “nerror” and the need to wait until the loop completes to exit.

15.3.6 Setting velocity

Since our client is designed as a batch, rather than interactive program, we replace code querying the user for velocity with the lines

```
double veloc_in; // m/yr
if (argc < 3) {
    std::cerr << "You must specify fluid velocity in m/yr as the "
                 "second program argument." << std::endl;
    exit_client(-1);
}
veloc_in = atof(argv[2]);
```

The code here assigns velocity as the second argument of the command that launched the program.

15.3.7 Linking

The code for linking instances

```
// Link the instances.
CpiLink link = cp[0].Link(cp_inlet);
link.Transmissivity(trans, "m3/s");
link.HeatTrans(ttrans, "W/K");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
    link = cp[i].Link(cp[i-1]);
    link.Transmissivity(trans, "m3/s");
    link.HeatTrans(ttrans, "W/K");
    link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");
```

is essentially unchanged from "RTM1.cpp", the monothreaded version of this client. Under MPI, however, the "cp[i].Link()" function call works by gathering the information it needs about the linked instance, transferring data from another client copy when necessary, as described in the next section.

15.3.8 Time marching loop

The time marching loop in "RTM3.cpp"

```
while (true) {
   double deltat = 1e99, deltat_world;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   MPI_Allreduce(&deltat, &deltat_world, 1,
                        MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat_world)) return exit_client(0);

   for (int i=0; i<nx; i++)
      for (int l=0; l<cp[i].nLinks(); l++)
         if (cp[i].Link(l).MpiUpdateLink()) return exit_client(-1);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return exit_client(-1);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceHeatTransport()) return exit_client(-1);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceChemical()) return exit_client(-1);

   write_results(cp, nx, delta_years, next_output);
}
```

differs in only a couple of places from that in the serial version "RTM1.cpp".

First, the code for determining the time step

```
   double deltat = 1e99, deltat_world;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   MPI_Allreduce(&deltat, &deltat_world, 1,
                        MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
```

begins by scanning for the smallest time step "deltat" reported by local instances, given that "cp[i].ReportTimeStep()" returns a large number for instances belonging to another client copy. It then calls the library function "MPI_Allreduce()" to find the least value determined by the entire group, which is returned in "deltat_world".

Second, after advancing the time step at the ChemPlugin instances, the code calls "MpiUpdateLink()" to sync each link among the instances

```
for (int i=0; i<nx; i++)
   for (int l=0; l<cp[i].nLinks(); l++)
      if (cp[i].Link(l).MpiUpdateLink()) return exit_client(-1);
```

The program terminates if "MpiUpdateLink()" returns an error code.

The procedure for advancing the transport and chemical equations is unchanged from the serial version.

15.4 Running the example

The procedure for compiling and executing a ChemPlugin client on a computing cluster varies from one cluster to another and may involve console commands, graphical dashboards, or both. You should consult your system administrator before attempting to run your program.

To compile a C++ program for MPI, you normally use a wrapper such as "mpicl" designed for cluster computing, in place of a call to a conventional compiler like "icl". Among other benefits, using the correct command will cause the system to pull in the MPI include files and libraries.

The command to compile program "RTM3.cpp" under Windows using "mpicl" might be

```
mpicl RTM3.cpp -I "C:\Program Files\ChemPlugin\src" \
      "C:\Program Files\ChemPlugin\ChemPlugin.lib"
```

assuming the ChemPlugin software is installed in "C:\Program Files\ChemPlugin". Under Linux, the command might be

```
mpicl++ RTM3.cpp -I "/usr/local/ChemPlugin/src" \
      "/usr/local/ChemPlugin/ChemPlugin.lib"
```

where ChemPlugin is installed under "/usr/local/ChemPlugin". The compilation and subsequent linking should produce an executable file "RTM3.exe".

Among the ways that may be available to you to launch an MPI communication group, perhaps the simplest is the "mpiexec" command. As an example, entering

```
mpiexec -n 32 RTM3 myInput.cpi 100.0
```

launches 32 copies of RTM3, using "myInput.cpi" as the input file and "100.0' as the velocity. Again, you should discuss your needs with your system administrator before attempting to run the program.

15.5 C++ source code

The full C++ code for the client program is available on the ChemPlugin.GWB.com website as file "RTM3.cpp", and is listed below.

Note: The code is also available in FORTRAN from ChemPlugin.GWB.com.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include "ChemPlugin.h"
#include <mpi.h>

int exit_client(int status)
{
   if (status != 0)
      MPI_Abort(MPI_COMM_WORLD, status);
   MPI_Finalize();
   return status;
}

void open_input(std::ifstream& input, int argc, char** argv) {
   std::string filename;
   if (argc < 2) {
      std::cerr << "You must specify an input file as the first program argument."
                << std::endl;
      exit_client(-1);
   }
   else {
      filename = argv[1];
   }

   input.open(filename);

   if (!input.is_open()) {
      std::cerr << "The input file does not exist" << std::endl;
      exit_client(-1);
   }
}

void write_results(std::vector<ChemPlugin>& cp, int nx, double gap, double& then)
{
   double now = cp[0].MpiReport1("Time", "years");
   int step = nx / 100;
   if ((then - now) < gap / 1e4) {
      int last_rank = cp[0].MpiRank();
      for (int i=0; i<nx; i+=step) {
```

```
            if (cp[i].MpiRank() != last_rank) {
                last_rank = cp[i].MpiRank();
                MPI_Barrier(MPI_COMM_WORLD);
            }
            std::string label = "Instance " + std::to_string(i);
            cp[i].PrintOutput("RTM.txt", label);
        }
        then += gap;
    }
}

int main(int argc, char** argv) {
    // Initialize MPI
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0)
        std::cout << std::endl
                  << "Model reactive transport in one dimension using MPI"
                  << std::endl << std::endl;

    // Simulation parameters.
    int nx = 4000; // number of instances along x
    double length = 100; // m
    double deltax = length / nx; // m
    double deltay = 1.0, deltaz = 1.0; // m

    double time_end = 10.0;  // years
    double delta_years = time_end / 5; // years
    double next_output = 0.0;  // years

    // Create the inlet and interior instances.
    ChemPlugin cp_inlet;
    std::vector<ChemPlugin> cp(nx);

    // Assign instances to ranks
    cp_inlet.MpiAssign(0);
    for (int i=0; i<nx; i++)
        cp[i].MpiAssign(i * world_size / nx);

    cp_inlet.Console("stdout");
    cp[0].Console("stdout");
    cp[0].Config("pluses = banner");
```

```
std::string cmd = "volume = " + std::to_string(deltax * deltay * deltaz) +
                  " m3; time end = " + std::to_string(time_end) + " years";
for (int i=0; i<nx; i++)
   cp[i].Config(cmd);

// Set input file and configure inlet, initial domain.
std::ifstream input;
open_input(input, argc, argv);
int scope = 0;
while (!input.eof()) {
   std::string line;
   std::getline(input, line);
   if (line == "go")
      break;
   else if (line == "scope inlet")
      scope = 1;
   else if (line == "scope initial")
      scope = 2;
   else if (scope == 1)
      cp_inlet.Config(line);
   else if (scope == 2)
      for (int i=0; i<nx; i++)
         cp[i].Config(line);
}

// Initialize the inlet and interior instances; write out initial conditions.
if (cp_inlet.Initialize()) {
   std::cout << "Inlet failed to initialize" << std::endl;
   return exit_client(-1);
}
cp_inlet.PrintOutput("RTM.txt", "Inlet fluid");

for (int i=0; i<nx; i++) {
   if (cp[i].Initialize()) {
      std::cout << "Instance " << i << " failed to initialize" << std::endl;
      return exit_client(-1);
   }
}
write_results(cp, nx, delta_years, next_output);

// Set velocity; calculate flow rate and transmissivities.
double veloc_in; // m/yr
if (argc < 3) {
   std::cerr << "You must specify fluid velocity in m/yr as the "
                "second program argument." << std::endl;
   exit_client(-1);
}
```

```
veloc_in = atof(argv[2]);

double velocity = veloc_in / 31557600.;  // m/s
double porosity = cp[0].MpiReport1("porosity");  // volume fraction
double flow = deltay * deltaz * porosity * velocity;  // m3/s

double diffcoef = 1e-10;  // m2/s
double dispersivity = 1.0; // m
double dispcoef = velocity * dispersivity + diffcoef; // m2/s
double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

double tcond = 2.0;  // W/m/K
double ttrans = deltay * deltaz * tcond / deltax;  // W/K

// Link the instances.
CpiLink link = cp[0].Link(cp_inlet);
link.Transmissivity(trans, "m3/s");
link.HeatTrans(ttrans, "W/K");
link.FlowRate(flow, "m3/s");

for (int i=1; i<nx; i++) {
   link = cp[i].Link(cp[i-1]);
   link.Transmissivity(trans, "m3/s");
   link.HeatTrans(ttrans, "W/K");
   link.FlowRate(flow, "m3/s");
}

link = cp[nx-1].Link();
link.FlowRate(-flow, "m3/s");

// Time marching loop.
while (true) {
   double deltat = 1e99, deltat_world;
   for (int i=0; i<nx; i++)
      deltat = std::min(deltat, cp[i].ReportTimeStep());
   MPI_Allreduce(&deltat, &deltat_world, 1,
                        MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTimeStep(deltat_world)) return exit_client(0);

   for (int i=0; i<nx; i++)
      for (int l=0; l<cp[i].nLinks(); l++)
         if (cp[i].Link(l).MpiUpdateLink()) return exit_client(-1);

   for (int i=0; i<nx; i++)
      if (cp[i].AdvanceTransport()) return exit_client(-1);
```

```
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceHeatTransport()) return exit_client(-1);

        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) return exit_client(-1);

        write_results(cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Hybrid Parallelization

A hybrid client is a cluster computing program in which each program copy in a communication group is multithreaded. As such, a hybrid client takes advantage of two types of parallelism at once: it splits the work across the machines in a cluster, and spreads each machine's work over its computing cores.

Once you feel comfortable with the previous two chapters, **Multithreading** and **Cluster Computing**, coding a hybrid client is straightforward. If you are already working on a computing cluster, the upside to multithreading your app is significant, and there is little practical reason not to do so.

We develop in this chapter a hybrid client program "RTM4.cpp", which is set out almost entirely as a line-by-line interweaving of the OpenMP client "RTM2.cpp' from the **Multithreading** chapter with the MPI client "RTM3.cpp" from the previous chapter, **Cluster Computing**.

In laying out the hybrid MPI/OpenMP client in this way, two considerations come into play. First, loops that create links with the "Link()" member function, those that update links by calling "MpiUpdateLink()", as well as loops calling any of the "Report()" family of functions may not be multithreaded; instead the loops need to run serially on each client copy. This requirement arises because under MPI the functions need to be called in a specific order, and OpenMP does not preserve loop order within a parallel region.

Second, care should be taken in scheduling OpenMP loops, to achieve optimal speedup. This point is addressed on the next section.

16.1 Loop scheduling

Recall from the **Loop scheduling** section of the **Multithreading** chapter that OpenMP by *de facto* default employs static scheduling. In a static schedule, a multithreaded app splits the passes through a loop into contiguous chunks, sending each chunk to run on one of the cores. Take, for example, a client running on a computer with eight cores that has spawned 32,000 ChemPlugin instances. OpenMP, by default, would divide the instances into chunks of 4,000; core zero would be responsible the first 4,000, core one for the next 4,000, and so on.

In cluster computing, such an arrangement may work poorly, because each client copy does work on only a fraction of the ChemPlugin instances, and those instances

may not be distributed among the chunks. If a client copy from a communication group of 32 ran the loop

```
#pragma omp parallel for schedule(static)
   for (int i=0; i<nx; i++)
      cp[i].Config(cmd);
```

where "nx" was 32,000, for example, each copy would be responsible for 1,000 of the instances. Following a static schedule, those 1,000 instances would fall within a single chunk of 4,000 loop passes, so on each client copy only one of the cores would do work.

An effective strategy to avoid such a situation is to set the OpenMP loop to work in smaller chunks. In our example, if we were to divide the loop into chunks of 125 passes, each of the eight cores would end up working on its share of the 1,000 instances assigned to the client copy. The code

```
   int chunk_size = nx / world_size / omp_get_max_threads();

#pragma omp parallel for schedule(static, chunk_size)
   for (int i=0; i<nx; i++)
      cp[i].Config(cmd);
```

divides the number of instances by the group size, and then by the number of cores on the computer to arrive at a chunk size that allocates work evenly.

16.2 Running the example

The procedure for compiling and executing a hydrid ChemPlugin client on a computing cluster is commonly similar to that for running a simple MPI client, as described in the **Running the example** section from the **Cluster Computing** chapter. You should consult your system administrator for detailed information.

16.3 C++ source code

The full C++ code for the client program is available on the ChemPlugin.GWB.com website as file "RTM4.cpp", and is listed below.

Note: The code is also available in FORTRAN from ChemPlugin.GWB.com.

```
#include <iostream>
#include <fstream>
#include <string>
#include <omp.h>
#include <vector>
#undef min
#include <algorithm>
#include "ChemPlugin.h"
```

```
#include <mpi.h>

int exit_client(int status)
{
   if (status != 0)
      MPI_Abort(MPI_COMM_WORLD, status);
   MPI_Finalize();
   return status;
}

void open_input(std::ifstream& input, int argc, char** argv) {
   std::string filename;
   if (argc < 2) {
      std::cerr << "You must specify an input file as the first program argument."
                << std::endl;
      exit_client(-1);
   }
   else {
      filename = argv[1];
   }

   input.open(filename);

   if (!input.is_open()) {
      std::cerr << "The input file does not exist" << std::endl;
      exit_client(-1);
   }
}

void write_results(std::vector<ChemPlugin>& cp, int nx, double gap, double& then)
{
   double now = cp[0].MpiReport1("Time", "years");
   int step = nx / 100;
   if ((then - now) < gap / 1e4) {
      int last_rank = cp[0].MpiRank();
      for (int i=0; i<nx; i+=step) {
         if (cp[i].MpiRank() != last_rank) {
            last_rank = cp[i].MpiRank();
            MPI_Barrier(MPI_COMM_WORLD);
         }
         std::string label = "Instance " + std::to_string(i);
         cp[i].PrintOutput("RTM.txt", label);
      }
      then += gap;
   }
}

int main(int argc, char** argv) {
```

```
   // Initialize MPI
   MPI_Init(NULL, NULL);
   // Get the number of processes
   int world_size;
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   // Get the rank of this process
   int world_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

   if (world_rank == 0)
      std::cout << std::endl
            << "Model reactive transport in one dimension using MPI and OPENMP"
            << std::endl << std::endl;

   // Simulation parameters.
   int nx = 4000; // number of instances along x
   double length = 100; // m
   double deltax = length / nx; // m
   double deltay = 1.0, deltaz = 1.0; // m

   double time_end = 10.0;  // years
   double delta_years = time_end / 5; // years
   double next_output = 0.0;  // years

   // Schedule OpenMP to split work evenly among the threads.
   int chunk_size = nx / world_size / omp_get_max_threads();

   // Create the inlet and interior instances.
   ChemPlugin cp_inlet;
   std::vector<ChemPlugin> cp(nx);

   // Assign instances to ranks
   cp_inlet.MpiAssign(0);
#pragma omp parallel for schedule(static, chunk_size)
   for (int i=0; i<nx; i++)
      cp[i].MpiAssign(i * world_size / nx);

   cp_inlet.Console("stdout");
   cp[0].Console("stdout");
   cp[0].Config("pluses = banner");

   std::string cmd = "volume = " + std::to_string(deltax * deltay * deltaz) +
                     " m3; time end = " + std::to_string(time_end) + " years";
#pragma omp parallel for schedule(static, chunk_size)
   for (int i=0; i<nx; i++)
      cp[i].Config(cmd);
```

```
   // Set input file and configure inlet, initial domain.
   std::ifstream input;
   open_input(input, argc, argv);
   int scope = 0;
   while (!input.eof()) {
      std::string line;
      std::getline(input, line);
      if (line == "go")
         break;
      else if (line == "scope inlet")
         scope = 1;
      else if (line == "scope initial")
         scope = 2;
      else if (scope == 1)
         cp_inlet.Config(line);
      else if (scope == 2)
#pragma omp parallel for schedule(static, chunk_size)
         for (int i=0; i<nx; i++)
            cp[i].Config(line);
   }

   // Initialize the inlet and interior instances; write out initial conditions.
   if (cp_inlet.Initialize()) {
      std::cout << "Inlet failed to initialize" << std::endl;
      return exit_client(-1);
   }
   cp_inlet.PrintOutput("RTM.txt", "Inlet fluid");

   int nerror = 0;
#pragma omp parallel for reduction(+ : nerror) schedule(static, chunk_size)
   for (int i=0; i<nx; i++) {
      if (cp[i].Initialize()) {
         std::cout << "Instance " << i << " failed to initialize" << std::endl;
         nerror++;
      }
   }
   if (nerror) return exit_client(-1);

   write_results(cp, nx, delta_years, next_output);

   // Set velocity; calculate flow rate and transmissivities.
   double veloc_in; // m/yr
   if (argc < 3) {
      std::cerr << "You must specify fluid velocity in m/yr as the "
                   "second program argument." << std::endl;
      exit_client(-1);
   }
   veloc_in = atof(argv[2]);
```

```
   double velocity = veloc_in / 31557600.;  // m/s
   double porosity = cp[0].MpiReport1("porosity");  // volume fraction
   double flow = deltay * deltaz * porosity * velocity;  // m3/s

   double diffcoef = 1e-10;  // m2/s
   double dispersivity = 1.0; // m
   double dispcoef = velocity * dispersivity + diffcoef; // m2/s
   double trans = deltay * deltaz * porosity * dispcoef / deltax;  // m3/s

   double tcond = 2.0;  // W/m/K
   double ttrans = deltay * deltaz * tcond / deltax;  // W/K

   // Link the instances.
   CpiLink link = cp[0].Link(cp_inlet);
   link.Transmissivity(trans, "m3/s");
   link.HeatTrans(ttrans, "W/K");
   link.FlowRate(flow, "m3/s");

   for (int i=1; i<nx; i++) {
      link = cp[i].Link(cp[i-1]);
      link.Transmissivity(trans, "m3/s");
      link.HeatTrans(ttrans, "W/K");
      link.FlowRate(flow, "m3/s");
   }

   link = cp[nx-1].Link();
   link.FlowRate(-flow, "m3/s");

   // Time marching loop.
   while (true) {
      double deltat = 1e99, deltat_world;
      int nerror = 0;

#pragma omp parallel for reduction(min : deltat) schedule(static, chunk_size)
      for (int i=0; i<nx; i++)
         deltat = std::min(deltat, cp[i].ReportTimeStep());
      MPI_Allreduce(&deltat, &deltat_world, 1,
                           MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

#pragma omp parallel for reduction(+ : nerror) schedule(static, chunk_size)
      for (int i=0; i<nx; i++)
         if (cp[i].AdvanceTimeStep(deltat_world)) nerror++;
      if (nerror)
         return exit_client(0);

      for (int i=0; i<nx; i++)
         for (int l=0; l<cp[i].nLinks(); l++)
```

```
            nerror += cp[i].Link(l).MpiUpdateLink();
        if(nerror)
            return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror) schedule(static, chunk_size)
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceTransport()) nerror++;
        if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror) schedule(static, chunk_size)
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceHeatTransport()) nerror++;
        if (nerror) return exit_client(-1);

#pragma omp parallel for reduction(+ : nerror) schedule(dynamic)
        for (int i=0; i<nx; i++)
            if (cp[i].AdvanceChemical()) nerror++;
        if (nerror) return exit_client(-1);

        write_results(cp, nx, delta_years, next_output);
    }

    // Never gets here.
    return 0;
}
```

Appendix: ChemPlugin Setup

This appendix describes how to set up a client program to use ChemPlugin objects. Specifically, we discuss installing the software and how to include ChemPlugin objects in the client program.

A.1 Preliminaries

To begin, you need to install a version of the GWB software that includes the ChemPlugin object. You also need to locate or install on your computer an appropriate software development environment.

A.1.1 Install ChemPlugin

Install ChemPlugin by double-clicking on the installer for an appropriate version of the GWB. You will be asked to choose between a 32-bit or a 64-bit version of the software. The choice is significant as the client program can only access the ChemPlugin library if the client program is built for the same bit version.

A 64-bit installation is most common, because it produces apps that run more quickly and with fewer memory constraints than 32-bit apps. The 32-bit version of the ChemPlugin object, on the other hand, is up to about 20% smaller in terms of its memory footprint, since the memory addresses it holds are half the size of those in the 64-bit object.

You can install both the 32-bit and 64-bit versions by running the installer twice. The second time, be sure to uncheck the flag for automatically uninstalling the first version.

The 64-bit version of ChemPlugin is installed by default under "C:\Program Files\ChemPlugin", and the 32-bit version under "C:\Program Files (x86)\ChemPlugin". If you install ChemPlugin somewhere else, you need to keep track of the installation directory, because you will need to reference it when setting up the development environment.

Within the top level of the installation directory—i.e., "C:\Program Files\ChemPlugin"— you will find a file "chemplugin.dll", which is the link library containing the ChemPlugin object itself. You will also find file "ChemPlugin.lib", which is a map of the library used by the link editor.

The cluster computing version of the ChemPlugin library is contained in file "chemplugin_mpi.dll" on Windows, or "chemplugin_mpi.so" under Linux.

The ChemPlugin wrapper files (which provide the link between the client program and ChemPlugin library), are installed within subdirectory "src" within the installation directory. The wrapper files are:

C++	"ChemPlugin.h"
FORTRAN	"ChemPlugin.f90"
Python	"ChemPlugin.py",

A client program written in C++ pulls in the wrapper file "ChemPlugin.h", a FORTRAN client references "ChemPlugin.f90", and so on.

A.1.2 Launch development environment

You need to check that a development environment for the language in which your client program is written is installed on your computer. You might need to install a C++ or FORTRAN compiler, for example, and its associated link editor. A few common choices include:

	C++	FORTRAN	Link editor
Intel	icl	ifort	xilink
Microsoft	cl	—	cl
Gnu	gcc	gfortran	ld

You might alternatively install a Python interpreter.

In any case, you can open the development environment as a command line prompt, or work from the Windows command line prompt. We'll assume in the sections below that you are working from the command line.

You may prefer to work within a GUI development environment, such as Visual Studio, rather than from the command prompt. The details of doing so differ depending on the environment you choose, but the principles are the same as working from the command line.

A.2 Running a Client Program

To show how to run client programs written in the various languages ChemPlugin supports, we in this section take as an example program "RTM1" from the **Reactive Transport Model** chapter. The versions of program "RTM1" can be found on the ChemPlugin.GWB.com website, or in the "CPI_clients" subfolder of the ChemPlugin installation directory.

Regardless of the language in use, Windows needs to know where to find the ChemPlugin library "chemplugin.dll", and various other libraries that library depends on. To accomplish this, we need to append the name of the installation directory to the "PATH" environmental variable. Depending on the version of the GWB software installed, the path may be "C:\Program Files\ChemPlugin", as assumed here, or "C:\Program Files\Gwb".

You can modify the copy of PATH used computer-wide under Windows from the Control Panel. Alternatively, from the command line environment, issuing the command

```
// For 64-bit app
set path=C:\Program Files\ChemPlugin;%path%

OR

//For 32-bit app
set path=C:\Program Files (x86)\ChemPlugin;%path%
```

sets the variable locally, for the current environment only.

A.2.1 C++

Invoking the Intel compiler “icl”, the command

```
icl -c RTM1.cpp -I"C:\Program Files\ChemPlugin\src"
```

compiles the program, looking to resolve the file “ChemPlugin.h” in “C:\Program Files\ChemPlugin\src”. The compilation step produces an object file “RTM1.obj” that can be linked against the “ChemPlugin.lib” map with the command

```
xilink RTM1.obj "C:\Program Files\ChemPlugin\ChemPlugin.lib"
```

The link step produces the executable program “RTM1.exe”, which can be run, as shown in the next section.

In the case of “icl”, the compiling and linking can be combined into a single step with the command

```
icl RTM1.cpp -I"C:\Program Files\ChemPlugin\src"
                                    "C:\Program Files\ChemPlugin\ChemPlugin.lib"
```

Again, the executable is written to “RTM1.exe”.

Compiling an OpenMP program such as “RTM2.cpp”, described in the **Multithreading** chapter, requires an additional keyword in order to produce a multithreaded executable. Under the Intel environment, the command to compile this client is

```
icl -c RTM2.cpp -Qopenmp -I"C:\Program Files\ChemPlugin\src"
```

and the command

```
icl RTM2.cpp -Qopenmp -I"C:\Program Files\ChemPlugin\src"
                                    "C:\Program Files\ChemPlugin\ChemPlugin.lib"
```

compiles and links the client in one step.

Running the program is simply a matter of typing in the name of the application

```
rtm1
```

or double-clicking on “rtm1.exe” in Windows Explorer.

A.2.2 FORTRAN

Invoking the Intel compiler “ifort”, the command

```
ifort RTM1.f90 -I"C:\Program Files\ChemPlugin\src"
                                    "C:\Program Files\ChemPlugin\ChemPlugin.lib"
```

compiles the client and links the ChemPlugin library. It produces the executable “RTM1.exe”.

Run the program by typing the name of the application

```
rtm1
```

or double-clicking on “rtm1.exe” in Windows Explorer.

A.2.3 Python

To import the ChemPlugin class in a Python script, it needs to know the location of “ChemPlugin.py” installed in the "src" directory of ChemPlugin installation. This is most easily accomplished by selecting the option for “Set user PATH and PYTHONPATH environment variables” when installing GWB.

PYTHONPATH can also be set manually with the commands

```
# Add the location of the Python wrapper file ChemPlugin.py
# to PYTHONPATH environment variable
set PYTHONPATH=C:\Program Files\ChemPlugin\src;%PYTHONPATH%
```

You can now run the example from the command line:

```
# run the example with Python
python RTM1.py
```

Appendix: Member Functions

This chapter describes the ChemPlugin member functions. The member functions allow a client to control instances of the ChemPlugin class. A client uses member function “Config()”, for example, to send configuration commands to an instance.

Given a reference “cp” to a ChemPlugin instance, the statement

```
// Syntax for C++ or Python
cp.Config("pH = 5");

! FORTRAN
Config(cp, "pH = 5")
```

passes a command telling instance “cp” to set initial pH to 5.

The ChemPlugin class works together with a class named CpiLink that represents connections between pairs of ChemPlugin instances. CpiLink carries member functions of its own. Whereas the ChemPlugin member functions act on ChemPlugin instances individually, CpiLink’s member functions act on connections between ChemPlugin instances.

For example,

```
// Syntax for C++
int cpi.Config(...)
int link.FlowRate(...)
```

We recognize “Config()” as a member of the ChemPlugin class and “FlowRate” as a member of class CpiLink.

Many of the member functions return an integer. In these cases, the return value is zero if the operation was successful; a non-zero value indicates the operation failed. In the latter case, diagnostic messages are generally written to the instance’s console output (see **Console messages** in the **Overview** chapter of this guide).

In the following subsections, we will discuss the functionality and syntax of each member function for all supported languages separately.

B.1 C++

Create an instance called "cp" of Chemplugin using the object's constructor

```
ChemPlugin cp;
```

The constructor can accept two optional arguments, as described in the **Overview** chapter. The first argument defines the console output stream, whereas the second is used to set option flags. For example, the statement

```
ChemPlugin cp("stdout", "-d mythermo.tdat -s mysurface.sdat");
```

creates an instance that sends its console output to the standard output, takes thermodynamic data from "mythermo.tdat", and reads surface data from "mysurface.sdat".

The cluster computing version of ChemPlugin accepts as an optional third argument the instance's rank, which identifies the client copy responsible for the instance. Setting the argument, you conflate the constructor and "MpiAssign()" calls into a single statement.

Note that any member function that accepts a character pointer (char*) as an argument can also accept a C++ standard string. Hence, the statements

```
ChemPlugin cp;
double pH = 9;
std::string command = "pH = " + std::to_string(pH);
cp.Config(command);
```

serve the same purpose as

```
ChemPlugin cp;
double pH = 9;
char command[128];
sprintf(command, "pH = %f", pH);
cp.Config(command);
```

If for some reason you wish to disable acceptance of C++ standard strings, use

```
#define ALLOW_STD_STRING 0
#include "ChemPlugin.h"
```

in the client's header.

B.1.1 Configuring and initializing instances

Member functions “Config()” and “Initialize()” let a client configure a ChemPlugin instance and prepare it for the start of a reaction simulation.

B.1.1.1 Config()

Syntax:

```
int cpi.Config(const char* command)
```

Use member function “Config()” to pass configuration commands to a ChemPlugin instance. The configuration commands and their syntax are listed in the **Configuration Commands** chapter of this guide. Examples:

```
char *cmd = "pH = 5";
cp.Config(cmd);
cp.Config("Na+ = 1 mmol/kg");
```

A client can pass multiple commands with a single function call, or continue a command onto another call

```
cp.Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg");
cp.Config("kinetic Quartz \");
cp.Config("rate_con = 2e-12, surface = 1000");
```

if it separates commands with semicolons, and marks incomplete commands with a trailing backslash.

A zero-value return indicates the command processed successfully.

B.1.1.2 Initialize()

Syntax:

```
int cpi.Initialize()
int cpi.Initialize(double end_time)
int cpi.Initialize(double end_time, const char* units)
```

The “Initialize()” member function triggers an instance to calculate its initial condition, including the distribution of mass across the various chemical species, and to prepare the instance to begin time marching.

The member function is commonly called without arguments, but the client can use the call to specify the end time of the simulation. For example, the statement

```
cp.Initialize(10.0, "years");
```

has the same effect as

```
cp.Config("time end = 10 years");
cp.Initialize();
```

An end time of zero is ignored, and the default unit for time is seconds. A non-zero return indicates the instance was unable to initialize. In this case, diagnostic messages are written to the instance's console output.

B.1.2 Linking instances

Member function "Link()" creates a link connecting two ChemPlugin instances, functions "Unlink()" and "ClearLinks()" remove links that have been created, and the function "nLinks()" reports the number of existing links. Similarly, function "Outlet()" creates an open-ended link from a ChemPlugin instance, and "nOutlets()" reports how many such links exist.

A client can create any number of links between two instances. For example, you might wish to create a link carrying fluid from an instance "cp1" to another, "cp2". The client could then create a second link to account for the possibility of back-flow.

Each of the functions in this section are reciprocal in operation. In other words, if a client links "cp2" to "cp1", both instances know about the link. You should then link "cp1" to "cp2" only if you wish to spawn a second link between the instances. Similarly, when the client unlinks "cp2" from "cp1", both instances are aware the link has been removed.

B.1.2.1 Link()

Syntax:

```
CpiLink cpi.Link(ChemPlugin another_cpi)
CpiLink cpi.Link()
CpiLink cpi.Link(int index)
```

The "Link()" member function connects two ChemPlugin instances. For example, the statements

```
ChemPlugin cp1, cp2;
cp1.Link(cp2);
```

link two ChemPlugin instances, "cp1" and "cp2". Once this statement is executed, there is no need to execute the reciprocal operation

```
cp2.Link(cp1);
```

Doing so, in fact, would create an additional link.

The member function returns a reference to the link, which a client can store for later operations. For example, the statements

```
CpiLink link1 = cp1.Link(cp2);
link1.FlowRate(0.2, "m3/s");
```

set a flow rate of 0.2 m^3 s^{-1} from “cp2” to “cp1”.

When a client calls “Link()” without an argument, the function creates an open link that functions as a free outlet. This syntax is an alternative to the “Outlet()” member function described in the next subsection.

When a client passes “Link()” an index, rather than a reference to a ChemPlugin instance, the member function returns a reference to the link in question. If, for example, we create two links

```
cp1.Link(cp2);
cp1.Link(cp3);
```

the links will have indices 0 and 1, respectively, assuming no earlier links exist. Then, the statement

```
CpiLink link1 = cp1.Link(1);
```

returns a reference to the second link, to be stored in “link1”.

B.1.2.2 Outlet()

Syntax:

```
CpiLink cpi.Outlet()
```

The “Outlet()” member function creates an open link that functions as a free outlet. A call to “Outlet()” is the same as calling “Link()” without an argument.

For example, the statement

```
CpiLink link1 = cp.Outlet();
```

creates an open link to which “link1” refers.

Note: Water can flow outward along the link, away from “cp”, but not inward toward the instance; no first-order transport, such as diffusion or heat conduction, is possible across an open link.

B.1.2.3 Unlink()

Syntax:

```
int link.Unlink()
int cpi.Unlink(ChemPlugin another_cpi)
int cpi.Unlink(int index)
```

A client can remove a link between two nodes using the "Unlink()" member function of either the CpiLink or ChemPlugin class. In the latter case, it can refer to the link either by index or reference to the linked instance.

Some examples:

```
ChemPlugin cp1, cp2;
CpiLink link1 = cp1.Link(cp2);
... some code ...
link1.Unlink();
    or
cp1.Unlink(cp2);
    or
cp1.Unlink(0);
```

Each of the three examples serves the same purpose. A zero-value return indicates success.

B.1.2.4 ClearLinks()

Syntax:

```
int cpi.ClearLinks()
```

The "ClearLinks()" member function removes all links to a ChemPlugin instance. Example:

```
cp.ClearLinks()
```

A zero-value return indicates success.

B.1.2.5 nLinks()

Syntax:

```
int cpi.nLinks()
int cpi.nLinks(ChemPlugin another_cpi)
```

Member function "nLinks()" returns the number of links that have been made to a ChemPlugin instance.

When a client calls the function without an argument, it reports the total number of links to the instance, including open links. For example, the statements

```
cp1.Link(cp2);
cp1.Link(cp3);
int m = cp1.nLinks();
```

result in a value of 2 being stored in variable "m".

Calling the function with a second ChemPlugin instance as an argument yields the number of links to the second instance. Continuing the previous example, the statement

```
... cont'd ...
int n = cp1.nLinks(cp3);
```

returns to “n” a value of one.

B.1.2.6 nOutlets()

Syntax:

```
int cpi.nOutlets()
```

Member function “nOutlets()” returns the number of open links connected to a ChemPlugin instance.

B.1.3 Transport across links

Member functions “FlowRate()”, “Transmissivity()”, and “HeatTrans()” control how chemical mass and heat energy are transported across links. In each case, calling the member function with a numerical value and optionally specifying units sets the quantity in question. Calling the function without a numerical value, in contrast, returns the current setting in the units specified, or in the default units.

B.1.3.1 FlowRate()

Syntax:

```
int link.FlowRate(double flow)
int link.FlowRate(double flow, const char* unit)
double link.FlowRate()
double link.FlowRate(const char* unit)
```

Use member function “FlowRate()” to set the rate at which water flows across a link. The default unit for flow rate is $m^3\ s^{-1}$, but a client can specify any other unit of volume flow, as listed in the **Units Recognized** appendix.

Calling “FlowRate()” without a numeric argument returns the current value for flow rate, as set by the most recent call to the function. When a link is created, the flow rate is guaranteed to be zero-value initially.

By ChemPlugin convention, flow into an instance is positive, and outward flow is negative in sign. Hence, the statements

```
CpiLink link1 = cp1.Link(cp2);
link1.FlowRate(2.0e6, "cm3/s")
flow = link1.FlowRate()
```

result in a value of 2.0 being stored in variable "flow", since this quantity is 2×10^6 cm^3 s^{-1}, expressed in m^3 s^{-1}. In this case, flow is positive, so water flows from "cp2" to "cp1".

B.1.3.2 Transmissivity()

Syntax:

```
int link.Transmissivity(double trans)
int link.Transmissivity(double trans, const char* unit)
double link.Transmissivity()
double link.Transmissivity(const char* unit)
```

The "Transmissivity()" member function sets the transmissivity describing mass transport across a link by first-order processes, such as diffusion, dispersion, and turbulent mixing. The definition of the mass transmissivity is described in this User's Guide, in the **Flow and Transport** chapter.

A client sets a value in units of m^3 s^{-1} by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial transmissivity is guaranteed to be zero-value.

B.1.3.3 HeatTrans()

Syntax:

```
int link.HeatTrans(double trans)
int link.HeatTrans(double trans, const char* unit)
double link.HeatTrans()
double link.HeatTrans(const char* unit)
```

The "HeatTrans()" member function sets the thermal transmissivity describing the transport of heat energy across a link by first-order processes, such as conduction, thermal dispersion, and turbulent mixing. Thermal transmissivity is described in the **Flow and Transport** chapter in this User's Guide.

A client sets a value in units of J K^{-1} s^{-1} by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial thermal transmissivity is guaranteed to be zero-value.

B.1.4 Time marching loop

Member functions "ReportTimeStep()", "AdvanceTimeStep()", "AdvanceTransport()", "AdvanceHeatTransport()", and "AdvanceChemical()" work together to make up a time marching loop. The function "ExtendRun()" can be used to prolong a time marching loop to a new end time, once the initial loop is complete.

B.1.4.1 ReportTimeStep()

Syntax:

```
double cpi.ReportTimeStep()
double cpi.ReportTimeStep(char *unit)
```

Member function “ReportTimeStep()” returns the largest time step an instance may take if it is to maintain numerical stability and honor any constraints the client program may have prescribed. The limiting time step is by default returned in seconds, but the optional argument allows a client to specify an alternative unit of time.

A client program, upon beginning a pass through a time marching loop, will in general query each ChemPlugin instance the program has spawned. The client program should then take the least of the values returned and use that value as the length Δt of the current time step.

B.1.4.2 AdvanceTimeStep()

Syntax:

```
int cpi.AdvanceTimeStep(double deltat)
int cpi.AdvanceTimeStep(double deltat, char *unit)
```

Member function “AdvanceTimeStep()” moves the current time level carried in an instance forward by the time step specified, adds or removes simple reactants, and adjusts sliding reactants. By default, the time step is provided in seconds, but the optional second argument lets a client set an alternative unit of time.

B.1.4.3 AdvanceTransport()

Syntax:

```
int cpi.AdvanceTransport()
```

Member function “AdvanceTransport()” triggers the instance to evaluate the effect of mass transport on the instance’s chemical composition over the course of the current time step.

B.1.4.4 AdvanceHeatTransport()

Syntax:

```
int cpi.AdvanceHeatTransport()
```

Member function “AdvanceHeatTransport()” triggers the instance to evaluate the effect of heat transport on the instance’s temperature over the course of the current time step.

B.1.4.5 AdvanceChemical()

Syntax:

```
int cpi.AdvanceChemical()
```

Member function "AdvanceChemical()" causes the instance to evaluate the effect of kinetic and equilibrium reactions on an instance's chemical state over the course of the current time step.

B.1.4.6 SlideFugacity()

Syntax:

```
int cpi.SlideFugacity(const char* gas_name, double value);
```

Use member function "SlideFugacity()" to adjust the fugacity of a fixed-fugacity gas within an instance, once the instance has been initialized. The "gas_name" is the name of the gas in question (e.g., "CO2(g)") and "value" is the revised fugacity of that gas.

In order to use the "SlideFugacity()" function, the fugacity of the gas in question must be fixed in the instance configuration. For example:

```
cpi.Config("fix fugacity CO2(g)");
```

Note the function, despite its name, can be used also to adjust the fixed activity of an aqueous species, or an activity ratio that has been fixed.

B.1.4.7 SlideTemperature()

Syntax:

```
int cpi.SlideTemperature(double temperature, const char* unit);
```

Use member function "SlideTemperature()" to adjust an instance's temperature, once it has been initialized. The "unit" field is optional and defaults to "C".

Do not use "SlideTemperature()" to set an instance's initial temperature; instead, call

```
cpi.Config("temperature = 45 C");
```

to configure the instance at the desired temperature, before initializing it.

B.1.4.8 ExtendRun()

Syntax:

```
int cpi.ExtendRun(double add_time)
int cpi.ExtendRun(double add_time, const char* unit)
```

Use member function "ExtendRun()" to extend a reaction simulation, once time marching is complete. The first argument is the amount of time to be added to the simulation,

in the unit specified as the second argument, or in seconds, by default. The function returns a zero value upon successful extension of the run.

By extending a simulation's time range, you extend the range in reaction progress to be traversed. Tracing a reaction path that spans 10 years, for example, carries the reaction progress variable ξ from zero to one. If you were to then add 20 years to the simulation, ξ would, over the course of the second round of time marching, increase from one to three.

B.1.5 Retrieving results

A client program uses the "Report()", "Report1()", "Report1i()", and "Report1c()" member functions to gather information about the current state of a ChemPlugin instance.

B.1.5.1 Report()

Syntax:

```
int cpi.Report(void *target, const char* keywords, const char *unit)
```

Use member function "Report()" to retrieve calculation results from a ChemPlugin instance. Here, "target" is the address in memory to which the results are to be written. The "keywords" argument identifies the specific result or results to be written, and the optional "unit" argument sets the unit in which the results are to be cast. The function returns the number of values copied to "target".

Whenever the function is unable to fulfill a request, such as when an impossible unit conversion has been requested, it fills the corresponding location or locations in "target" with parameter "ANULL", defined in "ChemPlugin.h".

The options for specifying "keywords" are listed in the **Report Function** appendix to this User's Guide, and the available units are shown in the **Units Recognized** appendix.

If a client passes NULL as the "target" argument, "Report()" returns the number of values scheduled to be copied, without actually copying the values.

"Report()" is used to retrieve an array of data, such as a vector of the concentrations of a group of aqueous species. To retrieve a single value of type double, such as the pH or a species' concentration, a client may instead call the shorter "Report1()" function. For a single integer or character string, use "Report1i()" or "Report1c()", respectively.

Please refer to the **Retrieving Results** chapter of this User's Guide for detailed information about the "Report()" and "Report1()" functions, including several examples of their use.

B.1.5.2 Report1(), Report1i(), and Report1c()

Syntax:

```
double cpi.Report1(const char* keywords, const char *unit)
int cpi.Report1i(const char* keywords)
char* cpi.Report1c(const char* keywords)
```

Use member function "Report1()" to retrieve from a ChemPlugin instance a single value of type double, such as the pH or the concentration of a specific aqueous species. The "keywords" argument identifies the specific result or results to be written, and the optional "unit" argument sets the unit in which the results are to be cast. Similarly, "Report1i()" and "Report1c()" retrieve an integer value and the pointer to a character string, respectively.

The options for specifying "keywords" are listed in the **Report Function** appendix to this User's Guide, and the available units are shown in the **Units Recognized** appendix. The function returns the retrieved value.

When "Report1()" fails, it returns a value of "ANULL", defined in "ChemPlugin.h"; "Report1i()" returns "ANULL" cast as an integer, and "Report1c()" returns a NULL pointer.

Please refer to the **Retrieving Results** chapter of this User's Guide for detailed information about the "Report()" and "Report1()" functions, including several examples of their use.

B.1.6 Output streams

A client program controls the output streams from a ChemPlugin instance with member functions "Console()", "PrintOutput()", "PlotHeader()", "PlotBlock()", and "PlotTrailer()". The first function is described in the **Overview** chapter of this User's Guide, and use of the latter four functions is explained in detail in the **Direct Output** chapter.

It is important avoid allowing more than one ChemPlugin instance to write into the same dataset. In such a situation, the output from the instances would appear intermingled and probably unintelligible.

B.1.6.1 Console()

Syntax:

```
int cpi.Console()
int cpi.Console(const char* stream)
```

Member function "Console()" controls where an instance's console output is directed. Console output consists of routine messages an instance produces as it initializes and undetakes calculations, as well as any warning and error messages that may be generated. By default, a ChemPlugin instance does not produce console output.

The function's optional argument is the target for the console stream, which may be "stdout", "stderr", or the name of a dataset. When a client calls the function without an argument, or with NULL or an empty string as the argument, output to the console stream is disabled. A client may enable, disable, or redirect an instance's console output at any time.

A client can direct an instance's console output at declaration:

```
ChemPlugin cp("stdout");
```

Since an instance produces no console output until directed to do so, this is the only way to capture messages produced when an instance comes into scope and initializes.

B.1.6.2 PrintOutput()

Syntax:

```
int cpi.PrintOutput()
int cpi.PrintOutput(const char *basename)
int cpi.PrintOutput(const char *basename, const char *label, bool rewind)
```

Calling member function “PrintOutput()” triggers a ChemPlugin instance to append a data block to a print-format dataset.

The optional argument sets the file’s base name. The full file name is the base name combined with any suffix that may be set. For example, the calls

```
cp.Config("suffix _1");
cp.PrintOutput("myPrint.txt")
```

cause a block of output to be written to dataset “myPrint_1.txt”.

When a client calls “PrintOutput()” without an argument, the instance appends a data block to whatever file is open for print-format output. If none is open, the instance writes to “ChemPlugin_output.txt”.

The optional “label” argument allows the program to pass an optional character string to be written into the print dataset, at the head of the data block. For example,

```
cp.PrintOutput(NULL, "Initial condition");
```

would write the string “Initial condition” and then a data block to the currently open dataset.

Passing a true (non-zero) value as the optional third argument causes the instance to rewind the print-format dataset and write a data block at the head of the file. Use

```
cp.PrintOutput(NULL, NULL, true);
```

to write a data block at the head of the currently open dataset.

B.1.6.3 PlotHeader()

Syntax:

```
int cpi.PlotHeader()
int cpi.PlotHeader(const char* basename)
```

Member function “PlotHeader()” opens and initializes a plot-format dataset by writing header information to it.

A client may specify the file’s base name as a character string; the full name is the base name combined with the suffix, if one has been set. If a client calls the function

without an argument, it writes to any file that may be open for plot output. If none is open, the function opens "ChemPlugin_plot.gtp", accounting for a suffix, if set. The ".gtp" extension identifies the file as **Gtplot** input.

B.1.6.4 PlotBlock()

Syntax:

```
int cpi.PlotBlock()
```

Member function "PlotBlock()" appends to the currently open plot-format dataset a block of data representing an instance's current state. The dataset must have been initialized with a call to "PlotHeader()".

B.1.6.5 PlotTrailer()

Syntax:

```
int cpi.PlotTrailer()
```

Member function "PlotTrailer()" completes the plot-format output dataset by writing the trailing data structure. A trailer is required by program **Gtplot** before it can read the dataset.

When functions "ReportTimeStep()" and "AdvanceTimeStep()" detect time marching is complete, they automatically write a trailer to any open plot dataset, and close the dataset. It is not necessary to write a trailer, then, at the end of a time marching loop.

Significantly, a client may continue to append data blocks to a plot dataset, even after it has used "PlotTrailer()" to write a trailer to it. The additional data blocks overwrite the trailer data, so the client needs to call "PlotTrailer()" once again to close the dataset.

B.1.7 Convenience

B.1.7.1 Version()

Syntax:

```
const char* cpi.Version()
```

Member function "Version()" returns a pointer to a character string identifying the version of ChemPlugin in use.

B.1.7.2 ConvertUnit()

Syntax:

```
double cpi.ConvertUnit(double value, const char* old_unit, const char* new_unit)
double cpi.ConvertUnit(double value, const char* old_unit, const char* new_unit,
                       double mw, double mv, double z)
double cpi.ConvertUnit(double value, const char* old_unit, const char* new_unit,
                       double mw, double mv, double z,
                       double wmass, double smass, double dens, double vbulk)
```

Member function “ConvertUnit” converts values from one unit to another. The **Units Recognized** appendix to this User’s Guide lists the units available for conversion.

There are three variants to the function. In the simplest variant, a client passes only the value to be converted, along with the old and new units, and the function returns the converted value.

Unit conversions involving mass and concentration may require additional information. To convert concentration from mmol/kg to mg/kg, for example, the function needs to know molecular weight. For other conversions, the function may require the molar volume, electrical charge on the species (to convert to and from meq/kg, for example), the mass of solvent water, the solution mass, the fluid density, and the bulk volume of the system.

In the second variant of the function call, a client also specifies the mole weight “mw” in g mol^{-1}, the mole volume “mv” in cm^3 mol^{-1}, and the ion charge “z” on the species in question. The function takes water mass (kg), solution mass (kg), density (g cm^{-3}), and bulk volume (cm^3) from the current state of the ChemPlugin instance.

In the final form, the client specifies the latter four variables, in the units listed. This variant of the function call can be helpful because it can be used before a client has calculated the initial system, by calling “Initialize()”.

B.1.8 Cluster computing

The MPI version of ChemPlugin includes a number of C++ member functions specific to cluster computing.

B.1.8.1 MpiAssign()

Syntax:

```
int cpi.MpiAssign(int rank)
```

Use the “MpiAssign()” member function to associate the stub of a ChemPlugin instance with a specific client copy. If argument “rank” is the same as the rank of the client copy calling the function, the stub is expanded within that copy into a full ChemPlugin instance, and that copy becomes responsible for calculations involving the instance.

For example,

```
cp.MpiAssign(0);
```

assigns instance “cp” to the first client copy, which has a rank of zero. Afterward, “cp” will reference a full ChemPlugin instance on the first client copy, whereas on the remaining client copies, it will refer to a stub of that instance.

As an alternative to calling MpiAssign(), you can assign rank to a ChemPlugin object using the ChemPlugin constructor, by setting the rank as the optional third argument; see the **Cluster Computing** chapter. Behavior in this case is the same as if the constructor and “MpiAssign()” had been called serially.

B.1.8.2 MpiOnRank()

Syntax:

```
int cpi.MpiOnRank()
```

The "MpiOnRank()" member function returns one when called from a client copy of the same rank as the instance, and zero otherwise.

For example, if instance "cp" is assigned to the first client copy

```
int is_local = cp.MpiOnRank()
```

the value of "is_local" on that copy will be one, but zero on the remaining copies.

B.1.8.3 MpiRank()

Syntax:

```
int cpi.MpiRank()
```

Member function "MpiRank()" returns the rank of the client copy to which an instance is assigned.

For example, if instance "cp" is assigned to the first client copy, executing the statement

```
int rank = cp.MpiRank()
```

will leave "rank" with a value of zero.

B.1.8.4 MpiReport()

Syntax:

```
int cpi.MpiReport(void *target, const char* keywords, const char *unit)
int cpi.MpiReport(void *target, const char* keywords, const char *unit, int collector)
```

A call to "MpiReport()" with two or three parameters behaves identically to calling "Report()", except that it can retrieve information from any ChemPlugin instance, rather than only those local to the client copy making the call. You use this form of the function call when every client copy needs to retrieve information about all of the ChemPlugin instances considered by the communication group.

When you call "MpiReport()" for an instance "cp", the client copy responsible for "cp" broadcasts information to the other client copies, which wait to receive the data. As such, the client copies need to call the function synchronously, i.e., in the same sequence. The function may not be called within a multithreaded loop.

The optional fourth parameter is used to specify the rank of a client copy that will collect the result. You use this form of the function call when you would like a single client copy to assemble information from all of the ChemPlugin instances considered by the communication group.

The function when used in this way does work on only the collector copy and the client copy responsible for “cp”. Called from those copies, the function writes the requested information to the target and returns the number of values written. Of course, the responsible copy may be the same as the collector copy. For the remaining client copies, the function writes no data and returns a value of zero; since nothing is written, no memory need be allocated for “target”.

Setting the collector to “GLOBAL” is equivalent to omitting the fourth argument.

B.1.8.5 MpiReport1(), MpiReport1i(), MpiReport1c()

Syntax:

```
double cpi.MpiReport1(const char* keywords, const char *unit)
int cpi.MpiReport1i(const char* keywords)
char* cpi.MpiReport1c(const char* keywords)

double cpi.MpiReport1(const char* keywords, const char *unit, int collector)
int cpi.MpiReport1i(const char* keywords, int collector)
char* cpi.MpiReport1c(const char* keywords, int collector)
```

Functions “MpiReport1()”, “MpiReport1i()”, and “MpiReport1c()” are the counterparts to “MpiReport()” for returning, respectively, a single floating point or integer value, or a character string.

Called with one or two arguments, the functions behave identically to “Report1()”, “Report1i()”, and “Report1c()”, except they can retrieve information from any ChemPlugin instance, rather than only those local to the client copy making the call. When you set a collector copy, the functions let you assemble information from any ChemPlugin instance within a single client copy, just as “MpiReport()” does. Setting the collector to “GLOBAL” is equivalent to omitting the third argument.

The “MpiReport()” family functions are synchronous and hence need to be called by each client copy in the same sequence; they may not be called within a multithreaded loop.

B.1.8.6 MpiUpdateLink()

Syntax:

```
int link.MpiUpdateLink()
int link.MpiUpdateLink(int task)
```

In a cluster computing application, a local ChemPlugin instance may be linked to an instance assigned to a different client copy. In this case, the corresponding CpiLink object caches information about the foreign instance, making it available to the local copy.

Whenever the state of the foreign instance has changed in an MPI application, its cached information needs to be updated before you call AdvanceTransport() or AdvanceHeatTransport(). The “MpiUpdateLink()” member function of the CpiLink object serves to update the cache.

"MpiUpdateLink()" needs to be called by every client copy in the communication group, and in the same sequence by each copy. The function may not be called from within a multithreaded loop.

A zero-value return from the function indicates success. Note that all client copies need to call this function in the same sequence, to avoid the possibility of entering a gridlocked state.

In specialized circumstances you may wish use the "task" argument to control which parts of the cache are updated. The argument may be

- "CpiLink::EVERYTHING" updates all of the below, the default
- "CpiLink::TEMPERATURE" updates temperature and thermal properties
- "CpiLink::DENSITY" updates fluid density and viscosity
- "CpiLink::CONCENTRATIONS" updates chemical and isotopic composition
- "CpiLink::NODE" updates the position and extent of a linked instance

Options can be combined with the bitwise or operator, e.g., "CpiLink::TEMPERATURE | CpiLink::DENSITY".

An example:

```
ChemPlugin cp1(NULL, NULL, 0), cp2(NULL, NULL, 1);
CpiLink link1 = cp1.Link(cp2);
... some code ...
link1.MpiUpdateLink();
```

B.2 FORTRAN

Conventional syntax for calling an object's member functions in Fortran, for example

```
err = Config(cp, "pH=5")
```

places the object in question, "cp" in this case, as the function's first argument. Compilers that honor the Fortran 2003 and later standards allow a second syntax

```
err = cp%Config("pH=5")
```

known as a type-bound procedure call that is understood more naturally to the object oriented programmer.

ChemPlugin supports both syntaxes. The discussion below emphasizes the latter, but except where noted you can convert to the conventional syntax by moving the object's reference to the first position in the argument list.

That said, a compiler may not recognize the new syntax where the object in question is itself returned by a member function call. For example, you may not be able to cast a call setting flow across a link

```
err = FlowRate(cp%Link(0), 0.2, "m3/s")
```

in the type-bound form

```
err = cp%Link(0)%FlowRate(0.2, "m3/s")
```

because the link in question is the result calling "cp%Link(0)". You can, of course, split the statement above across two lines of code, one a type-bound call to retrieve the link and a second to set the flow rate.

B.2.1 Instantiation

Create a ChemPlugin instance "cp" by either using the pseudo-constructor function

```
TYPE(ChemPlugin) :: cp
cp = ChemPlugin()
```

or calling the constructor subroutine

```
TYPE(ChemPlugin) :: cp
CALL ChemPlugin(cp)
```

The constructor in either case can accept two arguments, as described in the **Overview** chapter. The first argument defines the console output stream, whereas the second is used to set option flags. For example, the statements

```
TYPE(ChemPlugin) :: cp
cp = ChemPlugin("stdout", "-d mythermo.tdat -s mysurface.sdat")
```

create an instance that sends its console output to the standard output, takes thermodynamic data from "mythermo.tdat", and reads surface data from "mysurface.sdat".

The cluster computing version of the "ChemPlugin()" constructor accepts as an optional third argument the instance's rank, which identifies the client copy responsible for the instance. Setting the argument, you conflate the constructor and "MpiAssign()" calls into a single statement.

Once instantiated, you can delete a ChemPlugin instance with

```
CALL destroy(cp)
```

which leaves "cp" as a dangling reference that can be assigned once again by calling the constructor.

B.2.2 Configuring and initializing instances

Member functions "Config()" and "Initialize()" let a client configure a ChemPlugin instance and prepare it for the start of a reaction simulation.

B.2.2.1 Config()

Syntax:

```
FUNCTION Config(plugin, command) RESULT(retval)
  CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
  CHARACTER(LEN=*), INTENT(in) :: command
  INTEGER(C_INT) :: retval
```

Use member function "Config()" to pass configuration commands to a ChemPlugin instance. The configuration commands and their syntax are listed in the **Configuration Commands** chapter of this guide. Examples:

```
err = cp%Config("pH=5")
err = cp%Config("Na+ = 1 mmol/kg")
```

A client can pass multiple commands with a single function call, or continue a command onto another call

```
err = cp%Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg")

err = cp%Config("kinetic Quartz \")
err = cp%Config("rate_con = 2e-12, surface = 1000")
```

if it separates commands with semicolons, and marks incomplete commands with a trailing backslash.

A zero-value return indicates the command processed successfully.

B.2.2.2 Initialize()

Syntax:

```
FUNCTION Initialize(plugin, time_end, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    REAL(8), INTENT(in), OPTIONAL :: time_end    ! end time of the simulation
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT) :: retval
```

The “Initialize()” member function triggers an instance to calculate its initial condition, including the distribution of mass across the various chemical species, and to prepare the instance to begin time marching.

The member function is commonly called without arguments, but the client can use the call to specify the end time of the simulation. For example, the statement

```
err = cp%Initialize(10.0, "years")
```

has the same effect as

```
err = cp%Config("time end = 10 years")
err = cp%Initialize()
```

An end time of zero is ignored, and the default unit for time is seconds. A non-zero return indicates the instance was unable to initialize. In this case, diagnostic messages are written to the instance’s console output.

B.2.3 Linking instances

Member function “Link()” creates a link connecting two ChemPlugin instances, functions “Unlink()” and “ClearLinks()” remove links that have been created, and the function “nLinks()” reports the number of existing links. Similarly, function “Outlet()” creates an open-ended link from a ChemPlugin instance, and “nOutlets()” reports how many such links exist.

A client can create any number of links between two instances. For example, you might wish to create a link carrying fluid from an instance "cp1" to another, "cp2". The client could then create a second link to account for the possibility of back-flow.

Each of the functions in this section are reciprocal in operation. In other words, if a client links "cp2" to "cp1", both instances know about the link. You should then link "cp1" to "cp2" only if you wish to spawn a second link between the instances. Similarly, when the client unlinks "cp2" from "cp1", both instances are aware the link has been removed.

B.2.3.1 Link()

Syntax:

```
FUNCTION Link(cp1, cp2) RESULT(link)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp1
    TYPE(ChemPlugin), INTENT(in), TARGET, OPTIONAL :: cp2
    TYPE(CpiLink), INTENT(out) :: link

\\ OR

FUNCTION Link(cp, index) RESULT(link)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER, INTENT(in) :: index
    TYPE(CpiLink), INTENT(out) :: link
```

The "Link()" member function connects two ChemPlugin instances. For example, the statements

```
TYPE(ChemPlugin) :: cp1, cp2
TYPE(CpiLink) :: newlink
newlink = cp1%Link(cp2)
```

link two ChemPlugin instances, "cp1" and "cp2".

Once this statement is executed, there is no need to execute the reciprocal operation

```
newlink = cp2%Link(cp1)
```

Doing so, in fact, would create an additional link.

The member function returns a reference to the link, which a client can store for later operations. For example, the statements

```
newlink = cp1%Link(cp2)
newlink%FlowRate(0.2, "m3/s")
```

set a flow rate of 0.2 m^3 s^{-1} from "cp2" to "cp1".

The conventional syntax for calling the “Link()” function is

```
err = Link(newlink, cp1, cp2)
```

In this case, references to the link to be created as well as the originating instance are set out in the argument list.

When a client calls “Link()” with no arguments,

```
newlink = cp1%Link()
```

the function creates an open link that functions as a free outlet. This syntax is an alternative to the “Outlet()” member function described in the next subsection.

When a client passes “Link()” an index, rather than a reference to a ChemPlugin instance, the member function returns a reference to the link in question. If, for example, we create two links

```
newlink1 = cp1%Link(cp2)
newlink2 = cp1%Link(cp3)
```

the links will have indices 0 and 1, respectively, assuming no earlier links exist. Then, the statement

```
TYPE(CpiLink) :: link_ref
link_ref = cp1%Link(1)
```

returns a reference to the second link, to be stored in “link_ref”.

B.2.3.2 Outlet()

Syntax:

```
FUNCTION Outlet(cp) RESULT(link)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    TYPE(CpiLink), INTENT(out) :: link
```

The “Outlet()” member function creates an open link that functions as a free outlet. A call to “Outlet()” is the same as calling “Link()” with no arguments, as discussed previously.

For example, the statement

```
TYPE(CpiLink) :: newlink
newlink = cp1%Outlet()
```

creates an open link to which “newlink” refers. The conventional syntax is

```
err = Outlet(newlink, cp1)
```

Note: Water can flow outward along the link, away from "cp", but not inward toward the instance; no first-order transport, such as diffusion or heat conduction, is possible across an open link.

B.2.3.3 Unlink()

Syntax:

```
 FUNCTION Unlink(link) RESULT(retval)
     CLASS(CpiLink), INTENT(in), TARGET :: link
     INTEGER :: retval

\\ OR

FUNCTION Unlink(cp1, cp2) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp1
    TYPE(ChemPlugin), INTENT(in), TARGET, OPTIONAL :: cp2
    INTEGER :: retval

\\ OR

FUNCTION Unlink(cp1, index) RESULT(retval)
     CLASS(ChemPlugin), INTENT(in), TARGET :: plugin0
     INTEGER, INTENT(in) :: index
     INTEGER :: retval
```

A client can remove a link between two nodes using the "Unlink()" member function on the CpiLink or ChemPlugin class. In the latter case, it can refer to the link either by index or reference to the linked instance.

Some examples:

```
TYPE(ChemPlugin) :: cp1, cp2
TYPE(CpiLink) :: link1
link1 = cp1%Link(cp2)
... some code ...
err = link1%Unlink()
    or
err = cp1%Unlink(cp2)
    or
err = cp1%Unlink(0)
```

Each of the three examples serves the same purpose. A zero-value return indicates success.

B.2.3.4 ClearLinks()

Syntax:

```
FUNCTION ClearLinks(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER :: retval
```

The “ClearLinks()” member function removes all links to a ChemPlugin instance. Example:

```
err = cp%ClearLinks()
```

A zero-value return indicates success.

B.2.3.5 nLinks()

Syntax:

```
FUNCTION nLinks(cp1, cp2) RESULT(num_Links)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp1
    TYPE(ChemPlugin), INTENT(in), TARGET, OPTIONAL :: cp2
    INTEGER(C_INT) :: num_Links
```

Member function “nLinks()” returns the number of links that have been made to a ChemPlugin instance.

When a client calls the function without an argument for “cp2“, it reports the total number of links to the instance, including open links. For example, the statements

```
link = cp1%Link(cp2)
link = cp1%Link(cp3)
num_links = cp1%nLinks()
```

result in a value of 2 being stored in variable “num_links”.

Calling the function with a second ChemPlugin instance as an argument yields the number of links to the second instance. Continuing the previous example, the statement

```
... cont'd ...
num_links = cp1%nLinks(cp3)
```

returns to “num_links” a value of one.

B.2.3.6 nOutlets()

Syntax:

```
FUNCTION nOutlets(cp) RESULT(num_outlets)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER(C_INT) :: num_outlets
```

Member function "nOutlets()" returns the number of open links connected to a ChemPlugin instance.

B.2.4 Transport across links

Member functions "FlowRate()", "Transmissivity()", and "HeatTrans()" control how chemical mass and heat energy are transported across links. In each case, calling the member function with a numerical value and optionally specifying units sets the quantity in question. Calling the function without a numerical value, in contrast, returns the current setting in the units specified, or in the default units.

B.2.4.1 FlowRate()

Syntax:

```
!! To set the flow rate
FUNCTION FlowRate(link, flow, units) RESULT(retval)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    REAL, INTENT(in), VALUE :: flow
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT) :: retval

\\ OR

!! To get the flow rate
FUNCTION FlowRate(link, units) RESULT(flow)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    REAL :: flow
```

Use member function "FlowRate()" to set the rate at which water flows across a link. The default unit for flow rate is $m^3\ s^{-1}$, but a client can specify any other unit of volume flow, as listed in the **Units Recognized** appendix.

Calling "FlowRate()" without a numeric argument returns the current value for flow rate, as set by the most recent call to the function. When a link is created, the flow rate is guaranteed to be zero-value initially.

By ChemPlugin convention, flow into an instance is positive, and outward flow is negative in sign. Hence, the statements

```
TYPE(CpiLink) :: link1
Real :: flow
link1 = cp1%Link(cp2)
err = link1%FlowRate(2.0d6, "cm3/s")
flow = link1%FlowRate()
```

result in a value of 2.0 being stored in variable “flow”, since this quantity is 2×10^6 cm^3 s^{-1}, expressed in m^3 s^{-1}. In this case, flow is positive, so water flows from “cp2” to “cp1”.

B.2.4.2 Transmissivity()

Syntax:

```
!! To set the mass transmissivity
FUNCTION Transmissivity(link, trans, units) RESULT(retval)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    REAL, INTENT(in), VALUE :: trans
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT) :: retval

\\ OR

!! To get mass transmissivity
FUNCTION Transmissivity(link, units) RESULT(trans)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    REAL :: trans
```

The “Transmissivity()” member function sets the transmissivity describing mass transport across a link by first-order processes, such as diffusion, dispersion, and turbulent mixing. The definition of the mass transmissivity is described in this User’s Guide, in the **Flow and Transport** chapter.

A client sets a value in units of m^3 s^{-1} by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial transmissivity is guaranteed to be zero-value.

B.2.4.3 HeatTrans()

Syntax:

```
!! To set the heat transmissivity
FUNCTION HeatTrans(link, trans, units) RESULT(retval)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    REAL, INTENT(in), VALUE :: trans
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT) :: retval

\\ OR

!! To get mass transmissivity
FUNCTION HeatTrans(link, units) RESULT(trans)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    REAL :: trans
```

The "HeatTrans()" member function sets the thermal transmissivity describing the transport of heat energy across a link by first-order processes, such as conduction, thermal dispersion, and turbulent mixing. Thermal transmissivity is described in the **Flow and Transport** chapter in this User's Guide.

A client sets a value in units of J K^{-1} s^{-1} by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial thermal transmissivity is guaranteed to be zero-value.

B.2.5 Time marching loop

Member functions "ReportTimeStep()", "AdvanceTimeStep()", "AdvanceTransport()", "AdvanceHeatTransport()", and "AdvanceChemical()" work together to make up a time marching loop. The function "ExtendRun()" can be used to prolong a time marching loop to a new end time, once the initial loop is complete.

B.2.5.1 ReportTimeStep()

Syntax:

```
FUNCTION ReportTimeStep(cp, units) RESULT(time_step)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    REAL :: time_step
```

Member function "ReportTimeStep()" returns the largest time step an instance may take if it is to maintain numerical stability and honor any constraints the client program may have prescribed. The limiting time step is by default returned in seconds, but the optional argument allows a client to specify an alternative unit of time.

A client program, upon beginning a pass through a time marching loop, will in general query each ChemPlugin instance the program has spawned. The client program should then take the least of the values returned and use that value as the length Δt of the current time step.

B.2.5.2 AdvanceTimeStep()

Syntax:

```
FUNCTION AdvanceTimeStep(cp, time_step, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    REAL, INTENT(in),VALUE :: time_step
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER :: retval
```

Member function “AdvanceTimeStep()” moves the current time level carried in an instance forward by the time step specified, adds or removes simple reactants, and adjusts sliding reactants. By default, the time step is provided in seconds, but the optional second argument lets a client set an alternative unit of time. A return value of 0 indicates success and non-zero value indicates either an error as occured or client has reached the final simulation time.

B.2.5.3 AdvanceTransport()

Syntax:

```
FUNCTION AdvanceTransport(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER :: retval
```

Member function “AdvanceTransport()” triggers the instance to evaluate the effect of mass transport on the instance’s chemical composition over the course of the current time step. A non-zero return value indicates failure to perform this step.

B.2.5.4 AdvanceHeatTransport()

Syntax:

```
FUNCTION AdvanceHeatTransport(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER :: retval
```

Member function “AdvanceHeatTransport()” triggers the instance to evaluate the effect of heat transport on the instance’s temperature over the course of the current time step. A non-zero return value indicates failure to perform this step.

B.2.5.5 AdvanceChemical()

Syntax:

```
FUNCTION AdvanceChemical(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER :: retval
```

Member function "AdvanceChemical()" causes the instance to evaluate the effect of kinetic and equilibrium reactions on an instance's chemical state over the course of the current time step. A non-zero return value indicates failure to perform this step.

B.2.5.6 SlideFugacity()

Syntax:

```
FUNCTION SlideFugacity(cp, gas_name, value) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN=*), INTENT(in) :: gas_name
    REAL, INTENT(in) :: value
    INTEGER :: retval
```

Use member function "SlideFugacity()" to adjust the fugacity of a fixed-fugacity gas within an instance, once the instance has been initialized. The "gas_name" is the name of the gas in question (e.g., "CO2(g)") and "value" is the revised fugacity of that gas.

In order to use the "SlideFugacity()" function, the fugacity of the gas in question must be fixed in the instance configuration. For example:

```
cp%Config("fix fugacity CO2(g)")
```

Note the function, despite its name, can be used also to adjust the fixed activity of an aqueous species, or an activity ratio that has been fixed.

B.2.5.7 SlideTemperature()

Syntax:

```
FUNCTION SlideTemperature(cp, temp, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    REAL, INTENT(in) :: temp
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval
```

Use member function "SlideTemperature()" to adjust an instance's temperature, once it has been initializied. The "unit" field is optional and defaults to "C".

Do not use “SlideTemperature()” to set an instance’s initial temperature; instead, call

```
cp%Config("temperature = 45 C")
```

to configure the instance at the desired temperature, before initializing it.

B.2.5.8 ExtendRun()

Syntax:

```
FUNCTION ExtendRun(cp add_time, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    REAL, INTENT(in) :: add_time
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval
```

Use member function “ExtendRun()” to extend a reaction simulation, once time marching is complete. The first argument is the amount of time to be added to the simulation, in the unit specified as the second argument, or in seconds, by default. The function returns a zero value upon successful extension of the run.

By extending a simulation’s time range, you extend the range in reaction progress to be traversed. Tracing a reaction path that spans 10 years, for example, carries the reaction progress variable ξ from zero to one. If you were to then add 20 years to the simulation, ξ would, over the course of the second round of time marching, increase from one to three.

B.2.6 Retrieving results

A client program uses the “Report()”, “Report1()”, “Report1i()”, and “Report1c()” member functions to gather information about the current state of a ChemPlugin instance.

B.2.6.1 Report()

Syntax:

```
!! To get data which is an array of strings
FUNCTION Report(cp, target, keywords, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval

\\ OR

!! To get an array of numeric data
FUNCTION Report(cp, target, keywords, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    REAL(8), INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
```

```
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval

\\ OR

!! To get an array of integers
FUNCTION Report(cp, target, keywords, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER, INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval
```

Use member function "Report()" to retrieve calculation results from a ChemPlugin instance. Here, "target" is the address in memory to which the results are to be written. The "keywords" argument identifies the specific result or results to be written, and the optional "unit" argument sets the unit in which the results are to be cast. The function returns the number of values copied to "target".

Whenever the function is unable to fulfill a request, such as when an impossible unit conversion has been requested, it fills the corresponding location or locations in "target" with parameter "ANULL", defined in "ChemPlugin.h".

The options for specifying "keywords" are listed in the **Report Function** appendix to this User's Guide, and the available units are shown in the **Units Recognized** appendix.

If a client passes no argument as the "target" argument, i.e using the following version of "Report"

```
FUNCTION Report(cp, keywords, units) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER :: retval
```

the function returns the number of values in the data for the specified keywords.

"Report()" is used to retrieve an array of data, such as a vector of the concentrations of a group of aqueous species. To retrieve a single value of type double, such as the pH or a species' concentration, a client may instead call the shorter "Report1()" function. For a single integer or character string, use "Report1i()" or "Report1c()", respectively.

Please refer to the **Retrieving Results** chapter of this User's Guide for detailed information about the "Report()" and "Report1()" functions, including several examples of their use.

B.2.6.2 Report1(), Report1i(), and Report1c()

Syntax:

```
!! To get a single real numeric data value
FUNCTION Report1(cp, value, units) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    REAL(8) :: data

!! To get a single integer data value
FUNCTION Report1i(cp, value, units) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER :: data

!! To get a string value
FUNCTION Report1c(cp, value, units) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: value
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    CHARACTER(LEN = 255) :: data
```

Use member function “Report1()” to retrieve from a ChemPlugin instance a single value of type real, such as the pH or the concentration of a specific aqueous species. The “keywords” argument identifies the specific result or results to be written, and the optional “unit” argument sets the unit in which the results are to be cast. Similarly, “Report1i()” and “Report1c()” retrieve an integer value and a string, respectively.

The options for specifying “keywords” are listed in the **Report Function** appendix to this User’s Guide, and the available units are shown in the **Units Recognized** appendix. The function returns the retrieved value.

When “Report1()” fails, it returns a value of “ANULL”, defined in “ChemPlugin.h”; “Report1i()” returns “ANULL” cast as an integer, and “Report1c()” returns “ANULL” as a character string.

Please refer to the **Retrieving Results** chapter of this User’s Guide for detailed information about the “Report()” and “Report1()” functions, including several examples of their use.

B.2.7 Output streams

A client program controls the output streams from a ChemPlugin instance with member functions “Console()”, “PrintOutput()”, “PlotHeader()”, “PlotBlock()”, and “PlotTrailer()”. The first function is described in the **Overview** chapter of this User’s Guide, and use of the latter four functions is explained in detail in the **Direct Output** chapter. For FORTRAN syntax of these functions, refer to the next sub-sections.

It is important to avoid allowing more than one ChemPlugin instance to write into the same dataset. In such a situation, the output from the instances would appear intermingled and probably unintelligible.

B.2.7.1 Console()

Syntax:

```
FUNCTION Console(cp, target) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    CHARACTER(LEN=*), INTENT(in) :: traget
    INTEGER :: retval
```

Member function "Console()" controls where an instance's console output is directed. Console output consists of routine messages an instance produces as it initializes and undetakes calculations, as well as any warning and error messages that may be generated. By default, a ChemPlugin instance does not produce console output.

The function's optional argument "target" is the target for the console stream, which may be "stdout", "stderr", or the name of a dataset. When a client calls the function without an argument, or an empty string as the argument, output to the console stream is disabled. A client may enable, disable, or redirect an instance's console output at any time.

A client can direct an instance's console output at declaration:

```
cp = ChemPlugin("stdout")
```

Since an instance produces no console output until directed to do so, this is the only way to capture messages produced when an instance comes into scope and initializes.

B.2.7.2 PrintOutput()

Syntax:

```
FUNCTION PrintOutput1(cp, basename) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: basename
    INTEGER :: retval

\\ OR

FUNCTION PrintOutput1(cp, basename, label, rewnd) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    CHARACTER(LEN=*), INTENT(in) :: basename
    CHARACTER(LEN=*), INTENT(in) :: label
    LOGICAL, INTENT(in), OPTIONAL :: rewnd
    INTEGER :: retval
```

Calling member function “PrintOutput()” triggers a ChemPlugin instance to append a data block to a print-format dataset.

The optional argument sets the file’s base name. The full file name is the base name combined with any suffix that may be set. For example, the calls

```
cp%Config("suffix _1")
cp%PrintOutput("myPrint.txt")
```

cause a block of output to be written to dataset “myPrint_1.txt”.

When a client calls “PrintOutput()” without basename(“PrintOutput(cp)”), the instance appends a data block to whatever file is open for print-format output. If none is open, the instance writes to “ChemPlugin_output.txt”.

The optional “label” argument allows the program to pass an optional character string to be written into the print dataset, at the head of the data block. For example,

```
cp%PrintOutput("", "Initial condition")
```

would write the string “Initial condition” and then a data block to the currently open dataset.

Passing a true (non-zero) value as the optional argument “rewnd” causes the instance to rewind the print-format dataset and write a data block at the head of the file. Use

```
cp%PrintOutput("", "", true)
```

to write a data block at the head of the currently open dataset.

B.2.7.3 PlotHeader()

Syntax:

```
FUNCTION PlotHeader(cp, basename) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: basename
    INTEGER :: retval
```

Member function “PlotHeader()” opens and initializes a plot-format dataset by writing header information to it.

A client may specify the file’s base name as a character string; the full name is the base name combined with the suffix, if one has been set. If a client calls the function without an argument, it writes to any file that may be open for plot output. If none is open, the function opens “ChemPlugin_plot.gtp”, accounting for a suffix, if set. The “.gtp” extension identifies the file as **Gtplot** input.

B.2.7.4 PlotBlock()

Syntax:

```
FUNCTION PlotBlock(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER(C_INT) :: retval
```

Member function "PlotBlock()" appends to the currently open plot-format dataset a block of data representing an instance's current state. The dataset must have been initialized with a call to "PlotHeader()".

B.2.7.5 PlotTrailer()

Syntax:

```
FUNCTION PlotTrailer(cp) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER(C_INT) :: retval
```

Member function "PlotTrailer()" completes the plot-format output dataset by writing the trailing data structure. A trailer is required by program **Gtplot** before it can read the dataset.

When functions "ReportTimeStep()" and "AdvanceTimeStep()" detect time marching is complete, they automatically write a trailer to any open plot dataset, and close the dataset. It is not necessary to write a trailer, then, at the end of a time marching loop.

Significantly, a client may continue to append data blocks to a plot dataset, even after it has used "PlotTrailer()" to write a trailer to it. The additional data blocks overwrite the trailer data, so the client needs to call "PlotTrailer()" once again to close the dataset.

B.2.8 Convenience

B.2.8.1 Version()

Syntax:

```
FUNCTION Version(cp)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN=255) :: Version
```

Member function "Version()" returns a pointer to a character string identifying the version of ChemPlugin in use.

B.2.8.2 ConvertUnit()

Syntax:

```
FUNCTION ConvertUnit1(cp, value, old_unit, new_unit) RESULT(new_value)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    REAL, INTENT(in), VALUE :: value
    CHARACTER(LEN = *), INTENT(in) :: old_unit, new_unit
    REAL(C_DOUBLE) :: new_value

FUNCTION ConvertUnit1(cp, value, old_unit, new_unit, mw, mv, z) RESULT(new_value)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    REAL, INTENT(in), VALUE :: value, mw, mv, z
    CHARACTER(LEN = *), INTENT(in) :: old_unit, new_unit
    REAL(C_DOUBLE) :: new_value

FUNCTION ConvertUnit1(cp, value, old_unit, new_unit, mw, mv, z, &
                      wmass, smass, density, vbulk) RESULT(new_value)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    REAL, INTENT(in), VALUE :: value, mw, mv, z, &
                               wmass, smass, density, vbulk
    CHARACTER(LEN = *), INTENT(in) :: old_unit, new_unit
    REAL(C_DOUBLE) :: new_value
```

Member function “ConvertUnit” converts values from one unit to another. The **Units Recognized** appendix to this User’s Guide lists the units available for conversion.

There are three variants to the function. In the simplest variant, a client passes only the value to be converted, along with the old and new units, and the function returns the converted value.

Unit conversions involving mass and concentration may require additional information. To convert concentration from mmol/kg to mg/kg, for example, the function needs to know molecular weight. For other conversions, the function may require the molar volume, electrical charge on the species (to convert to and from meq/kg, for example), the mass of solvent water, the solution mass, the fluid density, and the bulk volume of the system.

In the second variant of the function call, a client also specifies the mole weight “mw” in g mol^{-1}, the mole volume “mv” in cm^3 mol^{-1}, and the ion charge “z” on the species in question. The function takes water mass (kg), solution mass (kg), density (g cm^{-3}), and bulk volume (cm^3) from the current state of the ChemPlugin instance.

In the final form, the client specifies the latter four variables, in the units listed. This variant of the function call can be helpful because it can be used before a client has calculated the initial system, by calling “Initialize()”.

B.2.9 Cluster computing

The MPI version of ChemPlugin includes a number of Fortran member functions specific to cluster computing.

B.2.9.1 MpiAssign()

Syntax:

```
FUNCTION MpiAssign(plugin, f_dst_mpi_rank) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    INTEGER, INTENT(in) :: f_dst_mpi_rank
    INTEGER(C_INT) :: retval, dst_mpi_rank
```

Use the "MpiAssign()" member function to associate the stub of a ChemPlugin instance with a specific client copy. If argument "rank" is the same as the rank of the client copy calling the function, the stub is expanded within that copy into a full ChemPlugin instance, and that copy becomes responsible for calculations involving the instance.

For example,

```
cp%MpiAssign(0)
```

assigns instance "cp" to the first client copy, which has a rank of zero. Afterward, "cp" will reference a full ChemPlugin instance on the first client copy, whereas on the remaining client copies, it will refer to a stub of that instance.

As an alternative to calling MpiAssign(), you can assign rank to a ChemPlugin object using the ChemPlugin constructor, by setting the rank as the optional third argument; see the **Cluster Computing** chapter. Behavior in this case is the same as if the constructor and "MpiAssign()" had been called serially.

B.2.9.2 MpiOnRank()

Syntax:

```
FUNCTION MpiOnRank(plugin) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    INTEGER(C_INT) :: retval
```

The "MpiOnRank()" member function returns one when called from a client copy of the same rank as the instance, and zero otherwise.

For example, if instance "cp" is assigned to the first client copy

```
is_local = cp%MpiOnRank()
```

the value of INTEGER variable "is_local" on that copy will be one, but zero on the remaining copies.

B.2.9.3 MpiRank()

Syntax:

```
FUNCTION MpiRank(plugin) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: plugin
    INTEGER(C_INT) :: retval
```

Member function “MpiRank()” returns the rank of the client copy to which an instance is assigned.

For example, if instance “cp” is assigned to the first client copy, executing the statement

```
rank = cp%MpiRank()
```

will leave INTEGER variable “rank” with a value of zero.

B.2.9.4 MpiReport()

Syntax:

```
!! To get data which is an array of strings
FUNCTION MpiReport(cp, target, keywords, units, collector) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    INTEGER :: retval

\\ OR

!! To get an array of numeric data
FUNCTION MpiReport(cp, target, keywords, units, collector) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    REAL(8), INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    INTEGER :: retval

\\ OR

!! to get an array of integers
FUNCTION MpiReport(cp, target, keywords, units, collector) RESULT(retval)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    INTEGER, INTENT(out) :: target(:)
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN=*), INTENT(in), OPTIONAL :: units
```

```
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    INTEGER :: retval
```

A call to "MpiReport()" with two or three parameters behaves identically to calling "Report()", except that it can retrieve information from any ChemPlugin instance, rather than only those local to the client copy making the call. You use this form of the function call when every client copy needs to retrieve information about all of the ChemPlugin instances considered by the communication group.

When you call "MpiReport()" for an instance "cp", the client copy responsible for "cp" broadcasts information to the other client copies, which wait to receive the data. As such, the client copies need to call the function synchronously, i.e., in the same sequence. The function may not be called within a multithreaded loop.

The optional fourth parameter is used to specify the rank of a client copy that will collect the result. You use this form of the function call when you would like a single client copy to assemble information from all of the ChemPlugin instances considered by the communication group.

The function when used in this way does work on only the collector copy and the client copy responsible for "cp". Called from those copies, the function writes the requested information to the target and returns the number of values written. Of course, the responsible copy may be the same as the collector copy. For the remaining client copies, the function writes no data and returns a value of zero; since nothing is written, no memory need be allocated for "target".

Setting the collector to "GLOBAL" is equivalent to omitting the fourth argument.

B.2.9.5 MpiReport1(), MpiReport1i(), MpiReport1c()

Syntax:

```
!! To get a single real numeric data value
FUNCTION MpiReport1(cp, value, units, collector) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    REAL(8) :: data

!! To get a single integer data value
FUNCTION MpiReport1i(cp, value, units, collector) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
    CHARACTER(LEN = *), INTENT(in) :: keywords
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    INTEGER :: data

!! To get a string value
FUNCTION MpiReport1c(cp, value, units, collector) RESULT(data)
    CLASS(ChemPlugin), INTENT(in), TARGET :: cp
```

```
    CHARACTER(LEN = *), INTENT(in) :: value
    CHARACTER(LEN = *), INTENT(in), OPTIONAL :: units
    INTEGER(C_INT), INTENT(in), OPTIONAL :: collector
    CHARACTER(LEN = 255) :: data
```

Functions “MpiReport1()”, “MpiReport1i()”, and “MpiReport1c()” are the counterparts to “MpiReport()” for returning, respectively, a single floating point or integer value, or a character string.

Called with one or two arguments, the functions behave identically to “Report1()”, “Report1i()”, and “Report1c()”, except they can retrieve information from any ChemPlugin instance, rather than only those local to the client copy making the call. When you set a collector copy, the functions let you assemble information from any ChemPlugin instance within a single client copy, just as “MpiReport()” does. Setting the collector to “GLOBAL” is equivalent to omitting the third argument.

The “MpiReport()” family functions are synchronous and hence need to be called by each client copy in the same sequence; they may not be called within a multithreaded loop.

B.2.9.6 MpiUpdateLink()

Syntax:

```
FUNCTION MpiUpdateLink(link, f_update_values) RESULT(retval)
    CLASS(CpiLink), INTENT(in), TARGET :: link
    INTEGER, INTENT(in), OPTIONAL :: f_update_values
```

In a cluster computing application, a local ChemPlugin instance may be linked to an instance assigned to a different client copy. In this case, the corresponding CpiLink object caches information about the foreign instance, making it available to the local copy.

Whenever the state of the foreign instance has changed in an MPI application, its cached information needs to be updated before you call AdvanceTransport() or AdvanceHeatTransport(). The “MpiUpdateLink()” member function of the CpiLink object serves to update the cache.

“MpiUpdateLink()” needs to be called by every client copy in the communication group, and in the same sequence by each copy. The function may not be called from within a multithreaded loop.

A zero-value return from the function indicates success. Note that all client copies need to call this function in the same sequence, to avoid the possibility of entering a gridlocked state.

In specialized circumstances you may wish use the “task” argument to control which parts of the cache are updated. The argument may be

- “-1” updates all of the below, the default
- “1” updates temperature and thermal properties

- "2" updates fluid density and viscosity
- "4" updates chemical and isotopic composition
- "8" updates the position and extent of a linked instance

Options can be combined by integer addition, e.g., "1 + 2".
An example:

```
TYPE(ChemPlugin) :: cp1, cp2
TYPE(CpiLink) :: link0
INTEGER :: l, error

cp1 = ChemPlugin("", "", 0)
cp2 = ChemPlugin("", "", 1)
link0 = cp1%Link(cp2)

... some code ...

DO l = 0, cp1%nLinks() - 1
    MpiUpdateLink( cp1%Link(l) )
END IF
```

B.3 Python

Create an instance called “cp” of Chemplugin using the object’s constructor

```
cp = ChemPlugin()
```

The constructor can accept two arguments, as described in the **Overview** chapter. The first argument defines the console output stream, whereas the second is used to set option flags. For example, the statement

```
cp = ChemPlugin("stdout", "-d mythermo.tdat -s mysurface.sdat")
```

creates an instance that sends its console output to the standard output, takes thermodynamic data from “mythermo.tdat”, and reads surface data from “mysurface.sdat”.

B.3.1 Configuring and initializing instances

Member functions “Config()” and “Initialize()” let a client configure a ChemPlugin instance and prepare it for the start of a reaction simulation.

B.3.1.1 Config()

Syntax:

```
def Config(self, command):
```

Use member function “Config()” to pass configuration commands to a ChemPlugin instance. The configuration commands and their syntax are listed in the **Configuration Commands** chapter of this guide. Examples:

```
cp.Config("pH=5")
cp.Config("Na+ = 1 mmol/kg")
```

A client can pass multiple commands with a single function call, or continue a command onto another call

```
cp.Config("Na+ = 1 mmol/kg; Cl- = 1 mmol/kg")

cp.Config("kinetic Quartz \")
cp.Config("rate_con = 2e-12, surface = 1000")
```

if it separates commands with semicolons, and marks incomplete commands with a trailing backslash.

A zero-value return indicates the command processed successfully.

B.3.1.2 Initialize()

Syntax:

```
def Initialize(self, time_end=0.0, unit=None):
```

The "Initialize()" member function triggers an instance to calculate its initial condition, including the distribution of mass across the various chemical species, and to prepare the instance to begin time marching.

The member function is commonly called without arguments, but the client can use the call to specify the end time of the simulation. For example, the statement

```
cp.Initialize(10.0, "years")
```

has the same effect as

```
cp.Config("time end = 10 years")
cp.Initialize()
```

An end time of zero is ignored, and the default unit for time is seconds. A non-zero return indicates the instance was unable to initialize. In this case, diagnostic messages are written to the instance's console output.

B.3.2 Linking instances

Member function "Link()" creates a link connecting two ChemPlugin instances, functions "Unlink()" and "ClearLinks()" remove links that have been created, and the function "nLinks()" reports the number of existing links. Similarly, function "Outlet()" creates an open-ended link from a ChemPlugin instance, and "nOutlets()" reports how many such links exist.

A client can create any number of links between two instances. For example, you might wish to create a link carrying fluid from an instance "cp1" to another, "cp2". The client could then create a second link to account for the possibility of back-flow.

Each of the functions in this section are reciprocal in operation. In other words, if a client links "cp2" to "cp1", both instances know about the link. You should then link "cp1" to "cp2" only if you wish to spawn a second link between the instances. Similarly, when the client unlinks "cp2" from "cp1", both instances are aware the link has been removed.

B.3.2.1 Link()

Syntax:

```
// arg can be either another plugin or an index.
def Link(self, arg=None)
```

The "Link()" member function connects two ChemPlugin instances. For example, the statements

```
cp1 = ChemPlugin()
cp2 = ChemPlugin()
cp1.Link(cp2)
```

link two ChemPlugin instances, “cp1” and “cp2”. Once this statement is executed, there is no need to execute the reciprocal operation

```
cp2.Link(cp1)
```

Doing so, in fact, would create an additional link.

The member function returns a reference to the link, which a client can store for later operations. For example, the statements

```
link1 = cp1.Link(cp2)
link1.FlowRate(0.2, "m3/s")
```

set a flow rate of 0.2 m^3 s^{-1} from “cp2” to “cp1”.

When a client calls “Link()” without an argument, the function creates an open link that functions as a free outlet. Water can flow outward along the link, away from “cp”, but not inward toward the instance; no first-order transport, such as diffusion or heat conduction, is possible across an open link.

When a client passes “Link()” an index, rather than a reference to a ChemPlugin instance, the member function returns a reference to the link in question. If, for example, we create two links

```
cp1.Link(cp2)
cp1.Link(cp3)
```

the links will have indices 0 and 1, respectively, assuming no earlier links exist. Then, the statement

```
link1 = cp1.Link(1)
```

returns a reference to the second link, to be stored in “link1”.

B.3.2.2 Outlet()

Syntax:

```
def Outlet(self)
```

The “Outlet()” member function creates an open link that functions as a free outlet. A call to “Outlet()” is the same as calling “Link()” without an argument.

For example, the statement

```
link1 = cp.Outlet();
```

creates an open link to which "link1" refers.

Note: Water can flow outward along the link, away from "cp", but not inward toward the instance; no first-order transport, such as diffusion or heat conduction, is possible across an open link.

B.3.2.3 Unlink()

Syntax:

```
// member of CpiLink class
 def Unlink(self):

// member of ChemPlugin class
 // arg can be either a plugin or an index,
 // arg=None is for deleting free outlet links
 def cp.Unlink(self, arg=None):
```

A client can remove a link between two nodes using the "Unlink()" member function of either the CpiLink or ChemPlugin class. In the latter case, it can refer to the link either by index or reference to the linked instance.

Some examples:

```
cp1 = ChemPlugin()
cp2 = ChemPlugin()
link1 = cp1.Link(cp2)
... some code ...
link1.Unlink()
    or
cp1.Unlink(cp2)
    or
cp1.Unlink(0)
```

Each of the three examples serves the same purpose. A zero-value return indicates success.

B.3.2.4 ClearLinks()

Syntax:

```
def ClearLinks(self)
```

The "ClearLinks()" member function removes all links to a ChemPlugin instance. Example:

```
cp.ClearLinks()
```

A zero-value return indicates success.

B.3.2.5 nLinks()
Syntax:

```
def nLinks(self, another_cp=None):
```

Member function “nLinks()” returns the number of links that have been made to a ChemPlugin instance.

When a client calls the function without an argument, it reports the total number of links to the instance, including open links. For example, the statements

```
cp1.Link(cp2)
cp1.Link(cp3)
m = cp1.nLinks()
```

result in a value of 2 being stored in variable “m”.

Calling the function with a second ChemPlugin instance as an argument yields the number of links to the second instance. Continuing the previous example, the statement

```
... cont'd ...
n = cp1.nLinks(cp3)
```

returns to “n” a value of one.

B.3.2.6 nOutlets()
Syntax:

```
def nOutlets(self):
```

Member function “nOutlets()” returns the number of open links connected to a ChemPlugin instance.

B.3.3 Transport across links

Member functions “FlowRate()”, “Transmissivity()”, and “HeatTrans()” control how chemical mass and heat energy are transported across links. In each case, calling the member function with a numerical value and optionally specifying units sets the quantity in question. Calling the function without a numerical value, in contrast, returns the current setting in the units specified, or in the default units.

B.3.3.1 FlowRate()

Syntax:

```
// argv is an array. It can be either argv=[flow] or
// argv=[flow, unit]
def FlowRate(self, *argv):
```

Use member function "FlowRate()" to set the rate at which water flows across a link. The default unit for flow rate is $m^3\ s^{-1}$, but a client can specify any other unit of volume flow, as listed in the **Units Recognized** appendix.

Calling "FlowRate()" without a numeric argument returns the current value for flow rate, as set by the most recent call to the function. When a link is created, the flow rate is guaranteed to be zero-value initially.

By ChemPlugin convention, flow into an instance is positive, and outward flow is negative in sign. Hence, the statements

```
link1 = cp1.Link(cp2)
link1.FlowRate(2.0e6, "cm3/s")
flow = link1.FlowRate()
```

result in a value of 2.0 being stored in variable "flow", since this quantity is 2×10^6 $cm^3\ s^{-1}$, expressed in $m^3\ s^{-1}$. In this case, flow is positive, so water flows from "cp2" to "cp1".

B.3.3.2 Transmissivity()

Syntax:

```
// argv is an array. It can be either argv=[trans] or
// argv=[trans, unit]
def Transmissivity(self, *argv):
```

The "Transmissivity()" member function sets the transmissivity describing mass transport across a link by first-order processes, such as diffusion, dispersion, and turbulent mixing. The definition of the mass transmissivity is described in this User's Guide, in the **Flow and Transport** chapter.

A client sets a value in units of $m^3\ s^{-1}$ by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial transmissivity is guaranteed to be zero-value.

B.3.3.3 HeatTrans()

Syntax:

```
// argv is an array. It can be either argv=[trans] or
// argv=[trans, unit]
def HeatTrans(self, *argv):
```

The "HeatTrans()" member function sets the thermal transmissivity describing the transport of heat energy across a link by first-order processes, such as conduction, thermal dispersion, and turbulent mixing. Thermal transmissivity is described in the **Flow and Transport** chapter in this User's Guide.

A client sets a value in units of J K^{-1} s^{-1} by default, but it can specify other units, as described in the **Units Recognized** appendix. When a client calls the function without a numeric argument, the function returns the currently set value. Upon creating a link, the initial thermal transmissivity is guaranteed to be zero-value.

B.3.4 Time marching loop

Member functions "ReportTimeStep()", "AdvanceTimeStep()", "AdvanceTransport()", "AdvanceHeatTransport()", and "AdvanceChemical()" work together to make up a time marching loop. The function "ExtendRun()" can be used to prolong a time marching loop to a new end time, once the initial loop is complete.

B.3.4.1 ReportTimeStep()

Syntax:

```
def ReportTimeStep(self, unit = None):
```

Member function "ReportTimeStep()" returns the largest time step an instance may take if it is to maintain numerical stability and honor any constraints the client program may have prescribed. The limiting time step is by default returned in seconds, but the optional argument allows a client to specify an alternative unit of time.

A client program, upon beginning a pass through a time marching loop, will in general query each ChemPlugin instance the program has spawned. The client program should then take the least of the values returned and use that value as the length Δt of the current time step.

B.3.4.2 AdvanceTimeStep()

Syntax:

```
def AdvanceTimeStep(self, deltat, unit=None):
```

Member function "AdvanceTimeStep()" moves the current time level carried in an instance forward by the time step specified, adds or removes simple reactants, and adjusts sliding reactants. By default, the time step is provided in seconds, but the optional second argument lets a client set an alternative unit of time. A zero-return

value indicates the process was successful. A non-zero value indicates the time marching has either completed or failed.

B.3.4.3 AdvanceTransport()

Syntax:

```
def AdvanceTransport(self):
```

Member function "AdvanceTransport()" triggers the instance to evaluate the effect of mass transport on the instance's chemical composition over the course of the current time step.

B.3.4.4 AdvanceHeatTransport()

Syntax:

```
def AdvanceHeatTransport(self):
```

Member function "AdvanceHeatTransport()" triggers the instance to evaluate the effect of heat transport on the instance's temperature over the course of the current time step.

B.3.4.5 AdvanceChemical()

Syntax:

```
def AdvanceChemical(self):
```

Member function "AdvanceChemical()" causes the instance to evaluate the effect of kinetic and equilibrium reactions on an instance's chemical state over the course of the current time step.

B.3.4.6 SlideFugacity()

Syntax:

```
def SlideFugacity(self, gas_name, value):
```

Use member function "SlideFugacity()" to adjust the fugacity of a fixed-fugacity gas within an instance, once the instance has been initialized. The "gas_name" is the name of the gas in question (e.g., "CO2(g)") and "value" is the revised fugacity of that gas.

In order to use the "SlideFugacity()" function, the fugacity of the gas in question must be fixed in the instance configuration. For example:

```
cpi.Config("fix fugacity CO2(g)")
```

Note the function, despite its name, can be used also to adjust the fixed activity of an aqueous species, or an activity ratio that has been fixed.

B.3.4.7 SlideTemperature()

Syntax:

```
def SlideTemperature(self, temp, unit=None):
```

Use member function “SlideTemperature()” to adjust an instance’s temperature, once it has been initializied. The “unit” field is optional and defaults to “C”.

Do not use “SlideTemperature()” to set an instance’s initial temperature; instead, call

```
cp.Config("temperature = 45 C")
```

to configure the instance at the desired temperature, before initializing it.

B.3.4.8 ExtendRun()

Syntax:

```
def ExtendRun(self, add_time, unit=None):
```

Use member function “ExtendRun()” to extend a reaction simulation, once time marching is complete. The first argument is the amount of time to be added to the simulation, in the unit specified as the second argument, or in seconds, by default. The function returns a zero value upon successful extension of the run.

By extending a simulation’s time range, you extend the range in reaction progress to be traversed. Tracing a reaction path that spans 10 years, for example, carries the reaction progress variable ξ from zero to one. If you were to then add 20 years to the simulation, ξ would, over the course of the second round of time marching, increase from one to three.

B.3.5 Retrieving results

A client program uses the “Report()” and “Report1()” member functions to gather information about the current state of a ChemPlugin instance.

B.3.5.1 Report()

Syntax:

```
def Report(self, keywords, units=None):
```

Use member function “Report()” to retrieve calculation results from a ChemPlugin instance. The “keywords” argument identifies the specific result or results to be written, and the optional “unit” argument sets the unit in which the results are to be cast. The function returns a list of the requested data. If the requested keyword evaluates to single value, the data is returned as a single-entry list.

Whenever the function is unable to fulfill a request, such as when an impossible unit conversion has been requested, it fills the corresponding location or locations in “target” with parameter “ANULL”, defined in “ChemPlugin.py”.

The options for specifying "keywords" are listed in the **Report Function** appendix to this User's Guide, and the available units are shown in the **Units Recognized** appendix.

"Report()" is used to retrieve a list of data, such as a vector of the concentrations of a group of aqueous species. To retrieve a single value, such as the pH or a species' concentration, a client may instead call the shorter "Report1()" function. "Report1()" can return a floating point or integer value, or a character string; use of the "Report1i()" and "Report1c()" functions in Python is deprecated.

Please refer to the **Retrieving Results** chapter of this User's Guide for detailed information about the "Report()" and "Report1()" functions, including several examples of their use.

B.3.5.2 Report1()

Syntax:

```
def Report1(self, value, unit=None):
```

Use member function "Report1()" to retrieve from a ChemPlugin instance a single value, such as the pH or the concentration of a specific aqueous species. The value returned can be a floating point number, an integer, or a character string. The "keywords" argument identifies the specific result or results to be written, and the optional "unit" argument sets the unit in which the results are to be cast.

The options for specifying "keywords" are listed in the **Report Function** appendix to this User's Guide, and the available units are shown in the **Units Recognized** appendix. The function returns the retrieved value.

If "Report1()" fails when queried for a floating point value, it returns a value of "ANULL", defined in "ChemPlugin.py"; the function returns "ANULL" cast as an integer if it fails when queried for an integer value, and a NULL pointer in lieu of an unavailable character string.

Use of the "Report1i()" and "Report1c()" functions in Python is deprecated, in favor of the "Report1()" member function.

Please refer to the **Retrieving Results** chapter of this User's Guide for detailed information about the "Report()" and "Report1()" functions, including several examples of their use.

B.3.6 Output streams

A client program controls the output streams from a ChemPlugin instance with member functions "Console()", "PrintOutput()", "PlotHeader()", "PlotBlock()", and "PlotTrailer()". The first function is described in the **Overview** chapter of this User's Guide, and use of the latter four functions is explained in detail in the **Direct Output** chapter. For Python syntax of these functions, refer to following subsections.

It is important to avoid allowing more than one ChemPlugin instance to write into the same dataset. In such a situation, the output from the instances would appear intermingled and probably unintelligible.

B.3.6.1 Console()

Syntax:

```
public int cp.Console()
def cp.Console(self, stream=null)
```

Member function “Console()” controls where an instance’s console output is directed. Console output consists of routine messages an instance produces as it initializes and undertakes calculations, as well as any warning and error messages that may be generated. By default, a ChemPlugin instance does not produce console output.

The function’s optional argument is the target for the console stream, which may be “stdout”, “stderr”, or the name of a dataset. When a client calls the function without an argument, or with null or an empty string as the argument, output to the console stream is disabled. A client may enable, disable, or redirect an instance’s console output at any time.

A client can direct an instance’s console output at declaration:

```
cp = ChemPlugin("stdout")
```

Since an instance produces no console output until directed to do so, this is the only way to capture messages produced when an instance comes into scope and initializes.

B.3.6.2 PrintOutput()

Syntax:

```
def PrintOutput(self, basename=None, label=None, rewnd=False)
```

Calling member function “PrintOutput()” triggers a ChemPlugin instance to append a data block to a print-format dataset.

The optional argument sets the file’s base name. The full file name is the base name combined with any suffix that may be set. For example, the calls

```
cp.Config("suffix _1")
cp.PrintOutput("myPrint.txt")
```

cause a block of output to be written to dataset “myPrint_1.txt”.

When a client calls “PrintOutput()” without an argument, the instance appends a data block to whatever file is open for print-format output. If none is open, the instance writes to “ChemPlugin_output.txt”.

The optional “label” argument allows the program to pass an optional character string to be written into the print dataset, at the head of the data block. For example,

```
cp.PrintOutput(None, "Initial condition")
```

would write the string "Initial condition" and then a data block to the currently open dataset.

Passing a true value as the optional third argument causes the instance to rewind the print-format dataset and write a data block at the head of the file. Use

```
cp.PrintOutput(None, None, true)
```

to write a data block at the head of the currently open dataset.

B.3.6.3 PlotHeader()

Syntax:

```
def PlotHeader(self, basename=None):
```

Member function "PlotHeader()" opens and initializes a plot-format dataset by writing header information to it.

A client may specify the file's base name as a string; the full name is the base name combined with the suffix, if one has been set. If a client calls the function without an argument, it writes to any file that may be open for plot output. If none is open, the function opens "ChemPlugin_plot.gtp", accounting for a suffix, if set. The ".gtp" extension identifies the file as **Gtplot** input.

B.3.6.4 PlotBlock()

Syntax:

```
def PlotHeader(self, basename=None):
```

Member function "PlotBlock()" appends to the currently open plot-format dataset a block of data representing an instance's current state. The dataset must have been initialized with a call to "PlotHeader()".

B.3.6.5 PlotTrailer()

Syntax:

```
def PlotTrailer(self):
```

Member function "PlotTrailer()" completes the plot-format output dataset by writing the trailing data structure. A trailer is required by program **Gtplot** before it can read the dataset.

When functions "ReportTimeStep()" and "AdvanceTimeStep()" detect time marching is complete, they automatically write a trailer to any open plot dataset, and close the dataset. It is not necessary to write a trailer, then, at the end of a time marching loop.

Significantly, a client may continue to append data blocks to a plot dataset, even after it has used "PlotTrailer()" to write a trailer to it. The additional data blocks overwrite the trailer data, so the client needs to call "PlotTrailer()" once again to close the dataset.

B.3.7 Convenience

B.3.7.1 Version()

Syntax:

```
def Version(self):
```

Member function “Version()” returns a pointer to a character string identifying the version of ChemPlugin in use.

B.3.7.2 ConvertUnit()

Syntax:

```
def ConvertUnit(self, value, old_unit, new_unit, mw=0.0, \
                mv=0.0, z=0.0, wmass=ANULL, smass=ANULL,\
               density=ANULL, vbulk=ANULL):
```

Member function “ConvertUnit” converts values from one unit to another. The **Units Recognized** appendix to this User’s Guide lists the units available for conversion.

There are three variants to the function. In the simplest variant, a client passes only the value to be converted, along with the old and new units, and the function returns the converted value.

Unit conversions involving mass and concentration may require additional information. To convert concentration from mmol/kg to mg/kg, for example, the function needs to know molecular weight. For other conversions, the function may require the molar volume, electrical charge on the species (to convert to and from meq/kg, for example), the mass of solvent water, the solution mass, the fluid density, and the bulk volume of the system.

In the second variant of the function call, a client also specifies the mole weight “mw” in g mol^{-1}, the mole volume “mv” in cm^3 mol^{-1}, and the ion charge “z” on the species in question. The function takes water mass (kg), solution mass (kg), density (g cm^{-3}), and bulk volume (cm^3) from the current state of the ChemPlugin instance.

In the final form, the client specifies the latter four variables, in the units listed. This variant of the function call can be helpful because it can be used before a client has calculated the initial system, by calling “Initialize()”.

Appendix: Configuration Commands

This chapter serves as a reference to the commands used to configure a ChemPlugin instance, and their syntax. You pass commands to a ChemPlugin instance with the "Config" member function. To pass the command “pH = 6” to an instance pointed to by “cp”, for example, you could include the statement

```
cp.Config("pH = 6");
```

in a client program written in C or C++.

C.1 Comparison to React

The commands for configuring ChemPlugin are maintained to closely mirror the commands used by the **React** application in The Geochemist’s Workbench software package. The ChemPlugin commands, nonetheless, differ in certain aspects from those for **React**, and the differences are described in this section.

C.1.1 Default values

The default values for two variables differ between ChemPlugin and **React**. Variable “delxi” defaults to 1.0 in ChemPlugin, but 0.01 in **React**. In the absence of other constraints on the reaction step, then, ChemPlugin would take a single step to complete a reaction path, whereas **React** would take 100.

The default for variable “step_increase” in ChemPlugin is 2.0, rather than the value of 1.5 that **React** carries as the default. A ChemPlugin instance, therefore, might increase the step size by default somewhat more quickly than the **React** application.

To set a ChemPlugin instance to use **React**’s defaults for the two variables, you would issue a call

```
cp.Config("delxi = .01; step_increase = 1.5");
```

from a C or C++ client program.

In addition, ChemPlugin is set by default to not produce print-format and plot datasets, whereas **React** writes these files by default. To set **React**’s default behavior in ChemPlugin, call

```
cp.Config("print = on; plot = on");
```

from the client program.

A ChemPlugin instance does not know at initialization time whether or not it is embarking on a polythermal simulation, since it may later be linked directly or indirectly to instances of differing temperature. To set up a simulation known to be isothermal, the client should issue

```
cp.Config("temperature = isothermal");
```

In this case, when the ChemPlugin instance assigns log Ks to reactions, it will behave as **React** does. Specifically, to assign a log K at one of the thermo dataset's primary temperatures, it will use the corresponding value directly, rather than fit log K to a polynomial versus temperature.

Finally, ChemPlugin by default writes no acknowledgment to the console upon completing a reaction step, but **React** writes a banner showing the step number, the point in reaction progress attained, and the number of iterations required to complete the step. The call

```
cp.Config("pluses = banner");
```

sets a ChemPlugin instance to behave in this sense like **React**.

C.1.2 Omitted commands

React maintains a user interface, whereas the client program serves as the interface for a ChemPlugin application. As such, the following commands are available in **React**, but not ChemPlugin: "clear", "clipboard", "go", "grep", "gtplot", "help", "polymorphs", "quit", "resume", "system", and "usgovt".

As well, **React**'s "audit" command is not available in ChemPlugin, since only the client program can audit mass balance globally.

C.1.3 Additional commands

ChemPlugin recognizes two commands, "Courant" and "Xstable", that are used to control time stepping, in light of the stability criteria for mass and heat transport. **React** does not consider transport and hence the commands are not needed.

ChemPlugin recognizes a command "pressure" that lets the client program control the density of the fluid, and hence its compression. ChemPlugin also recognizes commands "adjust_mass", "adjust_rate", and "resize", which are not available in **React**.

C.2 Command reference

The syntax of each command available to configure a ChemPlugin instance is shown below.

C.2.1 <unit>

<value> <**free**> <unit> <**as** element symbol> <basis entry>

To constrain the initial system, enter a command containing only the above entries. Entries may appear in any order. The qualifier “free” specifies that the constraint applies to the free rather than to the bulk basis entry. **ChemPlugin** recognizes the following units for constraining the initial system:

By mass or volume:

mol	mmol	umol	nmol	
kg	g	mg	ug	ng
eq	meq	ueq	neq	
cm3	m3	km3	l	

By concentration:

mol/kg	mmol/kg	umol/kg	nmol/kg	
molal	mmolal	umolal	nmolal	
mol/l	mmol/l	umol/l	nmol/l	
g/kg	mg/kg	ug/kg	ng/kg	
wt%	"wt fraction"			
g/l	mg/l	ug/l	ng/l	
eq/kg	meq/kg	ueq/kg	neq/kg	
eq/l	meq/l	ueq/l	neq/l	

By carbonate alkalinity (applied to bicarbonate component):

eq_acid	meq_acid	ueq_acid	neq_acid	
eq_acid/kg	meq_acid/kg	ueq_acid/kg	neq_acid/kg	
eq_acid/l	meq_acid/l	ueq_acid/l	neq_acid/l	
g/kg_as_CaCO3	mg/kg_as_CaCO3	ug/kg_as_CaCO3	ng/kg_as_CaCO3	
wt%_as_CaCO3				
g/l_as_CaCO3	mg/l_as_CaCO3	ug/l_as_CaCO3	ng/l_as_CaCO3	
mol/kg_as_Ca...	mmol/kg_as_Ca...	umol/kg_as_Ca...	nmol/kg_as_CaCO3	
mol/l_as_CaCO3	mmol/l_as_CaCO3	umol/l_as_CaCO3	nmol/l_as_CaCO3	

Per volume of the system:

mol/cm3	mmol/cm3	umol/cm3	nmol/cm3
kg/cm3	g/cm3	mg/cm3	ug/cm3
ng/cm3			
mol/m3	mmol/m3	umol/m3	nmol/m3
kg/m3	g/m3	mg/m3	ug/m3
ng/m3			
volume%	"vol. fract."		

By activity:

activity	fugacity	ratio	
pH	V	pe	
By partial pressure:			
Pa	MPa	atm	bar
psi			

Activity and fugacity may be abbreviated to "a" or "f". Keyword "total" reverses a setting of "free".

Use the "as" keyword to constrain mass in terms of elemental equivalents. For example, the command

```
CH3COO- = 10 umol/kg as C
```

specifies 5 umol/kg of acetate ion, since each acetate contains two carbons, whereas

```
20 mg/kg SO4-- as S
```

would specify 59.9 mg/kg of sulfate, since the ion's mole weight is about 3 times that of sulfur itself.

Examples:

```
55 mg/kg HCO3-
Na+ = 1 molal
1 ug/kg U++++
100 free cm3 Dolomite
pH = 8
Eh = .550 V
log f O2(g) = -60
HCO3- = 30 mg/kg as C
```

You may in a similar fashion constrain the concentration of a kinetic aqueous or surface complex. For example, the commands

```
kinetic AlF++
AlF++ = 1 umol/kg
```

set the concentration of the kinetic complex AlF^{++} to 1 μmol kg^{-1}.

C.2.2 <isotope>

```
<isotope | symbol> <fluid | reactant | segregated mineral> = <value>
```

Use the name of any isotope system loaded in the isotope dataset (also, the isotope's symbol) to set the isotopic composition of the initial fluid, reactant species (aqueous species, minerals, end members, gases, or oxides) or segregated minerals. The

composition may be set on any scale (e.g., SMOW, PDB, ...), but you must be consistent throughout the calculation.

For example, if the ^{17}O isotope system has been added to the isotope dataset, you could enter:

```
oxygen-17 fluid = -10, Quartz = +15
   or
17-O fluid = -10, Quartz = +15
```

Note that you use the name of the corresponding mineral to set the isotopic composition of an end member.

The commands

```
oxygen-17 remove
oxygen-17 off
```

clear all settings for ^{17}O isotopes from the calculation.

C.2.3 activity

activity <species> **=** <value>

Use the “activity” command (abbrev.: “a”) to constrain the activity of an aqueous species in the initial system. Examples:

```
activity Na+ = 0.3
log a H+ = -5
```

See also the “pH”, “Eh”, “pe”, “ratio”, “fugacity”, “fix”, and “slide” commands.

C.2.4 add

add <basis species>

Use the “add” command to include a basis species in the calculation. Example:

```
add HCO3-
```

See also the “swap”, “activity”, “fugacity”, “pH”, “pe”, and “Eh” commands.

C.2.5 adjust_mass

adjust_mass <mineral> <mass_increment> <unit> <**as** <element symbol>>

In ChemPlugin, use the “adjust_mass” command to add mass to (or remove mass from) a kinetically reacting mineral “on the fly”, i.e., during the course of a simulation. Arguments to the command are the kinetic mineral in question, the increment (positive)

or decrement (negative) to the reactant's mass, and the unit in which the mass change is provided. The unit may be any recognized by the "kinetic" (or "react") command.

Consider a client program that within a ChemPlugin instance "cpi" sets "K-feldspar" to react according to a kinetic rate law. You might embed within the client's time marching loop the member function call:

```
cpi.Config("adjust_mass K-feldspar 2 cm3");
```

The statement causes the instance to add 2 cm^3 to the current value for K-feldspar mass; if there are currently 10 cm^3 of the mineral in the reacting system, calling the member function in this way increases mass to 12 cm^3.

C.2.6 adjust_rate

adjust_rate <species | mineral | gas> <new_rate> <unit> <**as** <element symbol>>

ChemPlugin's "adjust_rate" command changes the rate at which a simple reactant is being added to the system "on the fly", i.e., during the course of a simulation. Arguments to the command are the simple reactant in question, the new rate of addition, and the unit in which the new rate is provided. The unit may be any recognized by the "react" command.

For example, consider a client program that adds "NaOH" as a simple reactant to ChemPlugin instance "cpi". You might embed within the client's time marching loop the statement:

```
cpi.Config("adjust_rate NaOH 50 mg/s");
```

The statement causes the instance to begin adding NaOH at a rate of 50 mg s^{-1}.

C.2.7 alkalinity

alkalinity = <value> <unit>

Use the "alkalinity" command to constrain the total concentration of HCO_3^- to reflect the solution's carbonate alkalinity. You can specify one of the units listed in the **Units Recognized** appendix; "mg/kg_as_CaCO3" is the default. To use this option, the solution pH must be set explicitly.

C.2.8 alter

alter <species | mineral | gas> <log *K*s>
alter <species | mineral | gas> <poly coefs> **TminK =** <value> **TmaxK =** <value>
alter <surface species> <poly coefs> **TminK =** <value> **TmaxK =** <value>
alter <surface species> **logK =** <value> **dlogK/dT =** <value>
alter <exchange species> **beta =** <value>
alter <sorbed species> **Kd =** <value>
alter <sorbed species> **Kf =** <value> **nf =** <value>
alter <solid solution> <type> <**discrete** | **continuous**> \
<**from** <value> **to** <value> **step** <value>> <<variable> **=** <value>>

Use the "alter" command to change the temperature expansion for a species', mineral's, or gas' log K, to adjust the stability of a surface species, or to change the properties of a solid solution.

Temperature expansions for log Ks are given in the thermo dataset either as T-tables, or by up to six coefficients of a polynomial. When the current thermo dataset uses T-tables, you list replacement values at each of the principal temperatures specified in the dataset, most commonly 0°C, 25°C, 60°C, 100°C, 150°C, 200°C, 250°C, and 300°C. Example:

```
alter Anhydrite -4.3009 -4.4199 -4.7126 -5.1758 -6.2299 500 500 500
```

Values of "500" represent a lack of data at the corresponding temperature.

For a thermo dataset constructed from polynomial expansions, on the other hand, list instead up to six polynomial coefficients

```
alter Anhydrite 4186 2.475 -0.001305 -85377 0 -794.4
```

You can optionally append a temperature range of validity, in Kelvins, for the polynomial:

```
alter Anhydrite 4186 2.475 -0.001305 -85377 0 -794.4 TminK= 293 TmaxK= 383
```

Absent a range, the polynomial is taken to span the principal temperatures. You may adjust a reaction's temperature range without specifying polynomial coefficients, but setting only coefficients resets the temperature range.

For a surface complex, use the "alter" command to set log K, its temperature derivative, or both

```
alter >(w)FeOCa+  logK = 6.0  dlogK/dT = 0.02
```

whereas for an ion exchange reaction, you set the selectivity coefficient β

```
alter >X2:Ca  beta = .033
```

directly, rather than as a logarithm. For a sorbing species, set K_d or, for a Freundlich species, K_f, n_f, or both, as follows

```
alter >Pb++ Kd = .03
alter >Sr++ Kf = .015 nf = .8
```

For a solid solution, use the "alter" command to set the activity coefficient model, discrete or continuous behavior, the composition range in terms of the mole fraction of the most recently specified end member, and parameters for the chosen activity model, as described for the "solid_solution" command. A solid solution "my_ss" might be set as a subregular Guggenheim solution with the command

```
alter my_ss guggenheim a0 = 1 a1 = 2
```

You may prefer to use the "alter" command to adjust an entry from the thermo database, and the "solid_solution" command to reconfigure a user-defined solution, but the two commands work interchangeably for this purpose.

Type "show alter" to list altered species or solutions and their settings; the "unalter" command reverses the process.

C.2.9 b-dot

b-dot

The "b-dot" command (formerly: "debye-huckel") causes the program to read the "thermo.tdat" thermodynamic dataset, which invokes the "B-dot" form of the extended Debye-Hückel equation to calculate activity coefficients for aqueous species.

C.2.10 balance

balance <**on**> <basis entry>
balance <**off**>

Use the "balance" command to specify the basis entry to be used for electrical charge balancing. The basis entry must be a charged aqueous species. By default, **ChemPlugin** balances on Cl^-.

The command "balance off" disables **ChemPlugin's** charge balancing feature. In this case, the user is responsible for prescribing charge-balanced input constraints.

C.2.11 carbon-13

carbon-13 <**fluid** | reactant | segregated mineral> **=** <value>

Use the "carbon-13" command (also, "13-C") to set the ^{13}C isotopic composition of the initial fluid, reactant species (aqueous species, minerals, end members, gases, or oxides) or segregated minerals. The composition may be set on any scale (e.g., PDB), but you must be consistent throughout the calculation. Example:

```
carbon-13 fluid = -10, Calcite = +4
```

Note that you use the name of the corresponding mineral to set the isotopic composition of an end member.

The commands

```
carbon-13 remove
carbon-13 off
```

clear all settings for ^{13}C isotopes from the calculation.

See also the "<isotope>" section above, and the "hydrogen-2", "oxygen-18", and "sulfur-34" commands.

C.2.12 chdir

chdir <directory>

Use the "chdir" command (abbrev.: "work_dir", "cd") to change the working directory. The program reads input scripts relative to the current working directory and writes output into it. Typing the command "chdir" without an argument causes the program to display the name of the working directory. The command

```
chdir ~
```

changes to the user's home directory, if one is defined by the operating system.

C.2.13 conductivity

conductivity <conductivity dataset>

Use the "conductivity" command to change the input file of coefficients used to calculate electrical conductivity. Example:

```
conductivity "..\my_conductivity.dat"
```

The dataset name may need to be enclosed in quotes if it contains unusual characters. Beginning with GWB11, the applications compute electrical conductivity using either of two different approaches, the USGS and APHA methods; the USGS method is the default. The required coefficients are defined in the files "conductivity-USGS.dat" and "conductivity-APHA.dat", respectively, which are installed in the same directory as the thermo datasets (commonly "\Program Files\GWB\Gtdata").

C.2.14 couple

couple <redox species | element(s) | **ALL**>

Use the "couple" command to enable any redox coupling reactions that have been disabled with the "decouple" command. You specify one or more redox species or elements. For example, the command

```
couple Carbon
```

couples all redox reactions involving the element carbon. Argument "ALL" enables all of the coupling reactions in the thermo dataset.

C.2.15 Courant

Courant = <value | **?**>

Use the "Courant" command to constrain the time step according to the Courant condition. You enter a value for the Courant number, which is the ratio of the distance fluid travels over a time step to the length of the nodal blocks. If you set a Courant number of one, then ChemPlugin will select a time step over which the fluid will exactly traverse the nodal blocks. For a value of 0.5, the fluid will move halfway across the nodal blocks, and so on. Values greater than one for the Courant number typically give unstable solutions and are therefore not recommended. By default, ChemPlugin assumes a Courant number of 1.0. The "?" argument resets the default value.

C.2.16 cpr

cpr = <field variable | **?**> <unit> <**steady** | **transient**>

Use the "cpr" command to set the heat capacity of the rock (mineral) framework. You can specify one of the units listed in the **Units Recognized** appendix; "cal/g°C" is the default. This value is used during polythermal simulations in calculating the effects of advective heat transport. The "transient" keyword causes the model to evaluate the field variable continuously over the course of the simulation, if it is set with an equation, script, or function.

By default, this variable is set to 0.2 cal/g°C. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.17 cpu_max

cpu_max = <value | **?**>

Use the "cpu_max" command to limit the amount of computing time a simulation may take. You set the maximum computing time in seconds, or use a "?" to restore the default state, which is no prescribed limit. To see the current setting, type "show variables".

C.2.18 cpw

cpw = <field variable | **?**> <unit> <**steady** | **transient**>

Use the “cpw” command to set in cal/g°C the heat capacity of the fluid. You can specify one of the units listed in the **Units Recognized** appendix; “cal/g°C” is the default. This value is used during polythermal simulations in calculating the effects of advective heat transport. The “transient” keyword causes the model to evaluate the field variable continuously over the course of the simulation, if it is set with an equation, script, or function.

By default, this variable is set to 1.0 cal/g°C. To restore the default value, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.19 data

data <thermo dataset> <**verify**>

Use the “data” command to change the input file of thermodynamic data. Example:

```
data "..\my_thermo.tdat"
```

The dataset name may need to be enclosed in quotes if it contains unusual characters. The “verify” option causes the program to read the named dataset only if it has not already been read.

C.2.20 decouple

decouple <redox species | element(s) | **ALL**>

Use the “decouple” command to disable the coupling reactions for one or more redox species, in order to calculate a model assuming redox disequilibrium. The redox species then become available for use as basis species and may be constrained independently of the original basis entries. You can disable as many coupling reactions as you want.

You specify either one or more redox species or elements. For example, the command

```
decouple Carbon
```

decouples all redox reactions involving the element carbon. Argument “ALL” disables all of the coupling reactions in the thermo dataset. Use the “couple” command to enable coupling reactions, once they have been disabled.

C.2.21 delQ

delQ = <value | **?**>

Use the "delQ" command to control the lengths of time steps taken in a simulation accounting for reaction kinetics. The program limits how much the ion activity product Q can change over a step, for each kinetic reaction considered. The setting for "delQ" is the projected change $\Delta Q/Q$ allowed in the relative value of the activity product. You can set a larger value to permit longer time steps, or a smaller value to improve stability. The default setting is 0.1. Type the command with no argument or with an argument of "?" to restore the default. To see the current setting, type "show variables".

C.2.22 delxi

delxi = <value | **?**> <**linear** | **log**>

Use the "delxi" command to set the maximum length (in terms of reaction progress, which varies from zero to one over the course of the simulation) of the reaction step, and to specify reaction stepping on a linear or logarithmic scale.

The command

```
delxi = .1 linear
```

for example, causes the program, in the absence of other constraints on the reaction step, to take steps through reaction progress of .1, .2, .3, . . . , 1.0. Alternatively, the commands

```
dx_init = .001
delxi = .5 log
```

produce a path with steps .001, .003, .01, .03, .1, .3, and 1.0 (see the "dx_init" command).

By default, this variable is set to 1.0 and "linear", which means in the absence of other constraints on the reaction step, the ChemPlugin will pass to the end of the simulation in a single step. To restore the default settings, type the command with no argument or with an argument of "?". To see the current settings, type "show variables".

C.2.23 density

density = <value | **?**>
density = <**TDS** | **chlorinity**>
density = <**batzle-wang** | **phillips**>
density = <**external** value>

You can use the "density" command to set in g/cm^3 the fluid density the program uses to convert compositional constraints to molality, the concentration unit it carries internally. If you set the initial Na^+ composition in mg/l, for example, the program needs to know the density of the initial fluid to determine Na^+ molality.

The program by default converts units using a density value it calculates automatically, as discussed below. This value is sufficient for most purposes, and

hence it is generally not necessary to set fluid density explicitly. You might, however, want to set the density if you are working at high temperature, but your analysis is expressed per liter of solution at room temperature.

You can also use the “density” command to tell the program how to calculate the default density it uses to convert units, and the fluid density it reports in the simulation results. The program by default uses the Batzle-Wang equation to figure density, but you can use the command ”density = phillips“ to select the method of Phillips et al., instead.

As well, the program normally figures density as that of an NaCl solution with the same TDS as the fluid in question, at the temperature of interest. With the command “density = chlorinity” you can tell the program to instead use the density of an NaCl solution of equivalent chlorinity.

The “external” keyword allows a client program to specify at any point in a simulation the fluid density for a ChemPlugin instance to use in its internal calculations. For example, the command

```
density = external 1.05
```

causes the ChemPlugin instance to carry a fluid density of 1.05 g/cm^3, instead of calculating fluid density on its own. By issuing a “density external” command at each step in a simulation, a client can ensure consistency between its own calculations, and those performed by a ChemPlugin instance the client has spawned.

To restore automatic calculation, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.24 dual_porosity

```
dual_porosity = <on | off> <spheres | blocks | fractures> \
  <geometry = <spheres | blocks | fractures | ?>> \
  <volfrac = <field variable | ?>> \
  <Nsubnode = <value | ?>> \
  <<radius | half-width> = <field variable | ?> <unit>> \
  <diff_length = <field variable | ?> <unit>> \
  <porosity = <field variable | ?>> \
  <retardation = <field variable | ?>> <steady | transient> \
  <diff_coef = <field variable | ?> <unit>> <steady | transient> \
  <thermal_con = <field variable | ?> <unit>> <steady | transient> \
  <theta = <value | ?>>  <reset | ?>
```

Use the “dual_porosity” command (abbrev: “dual”) to configure stagnant zones in the simulation, using the dual porosity feature. Enable and disable the feature with the “on” and “off” keywords. Disabling the feature does not affect other settings, so re-enabling the feature returns the model to its most recent configuration.

With the “geometry” keyword, you configure the stagnant zone into spheres, blocks, or a fractured domain, the latter being slabs separated by fractures arrayed along an arbitrary direction. Alternatively, you can set the three configurations directly with

keywords "spheres", "blocks", and "fractures". Keyword "volfrac" sets the fraction of the nodal block's bulk volume occupied by the stagnant zone.

The "Nsubnode" (or "nx") keyword sets the number of nodes into which the stagnant zone within each node will be divided when solving for solute and temperature distributions. The "radius" (or "half-width", for blocks and fractures) keyword sets the zone's characteristic dimension, in units of distance (see the **Units Recognized** appendix; default is cm), and keyword "diff_length" sets the distance (same units) from the contact with the free-flowing zone over which the model will account for solute diffusion and heat conduction. Use keywords "porosity", "retardation", "diff_coef", and "thermal_con" to set values for the porosity, retardation factor, diffusion coefficient, and thermal conductivity of the stagnant zone. The default unit for the diffusion coefficient is cm^2/s, and thermal conductivity defaults to cal/cm/s/°C units; see the **Units Recognized** appendix for a list of options.

By default, the stagnant zone is configured in spheres divided into 5 subnodes. You must specify a value for the volume fraction of the stagnant zone, as well as one for the radius (or half-width); the diffusion length defaults to the latter value. The program uses whatever values are set for the free-flowing zone in the node in question as default values for the porosity, diffusion coefficient, and thermal conductivity of the stagnant zone; the retardation factor defaults to a value of one.

The "theta" keyword sets time weighting ($0 \leq \theta \leq 1$) for the numerical solution of diffusive transport within the stagnant zone. A weight $\theta = 0$ assigns the explicit method, and larger values invoke an implicit solution. By default, the program chooses θ automatically, using the explicit method unless it would force too many more time steps than would otherwise be necessary. In that case, the program uses the implicit method ($\theta = 0.6$), which requires more computing effort per time step, but can take long steps without becoming numerically unstable.

You can append the "transient" keyword when setting several of the parameters: the diffusion coefficient, thermal conductivity, and retardation factor. If the variable is defined by an equation, script, or external function, it will then be re-evaluated continuously over the course of the run.

As an example, the command

```
dual_porosity geometry = spheres, radius = 50 cm, volfrac = 75%
```

configures the stagnant zone into spheres of half-meter radius that occupy three-quarters of the domain. The command

```
dual_porosity reset
```

enables the feature after restoring default settings for each keyword.

C.2.25 dump

dump <**off**>

Use the “dump” command to eliminate minerals present at the beginning of the reaction before the program begins to trace the path. Use

```
dump
dump off
```

C.2.26 dx_init

dx_init = <value | **?**>

Use the “dx_init” command to set the length of the initial time step. You set this value in terms of reaction progress, which varies from zero to one over the course of the simulation.

By default, the variable is ignored. To restore the default state, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.27 dxplot

dxplot = <value | **?**>

Use the “dxplot” command to set the interval in reaction progress (which varies from zero to one over the course of the simulation) between entries in the “ChemPlugin_plot.gtp” dataset. A value of zero causes the program to write the results after each step in reaction progress. This variable setting does not apply to reactions paths with logarithmic reaction stepping (see the “delxi” command), in which case all points in reaction progress are written to the plot dataset.

By default, this variable is set to 0.005. To restore the default value, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.28 dxprint

dxprint = <value | **?**>

Use the “dxprint” command to set the interval in reaction progress (which varies from zero to one over the course of the simulation) between entries in the “ChemPlugin_output.txt” dataset. A value of zero causes the program to write the results after each step in reaction progress, which can produce large amounts of output This variable setting does not apply to reactions paths with logarithmic reaction stepping (see the “delxi” command), in which case all points in reaction progress are written to the plot dataset.

By default, this variable is set to 0.01. To restore the default value, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.29 Eh

Eh = <value in volts | **?**>

Use the "Eh" command to set Eh in the initial system. Example:

```
Eh = 200 mV
```

sets the system's oxidation state to correspond to an Eh of 0.2 volt. See also the "activity", "pH", "pe", "fugacity", "fix", and "slide" commands.

C.2.30 end-dump

end-dump <**off**>

Use the "end-dump" command (also: "enddump") to eliminate minerals present at the end of the reaction path after the path has been traced. Use

```
end-dump
end-dump off
```

This operation can also be accomplished using the command "pickup fluid".

C.2.31 epsilon

epsilon = <value | **?**>

Use the "epsilon" command to set the convergence criterion (dimensionless) for iterating to a solution of the equations representing the distribution of chemical mass. By default, this variable is set to 5×10^{-11}. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.32 exchange_capacity

exchange_capacity = <value | **?**> <units>
exchange_capacity on <type> = <value | **?**> <units>

Use the "exchange_capacity" command (abbrev.: "ex_capacity" or "exch_capacity") to set the exchange capacitiy (i.e., the CEC) of the system when modeling ion exchange reactions or sorption according to Langmuir isotherms. For ion exchange reactions, you set units of electrical equivalents ("eq", "meq", and so on) or equivalents per gram of dry sediment ("eq/g", "meq/g", . . .). For Langmuir reactions, you similarly set a value in mole units: "mol", "mmol", "mol/g", "mmol/g". If you set units per gram of sediment, the program multiplies the value entered by the mass of rock in the system (including equilibrium and kinetic minerals as well as inert volume) to get the system's capacity.

If you read in a surface that sorbs by ion exchange or Langmuir isotherms, you must set a value for its exchange capacity. If you have set more than one sorbing surface (using the "surface_data" command), you identify the surface in question by its "type". For example

```
exchange_capacity on IonEx = .0008 eq/g
```

The "type" associated with each surface is listed at the top of each dataset of surface reactions. The "type" of the surface represented by the sample dataset "IonEx.sdat", for example, is "IonEx". You can use the "show" command to display the "type" of each active surface. See also the "surface_data" and "inert" commands.

C.2.33 explain

explain <species | mineral(s) | solid solution(s) | gas(es) | surface species>

Use the "explain" command to get more information (such as the mole weight of a species, a mineral's formula, mole volume, and density, or a solid solution's end-member minerals) about species, minerals, solid solutions, and gases in the dataset. Example:

```
explain Analcime
```

C.2.34 explain_step

explain_step <**off**>

The "explain_step" option causes the program to report on the Results pane the factor controlling the length of each time step, whenever the step size is limited by the need to maintain numerical stability.

C.2.35 extrapolate

extrapolate <**on** | **off**>

Use the "extrapolate" option to cause the program to extrapolate log Ks for reactions forming aqueous species, minerals, and gases to temperatures beyond the data's known range of validity. The option should be used with considerable care.

The temperature range of validity for a species' log K is defined by its temperature expansion, taken from the thermo dataset. For T-table expansions, it is the span of principal temperatures at which log Ks are not 500, whereas for polynomial expansions, the range is listed explicity.

In normal operation, the program loads only species with ranges of validity encompassing the calculation's temperature range. When the "extrapolate" option is "on", in contrast, the program projects to the temperature of interest the log Ks of species that would not otherwise be loaded.

Temperature ranges of validity can also be set in the thermo dataset for the virial coefficients used to calculate "Pitzer" and SIT activity coefficients. The "extrapolate on" option works in a similar manner in such cases, allowing virial coefficients to be used at temperatures beyond the coefficients' known ranges of validity.

C.2.36 fix

fix <unit> <species | gas>

Use the "fix" command to hold the activity of a species, fugacity of a gas, or an activity ratio constant over the course of a run. The <unit> can be "activity" or "fugacity" ("a" or "f" for short), "ratio", "pH", "pe", or "Eh", or it can be omitted. Examples:

```
fix pH
fix a H+
fix f O2(g)
fix ratio Ca++/Na+^2
```

C.2.37 flash

flash <**on** | **off** | **fluid** | **system**>

Use the "flash" command to set a "flash" model in which the original fluid (or fluid and minerals, if keyword "system" is used) is removed over the course of the reaction path. Keywords "on" and "fluid" are synonymous. The command "flash" without an argument is the same as "flash fluid". Generally, the fluid is replaced by a reactant fluid. Use

```
flash
flash system
flash off
```

C.2.38 flow-through

flow-through <**off**>

Use the "flow-through" command to turn on or off the flow-through option by which mineral precipitates are isolated from back-reaction. Use

```
flow-through
flow-through off
```

C.2.39 flush

flush <**off**>

Use the “flush” command to turn on or off the flush option by which fluid reactants displace existing fluid from the system over the course of the reaction path. Use

```
flush
flush off
```

C.2.40 fugacity

fugacity <gas> **=** <value | **?**>

Use the “fugacity” command (abbrev.: “f”) to set gas fugacities (on an atm scale) in the initial system. Examples:

```
fugacity O2(g) = .2
f CO2(g) = 0.0003
log f S2(g) = -30
```

Use “?” to unset a fugacity value: See also the “activity”, “ratio”, “pH”, “Eh”, “pe”, “fix”, and “slide” commands.

C.2.41 h-m-w

h-m-w

Use the “h-m-w” command (abbrev.: “hmw”) to set the program to calculate species’ activity coefficients by using the Harvie-Møller-Weare equations. Executing this command automatically sets the input dataset of thermodynamic data to “thermo_hmw.tdat”. Note that dataset “thermo_hmw.tdat” supports calculations at 25°C only.

C.2.42 heat_source

heat_source = <field variable | **?**> <unit> <**steady** | **transient**> \
<**Tmin =** <value | **min** | **?**> <unit>> <**Tmax =** <value | **max** | **?**> <unit>>
heat_source = <**off** | **on** | **reset**>

Use the “heat_source” command (also: “heat_src”) to set the rate of internal heat production within the medium. You can specify one of the units listed in the **Units Recognized** appendix; “cal/cm^3/s” is the default. The “transient” keyword causes the model to evaluate the field variable continuously over the course of the simulation, if it is set with an equation, script, or function. By default, the program does not account for internal heat production. The “?” argument resets the default value of zero.

The “Tmin” and “Tmax” keywords (also: “temp_max”, “temp_min”) prescribe the allowable temperature range for the simulation. Set the keywords to specific temperatures, or to “min” or “max”, which respectively represent the lowest and highest temperatures considered in the thermo dataset, as loaded at run time. If you set the temperature range directly, it will be bracketed to that of thermo dataset. For example, the command

```
heat_source Tmin = -1000 C, Tmax = 1000 C
```

is functionally the same as

```
heat_source Tmin = 0 C, Tmax = 300 C
```

when the "thermo.tdat" dataset is loaded, since the dataset's range is 0°C to 300°C.

The upper and lower temperature bounds serve two purposes. First, the simulation will give an error message and stop if temperature at any point in the domain falls more than 5°C less than the minimum value, or exceeds the maximum value by more than this amount. Second, unless the "extrap" option is set, the model will load for the simulation only those species for which log K values are available in the thermodynamic dataset over the specified temperature range.

The temperature bounds specified with the "Tmin" and "Tmax" keywords are the same as those set with the "span" command: The command

```
heat_source Tmin = 20 C, Tmax = 100 C
```

may be equivalently expressed

```
span 20 C  to 100 C
```

as long as the heat source option is enabled.

Values for the keywords default to the temperature span of the thermodynamic database, as set in the database header. Keyword "off" disables the heat source, leaving the source rate and temperature range intact, and keyword "on" re-enables the heat source; "reset" disables the source and discards any settings for the source rate and temperature range.

C.2.43 hydrogen-2

hydrogen-2 <**fluid** | reactant | segregated mineral> **=** <value>

Use the "hydrogen-2" command (also, "2-H") to set the ^{2}H isotopic composition of the initial fluid, reactant species (aqueous species, minerals, end members, gases, or oxides) or segregated minerals. The composition may be set on any scale (e.g., SMOW), but you must be consistent throughout the calculation. Example:

```
hydrogen-2 fluid = -120, Muscovite = -40
```

Note that you use the name of the corresponding mineral to set the isotopic composition of an end member.

The commands

```
hydrogen-2 remove
hydrogen-2 off
```

clear all settings for ^{2}H isotopes from the calculation.

See also the “<isotope>” section above, and the “carbon-13”, “oxygen-18”, and “sulfur-34” commands.

C.2.44 inert

inert = <value | **?**> <units>

Use the “inert” command to set the volume of non-reacting space in the system. You may set a value in units of volume, including cm^3, m^3, and l, as well as volume% and “vol. fract.”. The default setting is zero and the default unit is cm^3.

Assuming you have not set a value for the initial fluid fraction in the system using the “porosity” command, the program figures the porosity over the course of the calculation as a derived variable. Specifically, it divides the fluid volume by the sum of the fluid volume, mineral volume, and inert volume, and reports this value as a result.

When you have set a value for initial porosity with the “porosity” command, on the other hand, the program works in the contrary sense. In this case, it calculates the inert volume as that required to form a system of the specified initial porosity; the program now ignores any entry you may have set using the “inert” command.

In ChemPlugin, you can use this command to set inert volume "on the fly", as the client program progresses through the time marching loop. To do so, once a ChemPlugin instance has been initialized, send the instance an “inert” command using the “Config” member function:

```
cpi.Config("inert = 12 vol%");
```

In this way, the client program can, for example, propagate changes in fluid saturation to a ChemPlugin instance.

To restore the default state, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show”.

C.2.45 isotope_data

isotope_data <dataset>

Use the “isotope_data” command (also: “idata”) to set the name of the database containing isotope fractionation factors for species, minerals, and gases. Example:

```
isotope_data Isotope.mydata
```

C.2.46 itmax

itmax = <value | **?**>

Use the "itmax" command to set the maximum number of iterations that may be taken in an attempt to converge to a solution for the equations representing the distribution of chemical mass. By default, this variable is set to 999. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.47 Kd

Kd <**off**>

The "Kd" command controls whether the program calculates K_d distribution coefficients for sorbing components, in units of liters per kg sediment mass. This calculation requires that the mineral mass in the system (as specified for individual minerals and/or in terms of inert volume) be set correctly.

C.2.48 kinetic

kinetic <species | mineral | end member | gas> <variable> **=** <value>
kinetic <species | mineral | end member | gas> <variable> **=** <field variable> \
<**steady** | **transient**>
kinetic <species | mineral | end member | gas> <**apower** | **mpower(**species**)**> **=** <value>
kinetic <**redox(**label**)**> <variable> **=** <value>
kinetic <**redox(**label**)**> <variable> **=** <field variable> <**steady** | **transient**>
kinetic <**redox(**label**)**> <**apower** | **mpower(**species**)**> **=** <value>
kinetic <**microbe(**label**)**> <variable> **=** <value>
kinetic <**microbe(**label**)**> <variable> **=** <field variable> <**steady** | **transient**>
kinetic <**microbe(**label**)**> <**apower** | **mpower(**species**)**> **=** <value>
kinetic <**microbe(**label**)**> <**apower** | **mpower(**species**)**> **=** <value> \
<**apowerA** | **mpowerA(**species**)**> **=** <value> \
<**apowerD** | **mpowerD(**species**)**> **=** <value>

Use the "kinetic" command to set variables defining a kinetic rate law for (1) dissolution or precipitation of any mineral or end member in the initial system or reactant list, (2) the association or dissociation of any aqueous or surface complex in the system modeled, (3) the transfer of gases into or out of an external reservoir, (4) a redox reaction, including those promoted by catalysis or enzymes, or (5) a microbial metabolism.

In the first three cases, you identify the kinetic reaction by the name of the mineral, end member, species, or gas involved. In the case of a redox reaction, you set a label that begins with the characters "redox", such as "redox-1" or "redox-Fe". For a microbial reaction, set a label that starts with "microbe", such as "microbe-Ecoli".

The rate law you specify in a "kinetic" command, by default, applies to the dissolution of a mineral or end member, dissociation of a complex, dissolution of a gas, or forward progress of a redox or microbial reaction. The synonymous keywords "forward",

"dissolution", and "dissociation" set this behavior. Including in a "kinetic" command the keyword "reverse" or its synonyms "precipitation", "complexation", "association", or "exsolution" invokes the opposite behavior. In this case, the rate law applies to the reverse reaction: mineral or end member precipitation, complex association, or gas exsolution.

You can append the "transient" keyword when setting the following field variables: rate constant, specific surface area, activation energy, pre-exponential factor, nucleus area, and critical saturation index. If the variable is defined by an equation, script, or external function, it will then be re-evaluated continuously over the course of the run.

See also the "react" and "remove reactant" commands.

The following paragraphs apply to all types of kinetic reactions. You set the rate constant either directly using the "rate_con" keyword, or by setting an activation energy and pre-exponential factor with keywords "act_en" and "pre-exp". In the absence of promoting and inhibiting species (see next paragraph), you set the rate constant and preexponential factor in (1) mol/cm^2 s for mineral, end member, and gas transfer reactions, (2) molal/s or molal/cm^2 s (the latter when accounting for heterogeneous catalysis) for complexation and redox reactions, and (3) mol/mg s for microbial reactions. The activation energy is specified in J/mol. Example:

```
kinetic "Albite low" rate_con = 1e-15
```

You can set "rate_con", "act_en", and "pre-exp" as field variables (see the **Heterogeneity** appendix to the **GWB Reactive Transport Modeling Guide**).

You use the "apower" or "mpower" (also "apow" or "mpow") keyword to specify any promoting or inhibiting species in the kinetic rate law. Keyword "apower" sets the exponent of a species activity, and "mpower" the exponent of a species molality. Promoting species have positive powers, and the powers of inhibiting species are negative. For example, the command

```
kinetic "Albite low" apower(H+) = 1
```

sets H^+ as a promoting species, the activity of which is raised to a power of one.

You can use aqueous species, minerals (represented by activity, which is one, or molality), end members (activity or mole fraction), gas species (fugacity or partial pressure), surface complexes (molal concentration), and solvent water (activity) as promoting and inhibiting species. When setting an end member, use keyword "xpower" to set the rate law in terms of mole fraction. When setting a gas, use keyword "fpower" to set the rate law in terms of fugacity, and "ppower" to use partial pressure, instead. The generic keyword "power" sets the activity of the solvent or an aqueous species, the activity of an end member, the fugacity of a gas, and the molality of a surface species.

The "order1" and "order2" keywords set nonlinear rate laws. Keyword "order1" represents the power of the Q/K term, and "order2" represents the power of the (1 - Q/K) term.

Use the "rate_law" keyword to set the form of the kinetic rate law for a specific mineral, end member, species, gas, redox reaction, or microbial metabolism. You may set the keyword equal to (1) a character string containing the rate law, (2) the name of a file containing a basic-like script, or (3) the name of a function in a library. The name of a file containing a rate law script must end in ".bas". To specify a function from a library, set the name of a dynamic link library (DLL) separated from the function name by a colon (":"), such as "rate_laws.dll:my_ratelaw"; the library file must end in ".dll". To return to the program's built-in rate law, enter "rate_law = off" or "rate_law = ?".

The following paragraphs apply to dissolution and precipitation reactions. You set the specific surface area of a kinetic mineral or end member (in cm^2/g) with the "surface" keyword. For example,

```
kinetic "Albite low" surface = 1000
```

The "cross-affinity" option lets you use the saturation state of one mineral to model the reaction rate of another (or of an end member), as is sometimes useful for example in studying glass dissolution. To do so, you use the "xaffin" option. For (a hypothetical) example, the command

```
kinetic Quartz xaffin = Cristobalite
```

causes the program to calculate the reaction rate of quartz according to the fluid's saturation state with respect to cristobalite. The command

```
kinetic Quartz xaffin = OFF
```

turns off the option.

Finally, you use the "nucleus" and "critSI" keywords to set the area available for nucleation (in cm^2/cm^3 fluid volume) and the critical saturation index above which the mineral or end member can nucleate. Each of these values, by default, is zero.

Keywords "surface", "nucleus", and "critSI" can be set as field variables (see the **Heterogeneity** appendix to the **GWB Reactive Transport Modeling Guide**).

The following paragraphs apply to reactions for aqueous and surface complexes.

When you specify a kinetic reaction for the association of an aqueous complex or surface complex, or its dissociation, you can set the complex's initial concentration directly. The concentration can be set heterogeneously, as a field variable. If you do not specify an initial concentration, or set an entry of "?", the program takes the complex at the start of the simulation to be in equilibrium with the initial fluid.

You specify the initial concentration within a "kinetic" command or as a separate command line. For example, the commands

```
kinetic AlF++ rate_con = 3.3e-6, mpow(AlF++) = 1
AlF++ = 1 umol/kg
```

are equivalent to

```
kinetic AlF++ 1 umol/kg rate_con = 3.3e-6, mpow(AlF++) = 1
```

Either case defines a kinetic reaction for decomposition of the AlF^{++} ion pair, setting it initially to a free concentration of 1 μmol kg^{-1}.

The following paragraphs apply to gas transfer reactions. Use the "f_ext" keyword to specify the fugacity of the gas in question in the external reservoir, or keyword "P_ext" to set its partial pressure. In the latter case, you may append a pressure unit; the default is bar. Keyword "contact" sets the contact area between fluid and external reservoir, in cm^2/kg of water. Example:

```
kinetic CO2(g) f_ext = 10^-3.5, contact = 10
```

Both values can be set as field variables, as described in the **Heterogeneity** appendix to the **GWB Reactive Transport Modeling Guide**.

If you do not set a value for the gas' external fugacity, or set "f_ext = ?", the program uses the fugacity in the initial fluid, at the start of the simulation, and the external fugacity.

The following paragraphs apply to redox reactions. You set the form of the redox reaction to be considered as a character string, using the "rxn" (or "reaction") keyword. For example,

```
kinetic redox-1 rxn = "Fe++ + 1/4 O2(aq) + H+ -> Fe+++ + 1/2 H2O"
```

To specify that the reaction be promoted by a heterogeneous catalyst, set keyword "catalyst" to the name of the catalyzing mineral, or simply to "on". In the former case, you use keyword "surface" to set the specific surface area of the catalytic mineral (in cm^2/g). If you have set "catalyst = on", however, you use the "surface" keyword to set total catalytic area, in cm^2. Setting "catalyst = off" disables the catalysis feature.

To set an enzymatically promoted reaction, set keyword "me" to the name of the aqueous species serving as the enzyme, or simply to the value to be used as the enzyme's molality. In the former case, the program tracks the enzyme molality m_E over the course of the simulation from the calculated distribution of species. If you have set a numeric value for m_E using the "mE" keyword, the program uses this value directly. You may alternatively specify the enzyme species or its activity a_E using keyword "aE", in which case variables m_E, m_A, and m_P in the rate law are replaced by the activities a_E, a_A, and a_P.

For an enzymatic reaction, you further set the half-saturation constants K_A and K_P for the forward and reverse reactions in molal with the "KA" and "KP" keywords. You must set a value for K_A, but may omit K_P, in which case the m_P/K_P term in the rate law will be ignored. Setting "enzyme = off" disables the enzyme feature.

The following paragraphs apply to microbial reactions. You set the form of the metabolic reaction using the "rxn" (or "reaction") keyword, in the same manner as with redox reactions. For example,

```
kinetic microbe-1 rxn = "CH4(aq) + 2 O2(aq) -> HCO3- + H+ + H2O"
```

Set the half-saturation constants K_D and K_A for the electron donating and accepting reactions with the "KD" and "KA" keywords. These values default to zero.

You set the powers of species in the numerator of the rate law with the "mpower" keyword, as with other types of kinetic reactions. Use keywords "mpowerD" and "mpowerA" (or "mpowD" and "mpowA") to set the powers p_D, etc., of species from the electron donating and accepting reactions, respectively, within the product functions in the rate law's denominator. For example,

```
kinetic microbe-1 mpower(CH4(aq)) = 1, mpowerD(CH4(aq)) = 1
```

sets the power of the electron-donating species $CH_4(aq)$ to one in both the rate law numerator and denominator. Keywords "PKD" and "PKA" set the overall powers p_{KD} and p_{KA} of the electron donating and accepting terms in the denominator of the rate law; by default, these are one.

You set the free energy ΔG_{ATP} of ATP hydrolysis (in kJ/mol) with the "ATP_energy" keyword, and the value of n_{ATP} with keyword "ATP_number". These values default to zero.

Use the "biomass" keyword to set the initial biomass concentration, in mg/kg. You can set this value as a field variable (see the **Heterogeneity** appendix to the **GWB Reactive Transport Modeling Guide**).

The "growth_yield" keyword sets the microbe's growth yield in mg biomass/mol of reaction progress, and "decay_con" sets its decay constant in s^{-1}; both values default to zero.

C.2.49 log

log <variable> **=** <value>

Use the "log" command to set variables on a logarithmic scale. Examples:

```
log fugacity O2(g) = -65
log activity U++++ = -10
```

C.2.50 mobility

mobility = <surface type> <field variable> <**steady** | **transient**>

Use the "mobility" command to set up a complexing surface in your model as a mobile colloid. A mobile colloid is composed of the mineral (or minerals) associated with

a complexing surface, as well as the ion complexes present on that surface. Only datasets with model type “two-layer”, “three-layer”, or “cd-music” as set in the dataset header are surface complexation models, and hence only those datasets can be used to form a mobile colloid.

Mobility refers to the fraction of the surface in question that can move in the model by advection and dispersion. A surface with a mobility of one moves freely, whereas a mobility of zero sets the surface to be stationary. Intermediate values arise, for example, when some of the surface is attached to the medium, or when colloid motion is impeded by electrostatic interactions. By default in the software the mobility of any surface is zero.

To set a mobile colloid, begin by reading in a surface complexation dataset using the “surface_data” command. Then, use the “mobility” command, referencing the surface’s label, to set the colloid’s mobility. The label is given at the head of the surface dataset, on a line beginning “Surface type”. The label in dataset “FeOH.sdat”, for example, is “HFO”. If you omit the label, the program will assume you are referring to the surface complexation dataset most recently read.

You can define the mobility as a field variable, which means you can have the program calculate mobility using an equation, script, or compiled function you provide. When you set the “transient” keyword, the program upon undertaking each time step in the simulation evaluates mobility at each nodal block. In the “steady” case, which is the default, the program evaluates mobility at each block just once, at the start of the run.

Example:

```
surface_data FeOH.sdat
mobility HFO = 100%
```

where “HFO” is the label for the surface defined by dataset “FeOH.sdat”.

Restore the default behavior of immobility by entering a command such as

```
mobility HFO ?
```

C.2.51 no-precip

no-precip <**off**>

Use the “no-precip” command (also: “noprecip”) to prevent new minerals from precipitating over the course of a simulation. By default, they are allowed to precipitate. Use:

```
no-precip
no-precip off
```

See also the “precip” command.

C.2.52 nswap

nswap = <value | **?**>

Use the "nswap" command to set the maximum number of times that the program may swap entries in the basis in an attempt to converge to a stable mineral assemblage. By default, this variable is set to 30. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.53 oxygen-18

oxygen-18 <**fluid** | reactant | segregated mineral> **=** <value>

Use the "oxygen-18" command (also, "18-O") to set the ^{18}O isotopic composition of the initial fluid, reactant species (aqueous species, minerals, end members, gases, or oxides) or segregated minerals. The composition may be set on any scale (e.g., SMOW), but you must be consistent throughout the calculation. Example:

```
oxygen-18 fluid = -10, Quartz = +15
```

Note that you use the name of the corresponding mineral to set the isotopic composition of an end member.

The commands

```
oxygen-18 remove
oxygen-18 off
```

clear all settings for ^{18}O isotopes from the calculation.

See also the "<isotope>" section above, and the "carbon-13", "hydrogen-2", and "sulfur-34" commands.

C.2.54 pause

pause

Use the "pause" command to cause the instance to pause temporarily during input. This command is useful when you are debugging scripts.

C.2.55 pe

pe = <value | **?**>

Use the "pe" command to set oxidation state in the initial system in terms of pe. Example:

```
pe = 10
```

is equivalent to

```
log activity e- = -10
```

where “e-” is the electron. Use “?” to unset a pe value: See also the “activity”, “Eh”, “pH”, “fugacity”, “fix”, and “slide” commands.

C.2.56 permeability

permeability <**intercept =** field variable | **?**> <unit> <**steady** | **transient**> \
<**porosity =** field variable | **?**> <**steady** | **transient**> \
<mineral **=** field variable | **?**> <**steady** | **transient**>

You use the “permeability” command to set the correlation by which the program calculates sediment permeability. **ChemPlugin** calculates this value as a reported variable consistent with **X1t** and **X2t**, but does not use it in its calculations.

The correlation gives log permeability in any of the units listed in the **Units Recognized** appendix (darcys by default) as a linear function of the porosity (expressed as a volume fraction) of a nodal block and, optionally, the volume fractions of one or more minerals. The “transient” keyword causes the model to evaluate the coefficient in question continuously over the course of the simulation, if it is set with an equation, script, or function.

Examples:

```
permeability intercept = -11 cm2 porosity = 15
permeability Kaolinite = -8
```

The latter command adds a term for the mineral Kaolinite to the existing correlation. To remove a term from the correlation, set a value of “?”. The entry

```
permeability Kaolinite = ?
```

for example, removes the correlation entry for that mineral.

The default correlation is

$$\log k = -5 + 15\phi$$

where k is permeability in darcys and ϕ is porosity (expressed as a fraction).

C.2.57 pH

pH = <value | **?**>

Use the "pH" command to set pH in the initial system. Example:

```
pH = 5
```

is equivalent to

```
log activity H+ = -5
```

Use "?" to unset a pH value: See also the "activity", "Eh", "pe", "fugacity", "fix", and "slide" commands.

C.2.58 phrqpitz

phrqpitz

Use the "phrqpitz" command to set the program to calculate species' activity coefficients using the Harvie-Møller-Weare equations, as implemented in the USGS program PHRQPITZ. Executing this command automatically sets the input dataset of thermodynamic data to "thermo_phrqpitz.tdat". Note that dataset "thermo_phrqpitz.tdat" is primarily intended to support calculations at or near 25°C.

C.2.59 pickup

pickup <**system =**> <**entire** | **fluid**>
pickup reactants = <**entire** | **fluid** | **minerals**>

Use the "pickup" command to take the results of a reaction path as the starting point for a new reaction path. You may pick up the entire system, or just the fluid or minerals resulting from a reaction path, and use this as your new initial system or reactant list. Default choices are "system" and "entire". Examples:

```
pickup
pickup fluid
pickup reactants
pickup reactants = minerals
```

When you do a simple "pickup" (i.e., "pickup system = entire"), the program retains within the system all kinetic reactions that were defined in the original path, at the reactions' endpoint state. A "pickup fluid" (fully, "pickup system = fluid") command retains only the kinetic reactions occurring in the fluid – the kinetic redox and aqueous complexation reactions – in the new reaction path; kinetic reactions involving minerals, surfaces, a gas phase, or microbes are discarded.

Picking up the entire endpoint system, or just the endpoint minerals, as reactants (i.e., “pickup reactants” or “pickup reactants = minerals”) causes the program to retain the kinetic reactions involving mineral precipitation and dissolution. In these cases, the other types of kinetic reactions are discarded. The command “pickup reactants = fluid” causes the program to discard any kinetic reactions that may be set.

The commands

```
pickup TDS
pickup density
```

are obsolete, because releases 7.0 and later of the software calculate the TDS and density automatically.

C.2.60 pitz_dgamma

pitz_dgamma = <value | **?**>

Use the “pitz_dgamma” command to control the relative change in an activity coefficient’s value the program allows during each Newton-Raphson iteration, when the Harvie-Møller-Weare activity model has been invoked. By default, the program allows a 10% change, which corresponds to a value of 0.1.

C.2.61 pitz_precon

pitz_precon = <value | **?**>

Use the “pitz_precon” command to control the maximum number of passes the program takes through the pre-conditioning loop before beginning a Newton-Raphson iteration, when a virial activity model (Harvie-Møller-Weare or SIT) has been invoked. By default, the program makes up to 10 passes. In cases of difficult convergence, counter-intuitively, it can sometimes be beneficial to decrease this value.

C.2.62 pitz_relax

pitz_relax = <value | **?**>

The “pitz_relax” command controls under-relaxation when evaluating the Harvie-Møller-Weare equations. The program at each Newton-Raphson iteration assigns activity coefficients as a weighted average of the newly calculated value and the corresponding value at the previous iteration level. Setting pitz_relax to zero eliminates under-relaxation, so the newly calculated values are used directly; a value of one, in contrast, should be avoided because it would prevent the activity coefficients from being updated. By default, the program carries an under-relaxation factor of 0.5.

C.2.63 plot

plot <**character** | **binary**> <**on** | **off**>

Use the "plot" command to set the format of the plot interface dataset. The dataset is written in XML, a standard format that is easy to parse for use with alternative plotting programs. Numerical data in the dataset can be represented in either standard decimal notation (keyword "character") for user readability or a binary encoding (keyword "binary") that maintains full precision of data. The default format, XML with binary encoded data, also zips the output file to reduce output size and improve file opening speed. The command "plot off" causes **ChemPlugin** to bypass writing calculation results to the "ChemPlugin_plot.gtp" dataset, which is used to pass input to **Gtplot**. By default, the program writes output to the dataset. The command "plot on" (or just "plot") re-enables the output. To see the current setting, type "show print".

C.2.64 pluses

pluses <**off**>

Use the "pluses" command to cause **ChemPlugin** to simply output a plus sign ("+") each time it iterates to a solution, rather than printing a banner showing the number of iterations required and final residual value.

C.2.65 porosity

porosity = <field variable | **?**>

Use the "porosity" command to set (as a volume fraction) the initial porosity of the system. Porosity, the fraction of the system occupied by fluid, is the ratio of fluid volume to the sum of fluid, mineral, and inert volume.

The examples

```
porosity = 0.30
porosity = 30%
```

are equivalent.

When you specify the porosity, the program will figure the difference between the volume of a system of the given porosity and fluid volume, and the volume taken up initially in the system by minerals and fluid. The program assigns this difference as inert, non-reactive volume (see the "inert" command). In this case, the program ignores any settings that may have been made with the "inert" command.

When you do not specify an initial porosity with the "porosity" command, on the other hand, the program calculates it from volumes in the system of fluid, minerals, and inert space. To restore this default behavior, enter the command with an argument of "?".

C.2.66 precip

precip <**off**>

Use the “precip off” command to prevent new minerals from precipitating over the course of a simulation. By default, they are allowed to precipitate. Use

```
precip
precip off
```

See also the “no-precip” command.

C.2.67 press_model

press_model <**Tsonopoulos** | **Peng-Robinson** | **Spycher-Reed** | **default** | **off**>

The “press_model” command (also: “pressure_model”) lets you control the method used to calculate fugacity coefficients and gas partial pressures. Three pressure models are coded in the software: Tsonopoulos, Peng-Robinson, and Spycher-Reed, as described in the **GWB Essentials Guide**.

By default, the pressure model is taken from the header lines of the thermo dataset in use, but you can use the “press_model” command to override the default setting. Keywords “Tsonopoulos”, “Peng-Robinson”, and “Spycher-Reed” set the pressure model directly (you need only enter the first three letters), whereas “default” returns to the setting in the thermo dataset, and “off” disables the feature, forcing all fugacity coefficients to one.

Examples:

```
press_model Peng-Robinson
press_model default
```

C.2.68 pressure

pressure <value> <unit>

The “pressure” command (also: “P”) allows the client program to control the value of fluid pressure used to calculate the fluid density. The command, in other words, controls compression of the fluid. The pressure setting also figures into the calculation of gas fugacity coefficients.

The setting serves no other purpose. Specifically, changing the pressure setting does not affect the log *K*s or the determination of activity coefficients.

Examples:

```
pressure 1
pressure 10 MPa
```

By default, the pressure carried is taken from the value given in the thermo dataset for the temperature of interest, interpolated as necessary. The default unit is bar.

C.2.69 print

print <option> **=** <**long** | **short** | **none**>
print <**off** | **on**>
print <**numeric** | **alphabetic**>

Use the "print" command (also: "printout") to control the amount of detail to be written into the "ChemPlugin_output.txt" dataset. For example, the dataset can contain information about each aqueous species, information on only species with concentrations greater than 10^{-8} molal, or no species information.

Options, which may be abbreviated to three letters, and their default settings are:

species	short
surfaces	long
saturations	short
gases	long
basis	none
orig_basis	long
elements	long
reactions	none
stagnant	none

The "print" command can also be used to arrange entries in the output dataset either numerically or alphabetically:

```
print numeric
print alphabetic
```

To see the current print settings, type "show print". Finally, the command "print off" causes **ChemPlugin** to bypass writing calculation results to the "ChemPlugin_output.txt" dataset. By default, the program writes to the datasets. The command "print on" (or just "print") re-enables the output.

C.2.70 pwd

pwd

The "pwd" command returns the name of the current working directory. The command has the same effect as typing "show directory". See the "chdir" command.

C.2.71 ratio

ratio <species ratio> **=** <value | **?**>

Use the “ratio” command to constrain an activity ratio in the initial system. Example:

```
swap Ca++/Na+^2 for Ca++
ratio Ca++/Na+^2 = 0.2
```

Use “?” to unset a ratio value: See also the “activity”, “pH”, “Eh”, “pe”, “fugacity”, “fix”, and “slide” commands.

C.2.72 react

react <amount> <unit> <**as** <element symbol>> \
<species | mineral | end member | gas | oxide> <**cutoff**> **=** <value>

Use the “react” command (abbrev.: “rct”) to define the reactants for the current simulation. To set a kinetic rate law for a reactant, use the “kinetic” command.

Units for the amount of reactant to add over a reaction path can be:

mol	mmol	umol	nmol	
kg	g	mg	ug	ng
eq	meq	ueq	neq	
cm3	m3	km3	l	
mol/kg	mmol/kg	umol/kg	nmol/kg	
molal	mmolal	umolal	nmolal	
mol/l	mmol/l	umol/l	nmol/l	
g/kg	mg/kg	ug/kg	ng/kg	
wt%	"wt fraction"			
g/l	mg/l	ug/l	ng/l	
eq/kg	meq/kg	ueq/kg	neq/kg	
eq/l	meq/l	ueq/l	neq/l	

Units of mass or volume can be expressed per volume of the porous medium. Examples:

mol/cm3	g/cm3
mmol/m3	ug/m3
volume%	"vol. fract"

Units of mass or volume can be set as absolute rates by appending “/s”, “/day”, “/yr”, or “/m.y.”. For example,

mmol/s	g/day	cm3/yr
mol/kg/s	mg/kg/day	cm3/kg/yr

Use the “as” keyword to specify reactant masses as elemental equivalents. For example, the command

```
react 10 umol/kg CH3COO- as C
```

specifies 5 umol/kg of acetate ion, since each acetate contains two carbons, whereas

```
react 20 mg/kg SO4-- as S
```

would cause the program to add 59.9 mg/kg of sulfate, since the ion's mole weight is about 3 times that of sulfur itself.

You can set a cutoff to limit the amount of a reactant. For example, if you set the amount of a reactant to two moles and set a cutoff of one, then **ChemPlugin** will add one mole of the reactant over the first half of the path and none over the second half. Enter the cutoff value in the same units as the amount of reactant. Examples:

```
react 10 grams Quartz
react 1e-2 mol Muscovite cutoff = .5e-2
react .01 mol/day HCl
```

See also the "kinetic" and "remove reactant" commands.

C.2.73 reactants

reactants times <value>

Use the "reactants" command to vary the total amounts of the reactants by a given factor.

Example:

```
reactants times 1/10
```

C.2.74 read

read <dataset>

Use the "read" command to begin reading commands from a script stored in a dataset. Example:

```
read Seawater
```

Control returns to the user after the script has been read, unless the script contains a "quit" command. You can also use the "read" command in place of the "data" or "surface_data" command to read a thermo or surface reaction dataset.

When typing a "read" command, you can use the spelling completion feature to complete dataset names: touch "[tab]" or "[esc]" to cycle through the possible completions, or **Ctrl+D** to list possible completions.

C.2.75 remove

remove <basis specie(s)> <solid solution(s)> <reactant(s)>
remove basis <basis specie(s)>
remove solid_solution <solid solution(s)>
remove reactant <reactant(s)>

Use the "remove" command (also: "rm") to eliminate one or more basis entries or reactants from consideration in the calculation. Example:

```
remove Na+
remove Quartz Calcite
remove reactant H2O
```

Components can be reentered into the basis using the "swap", "add", "activity", and "fugacity" commands. You can also use the "remove" command to remove solid solutions configured in a run (but not those set in the thermo dataset):

```
remove solid_solution mySS
```

C.2.76 report

report <option>
report set_digits <value>

Once the program has completed a calculation, you can use the "report" command to return aspects of the calculation results. For arguments available, see the **Report Command** appendix to this User's Guide.

C.2.77 reset

reset
reset system
reset reactants
reset variables

Use the "reset" command to begin defining the chemical system again with a clean slate. Your current settings will be lost, and all options will be returned to their default states. The command, however, does not alter the setting for the thermo dataset. The "reset system" command resets only the initial system. Similarly, typing "reset reactants" resets the reactant system, and "reset variables" sets each settable variable to its default value.

C.2.78 resize

resize <**system** | **fluid** | **minerals** | **inert** | **rock**> <value> <unit | **times**>

Use the "resize" command once a ChemPlugin instance has been initialized to change the volume extent of the entire system, or just that of the fluid, mineral, inert, or rock fraction. The latter option comprises the minerals in the system plus any inert volume.

You may set a value in terms of a volume unit: cm^3, m^3, km^3, or liters

```
resize system 100 m3
resize fluid 20 L
```

You can also use the "times" keyword to scale the target by a given factor. The command

```
resize system times 2
```

for example, doubles the volume extent of the system. Default settings are "system" as the target, and "cm3" for the unit.

Note the "resize" command can be executed only after the instance has been initialized by calling the "Initialize()" member function. Running the command before initialization has no effect on the instance's volume extent.

C.2.79 save

save <dataset> <**hex**>

Use the "save" command to write the current chemical system into a dataset in **ChemPlugin** format commands. The dataset can be used as an **ChemPlugin** input script. Examples:

```
save
save kspar.rea
```

The optional keyword "hex" causes the program to output numbers as hexadecimal values.

C.2.80 script

script
script end

Use the "script" command to mark the beginning, and optionally the end, of a control script. Control scripts differ from standard input files in that they can contain not only **ChemPlugin** commands, but control structures such as loops and if-else branches. Control scripts follow the Tcl syntax, described in www.tcl.tk and mini.net/tcl, as well as several widely available textbooks.

Within a control script, filenames are written with double rather than single backslashes. For example, a "read" command might appear as

```
read GWB_files\\My_file.rea
```

within a control script.

C.2.81 segregate

segregate <mineral(s)>
segregate <mineral> <value>
segregate <mineral> <initial value> <final value>
segregate <mineral> <value> **Xi =** <value> <value> **Xi =** <value>

The “segregate” command causes minerals to be isolated from isotopic exchange over the course of a reaction path. By default, a mineral in the equilibrium system remains in isotopic equilibrium with the fluid and other minerals. A segregated mineral, on the other hand, changes in isotopic composition only when it precipitates from solution; it alters the system’s composition only if it dissolves. Example:

```
segregate Quartz Calcite "Maximum Microcline"
```

Optionally, a fraction of a mineral’s mass may be isotopically segregated, and that fraction may vary linearly with reaction progress. Examples:

```
segregate Quartz 100%, Muscovite 7/10
segregate Ca-Saponite 100% 0%
segregate Ca-Saponite 80% Xi = .3, 20% Xi = .7
```

In the latter example, the program segregates 80% of the mass of Ca-Saponite until the reaction progress variable Xi reaches .3, decreases the segregated fractionation until it reaches 20% when Xi equals .7, and then holds the value constant until the end of the path. To display the isotopically segregated minerals, type “show isotopes”.

Note, you use the name of the corresponding mineral to segregate a solid solution end member.

C.2.82 show

show <option>
show <**aqueous** | **minerals** | **solid_solutions** | **gases** | **oxides** | **surfaces**> \
  <**with** | **w**/> <basis entry | string>

Use the "show" command to display specific information about the current system or database. The options are:

```
show                show all           show altered
show aqueous        show basis         show commands
show couples        show directory     show elements
show gases          show initial       show isotopes
show minerals       show oxides        show printout
show reactants      show show          show solid_solutions
show suppressed     show surfaces      show system
show variables
```

The command "show show" gives a list of show command options. When you type "show aqueous" (or "minerals", "solid_solutions", "gases", "oxides", or "surfaces"), the program lists all entries of that type in the thermo database. The "solid_solutions" option additionally includes solutions defined locally. A long form lets you limit the query to entries composed of a particular basis species or containing a text string in the name:

```
show aqueous with Al+++
show minerals w/ chal
```

There is also a compound form of the "show couples" command:

```
show coupling reactions
```

This command produces a complete list of the redox couples, in reaction form.

C.2.83 simax

simax = <value | **?**>

The "simax" command sets in molal units the maximum value of the stoichiometric ionic strength used in calculating water activity when Helgeson's B-dot Debye-Hückel model is employed. By default, this variable is set to 3 molal. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.84 slide

slide <unit> <species | gas> **to** <value>

Use the "slide" command to linearly adjust the activity of the specified species, fugacity of the gas, or an activity ratio toward <value>, which is attained at the end of the path. Note that the interpolation is made linearly on the logarithm of activity or fugacity if <value> is set as a log, and that <unit> can be "activity" or "fugacity" ("a" or "f" for short), "ratio", "pH", "pe", or "Eh", or omitted. Examples:

```
slide pH to 5
slide activity Cl- to 2/3
slide f CO2(g) to 10^-3.5
slide log f O2(g) to -65
```

C.2.85 solid_solution

solid_solution <**name =** <label>> <type> <**discrete** | **continuous**> \
  <<mineral> **tag =** <label>> <**contains**> <<mineral> **tag =** <label>> \
  <**from** <value>> <**to** <value>> <**step =** <value>> <<variable> **=** <value>>
solid_solution <solid solution> <options>
solid_solution remove <solid solution>

Use the “solid_solution” command (abbrev.: “solid_soln”) to define binary solid solutions in addition to any provided in the current thermo database. Supply a name with the “name” keyword, or simply set one as the first argument.

You can select the type of solid solution, either “ideal” (the default), “site_mixing”, “guggenheim”, “regular”, “cubic”, or “third_order”. For a site-mixing solution, set the site number “bsite”, which defaults to 1. For a Guggenheim solution, set dimensionless parameters “a0”, “a1”, and “a2” or a temperature expansion for their J/mol counterparts using coefficients “p1”-“p9”, each of which defaults to zero. For the other nonideal models, similarly, set coefficients “p1”, “p2”, etc., as appropriate. For more information, see **Solid solutions** under **Configuring the programs** in the **GWB Essentials Guide** and the **Thermo Datasets** chapter in the **GWB Reference Manual**. You can as well choose whether the solid solution is to be treated as continuous, the default option, or as a sequence of “discrete” minerals.

You need to specify two end members for a solid solution, each associated with an abbreviated “tag”. The tag cannot be the name of another entry in the database. You can set a composition range in terms of the mole fraction of the most recently referenced end member, using keywords “from” and “to”, as well as a compositional “step” separating tranches of a discrete solution; by default, composition ranges from zero to one and the step is one-twentieth of the range. Keyword “contains” may be inserted for clarity. Examples:

```
solid_solution name = my_ss Calcite contains Strontianite from 0% to 10% a0= 0.05
solid_solution my_ss Calcite Strontianite from 0 to .1 a0= 0.05
solid_solution Biotite site_mixing Annite Phlogopite bsite= 3
```

If you don’t specify a solid solution’s name, the app will create one automatically using the tags for the two end members, separated by a hyphen. If you don’t set a tag for an end member, the app will pick up the first six characters of the corresponding mineral’s name. For example, the command

```
solid_solution Calcite contains Strontianite
```

produces a solid solution named "Calcit-Stront", whereas typing

```
solid_solution Calcite tag Cc contains Strontianite tag Str
```

yields the name "Cc-Str". To redefine the properties of a solid solution, enter the name followed by the properties to be changed:

```
solid_solution my_ss a0 = 1 a1 = 2
```

(You may prefer to use the "alter" command to change solutions defined within the thermo database; the two commands are functionally equivalent.) The command

```
solid_solution remove my_ss
```

discards a previously defined solution.

C.2.86 sorbate

sorbate <**exclude** | **include**>

Use the "sorbate" command to tell the program, when considering sorption onto surfaces (see the "surface_data" command), whether to include or exclude sorbed species in figuring the composition of the initial system. By default, the program does not include sorbed species in this calculation. If you set the Ca^{++} concentration to 15 mg/kg, for example, the initial system would contain that amount in the fluid and an additional amount sorbed onto mineral surfaces. If you type the command "sorbate include", however, that amount would apply to the sum of the Ca^{++} sorbed and in solution.

C.2.87 span

span <value | **min** | **?**> <unit> <**to**> <value | **max** | **?**> <unit>
span <**off** | **on** | **reset**>

Use the "span" command to configure a ChemPlugin instance to be ready to vary in temperature when heat is transfered to or from neighboring instances. You set lower and upper limits to the anticipated variation. The lower limit may be identified with keyword "from" or "Tmin", and the upper denoted by "to" or "Tmax". A value of "?" unsets the bound in question.

A "span" command is generally required when creating polythermal simulations involving heat transfer among instances, because an instance cannot know as it is initialized whether such transfer will occur, or how it will affect temperature. For example, the commands

```
T = 25 C
span to 200 C
```

cause a ChemPlugin instance to be initialized at 25°C and accept net heat transfer that warms it to no more than 200°C.

The instance will load only species for which log K values are available in the thermodynamic dataset across the specified temperature range, unless the "extrap" option is set. Member function "AdvanceHeatTransfer()" will furthermore return an error condition if temperature at any point in the domain falls more than 5°C less than the minimum value, or exceeds the maximum value by more than this amount.

The "span" command can be used for purposes beyond simulating heat transport. With T-table datasets, simply setting "span = on" without specifying a range forces the program to load species' log Ks in polynomial mode. In other words, it will take log Ks from polynomial fits against temperature to the stability constants in the thermo dataset, even if the calculation is performed entirely at one of the dataset's principal temperatures. Otherwise, an isothermal calculation run at one of the principal temperatures (commonly 0°C, 25°C, 60°C, ...) takes log Ks for aqueous species, minerals, and so on directly from the values given in the thermo dataset. You can also use the span command to force two instances at different temperatures to load the same set of aqueous species, minerals, and so on.

Temperature ranges of validity can also be set in the thermo dataset for the virial coefficients used to calculate "Pitzer" and SIT activity coefficients. The "span" command works in a similar manner in such cases, loading only virial coefficients with validity that spans the calculation's temperature range.

In any case, you set the bounds to specific temperatures, or to "min" or "max", which respectively represent the lowest and highest temperatures considered in the thermo dataset, as loaded at initialization. If you set the temperature range directly, it will be bracketed to that of the thermo dataset. For example, the command

```
span -1000 C to 1000 C
```

is functionally the same as

```
span 0 C to 300 C
```

when the "thermo.tdat" dataset is loaded, since the dataset's range is 0°C to 300°C.

The temperature bounds set here are the same as those set with the "heat_source" command: The command

```
span 20 C to 100 C
```

may be equivalently expressed

```
heat_source Tmin = 20 C, Tmax = 100 C
```

as long as the heat source option is enabled.

Keyword "off" disables the feature, leaving values for the limits intact, and keyword "on" re-enables the feature. The "reset" keyword clears the limits, disabling the feature. By default, the span feature is disabled.

C.2.88 start_date

start_date <value | **off**>

Use the "start_date" command to set an explicit starting date of the reaction. This can be used to coordinate the plotting of dated scatter data samples stored in a **GSS** spreadsheet on the reaction path in **Gtplot**. The date should be in the format "MM/DD/YYYY". Use "off" to return to the default of not set.

```
start_date 10/30/2008
start_date off
```

C.2.89 start_time

start_time <value | **off**>

Use the "start_time" command to set an explicit starting time of the reaction. This can be used to coordinate the plotting of timed scatter data samples stored in a **GSS** spreadsheet on the reaction path in **Gtplot**. The time should be in the format "HH:MM:SS". Use "off" to return to the default of not set.

```
start_time 11:30:00
start_time off
```

C.2.90 step_increase

step_increase = <value | **?**>

Use the "step_increase" command to set the greatest proportional increase, from one step to the next, in the size of the time step. This variable does not apply to reaction paths with logarithmic reaction stepping (see variable "delxi").

By default, this variable is set to 2.0. To restore the default value, type the command with no argument or with an argument of "?". To see the current setting of this variable, type "show variables".

C.2.91 step_max

step_max = <value | **?**>

Use the “step_max” command to limit the number of reaction steps an instance may take to trace a simulation. Use a “?” to restore the default state, which is no prescribed limit. To see the current setting, type “show variables”.

C.2.92 suffix

suffix <string>

Use the “suffix” command to alter the names of the output datasets (“ChemPlugin_output.txt”, and “ChemPlugin_plot.gtp”) by adding a trailing string. Example:

```
suffix _run2
```

produces output datasets with names such as “ChemPlugin_output_run2.txt”.

C.2.93 sulfur-34

sulfur-34 <**fluid** | reactant | segregated mineral> **=** <value>

Use the “sulfur-34” command (also, “34-S”) to set the ^{34}S isotopic composition of the initial fluid, reactant species (aqueous species, minerals, end members, gases, or oxides) or segregated minerals. The composition may be set on any scale (e.g., CDT), but you must be consistent throughout the calculation. Example:

```
sulfur-34 fluid = +45, H2S(g) = -2
```

Note that you use the name of the corresponding mineral to set the isotopic composition of an end member.

The commands

```
sulfur-34 remove
sulfur-34 off
```

clear all settings for ^{34}S isotopes from the calculation.

See also the “<isotope>” section above, and the “carbon-13”, “hydrogen-2”, and “oxygen-18” commands.

C.2.94 suppress

suppress <species, minerals, solid solutions, gases, surface species | **ALL**>

Use the "suppress" command (also: "kill") to prevent certain aqueous species, surface species, minerals, solid solutions, or gases from being considered in a calculation. Example:

```
suppress H3SiO4- Quartz "Maximum Microcline"
```

prevents the three entries listed from being loaded from the database. Typing "suppress ALL" suppresses all of the minerals and solid solutions in the thermodynamic database, as well as any solid solutions defined locally.

The "unsuppress" command reverses the process. To suppress all but a few minerals, you could type

```
suppress ALL
unsuppress Quartz Muscovite Kaolinite
```

C.2.95 surface_capacitance

surface_capacitance = <value | **?**>
surface_capacitance C1 = <value | **?**> **C2 =** <value | **?**>
surface_capacitance on <type> **=** <value | **?**>

Use this command (abbrev.: "surf_capacitance") to set, in units of F/m^2, the capacitance or capacitances of a sorbing surface.

For a two-layer complexation model, when you set a capacitance with the command, or if a value for capacitance is set in the header section of the surface reaction dataset, **ChemPlugin** will model surface complexation for the surface in question using the constant capacitance model, rather than the full two-layer model.

A triple-layer or CD-MUSIC complexation model, on the other hand, requires two capacitances, C_1 and C_2. Default capacitances are set in the header of the surface dataset, but you may use the "C1" and "C2" arguments to override the default settings from the command line.

If you have set more than one sorbing surface (using the "surface_data" command), you identify the surface in question by its "type". For example,

```
surface_capacitance on HFO = 2
```

The "type" associated with each surface is listed at the top of each dataset of surface reactions. The "type" of the hydrous ferric oxide surface represented by the dataset "FeOH.sdat", for example, is "HFO". You can use the "show" command to display the "type" of each active surface.

C.2.96 surface_data

surface_data <sorption dataset>
surface_data remove <sorption dataset | surface type>
surface_data OFF

Use the "surface_data" command (abbrev.: "surf_data") to specify an input dataset of surface sorption reactions to be considered in the calculation. The dataset name should be enclosed in quotes if it contains any unusual characters. Use the "remove" argument to eliminate a surface dataset, specified by name or surface type (e.g., "HFO"), from consideration. The argument "OFF" disables consideration of all surface complexes.

You can specify more than one sorbing surface in a model by repeating the "surface_data" command for different datasets (a dataset of surface reactions for sorption onto hydrous ferric oxide, several triple-layer and CD-MUSIC datasets, as well as example datasets for the ion exchange, K_d, Freundlich, and Langmuir models are distributed with the software). To remove a dataset of surface reactions from consideration, you use commands such as

```
surface_data remove FeOH.sdat
surface_data remove HFO
surface_data OFF
```

The latter command removes all of the surface datasets that have been loaded.

C.2.97 surface_potential

surface_potential = <value | **?**>
surface_potential on <type> = <value | **?**>

Use this command (abbrev.: "surf_potential") to set, in units of mV, the electrical potential for a two-layer sorbing surface. When you set this value (or if a value is set in the header section of the surface reaction dataset), **ChemPlugin** will model surface complexation for the surface in question using the constant potential method, rather than invoking the full two-layer model.

If you have set more than one sorbing surface (using the "surface_data" command), you identify the surface in question by its "type". For example,

```
surface_potential on HFO = 0
```

The "type" associated with each surface is listed at the top of each dataset of surface reactions. The "type" of the hydrous ferric oxide surface represented by the dataset "FeOH.sdat", for example, is "HFO". You can use the "show" command to display the "type" of each active surface.

C.2.98 swap

swap <new basis> <**for**> <basis species>

Use the "swap" command to change the set of basis entries. All reactions are written internally in terms of a set of basis species that you can alter to constrain the composition of the initial system. An aqueous species, mineral, gas, or activity ratio can be swapped into the basis in place of one of the original basis species listed in the database. Examples:

```
swap CO3--          for HCO3-
swap Quartz         for SiO2(aq)
swap CO2(g)         for H+
swap O2(g)          for O2(aq)
swap Ca++/Na+^2     for Ca++
```

Each end member of a continuous solid solution can similarly be swapped into the basis. Use the syntax <solid solution>::<end member tag>. For example, the sodium end member Albite (tag "Ab") of the plagioclase feldspar solid solution (name "Plag") can be swapped into the basis simultaneously with the calcium end member Anorthite (tag "An") as follows:

```
swap Plag::Ab for Na+
swap Plag::An for Al+++
```

The new species must contain in its composition the original basis species being swapped out (you can't swap lead for gold). For example, CO_2(g) is composed of HCO_3^-, H^+, and water in "thermo.tdat". The reactions in the thermo dataset (once reduced to the set of basis and redox species and modified to reflect enabled redox couples) show the basis entries for which a species may be swapped. For a list of original basis species, type "show basis". To reverse a swap, type "unswap <species>".

C.2.99 TDS

TDS = <value | **?**>

Use the "TDS" command to set in mg/kg the total dissolved solids for the initial fluid, if you don't want the program to calculate this value automatically. The program uses the TDS when needed to convert input constraints into molal units.

To restore automatic calculation of the TDS, type the command with no argument or with an argument of "?". To see the variable's current setting, type "show variables".

C.2.100 temperature

temperature = <value> <unit>
temperature initial = <value> <unit> **final =** <value> <unit>
temperature initial = <value> <unit> **reactants =** <value> <unit>
temperature constant = <**on** | **off**>
temperature reset

Use the “temperature” command (also: “T”) to set the temperature of the system or reactants. The **Units Recognized** appendix lists possible units, which default to “C”. Examples:

```
temperature 25 C
T initial = 25 C, final = 300 C
T initial = 398 K, reactants = 498 K
```

Temperature values can range over the span of the thermo dataset, from 0°C to 300°C for “thermo.tdat”; 25°C is the default.

The “constant” keyword lets you hold temperature constant over the course of the calculation at the initial value, regardless of other settings; “on” enables the feature, as does omitting an argument; “off” disables it. The “reset” keyword restores temperature settings to their default states.

C.2.101 theta

theta = <value | **?**>

Use the “theta” command to set the time weighting variable used in evaluating kinetic rate laws. The value may vary from zero (full weighting at the old time level) to one (full weighting at the new time level). By default, this variable is set to 0.6. To restore the default value, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.102 timax

timax = <value | **?**>

The “timax” command sets in molal units the maximum value of ionic strength used in calculating species’ activity coefficients when Debye-Hückel methods are employed. The limiting value is also used when water activity is calculated according to Wolery (1992b). By default, this variable is set to 3 molal. To restore the default value, type the command with no argument or with an argument of “?”. To see the current setting of this variable, type “show variables”.

C.2.103 time

time <**start =** <value> <unit>> <**end =** <value> <unit>>
time = off

Use the "time" command (also: "t") to set the time span of a kinetic reaction path. The starting time, by default, is zero, and the default end time is 1 day. Unit choices are listed in the **Units Recognized** appendix; "day" is the default. Argument "off" switches the program out of kinetic mode. Examples:

```
time end = 100 years
time start 10 days, end 20 days
```

C.2.104 title

title <character string>

Use the "title" command to set a title to be passed to the plot-format output file. Example:

```
title "Yucca Mountain Groundwater"
```

Be sure to put multiword titles in quotes.

C.2.105 unalter

unalter <species | mineral | solid solution | gas | surface species | **ALL**>

Use the "unalter" command to reverse the effect of having changed the log K's for a species, mineral, gas, or surface reaction, the selectivity coefficient for an exchange species, the K_d for a sorbed species, the K_f and n_f for a Freundlich species, or the properties of a solid solution in the current dataset. Example:

```
unalter Quartz
```

In this case, the log K values for quartz revert to those in the current thermodynamic dataset. The argument "ALL" resets the log K's, selectivity coefficients, K_d's, or K_f's and n_f's for all species, minerals, gases, and surface species, as well as the properties of all solid solutions in the dataset.

C.2.106 unsegregate

unsegregate <mineral(s) | **ALL**>

Use the "unsegregate" command to remove minerals from the list of minerals to be segregated isotopically.

C.2.107 unsuppress

unsuppress <species, minerals, solid solutions, gases, surface species | **ALL**>

Use the "unsuppress" command (also: "include") to include in the calculation aqueous species, surface species, minerals, solid solutions, or gases that have previously been suppressed. Examples:

```
unsuppress Quartz Albite "Albite low"
unsuppress ALL
```

The argument "ALL" clears any species, minerals, solid solutions, or gases that have been suppressed.

C.2.108 unswap

unswap <species | **ALL**>

Use the "unswap" command to reverse a basis swap. Example:

```
unswap Quartz (or unswap SiO2(aq))
```

to reverse the effect of the previous command

```
swap Quartz for SiO2(aq)
```

At this point, SiO_2(aq) is back in the basis. The "ALL" argument reverses all basis swaps.

C.2.109 volume

volume = <value | **?**> <unit>

Use the "volume" (also "bulk_volume") command to cause the program to scale the initial system to a specific size. By default, the program does not scale the system. In this case, the bulk volume V_b

$$V_b = V_f + V_{mn} + V_{in} + V_{stag} \tag{19.1}$$

is computed as the sum of the fluid volume V_f, mineral volume V_{mn}, inert volume V_{in}, and the volume $V_{stag}$of the stagnant zone, if one has been specified using the "dual_porosity" command.

If X_{stag} is the stagnant fraction of the system, which is zero if a dual porosity model has not been invoked, the above equation can be rewritten

$$\left(1 - X_{stag}\right) V_b = V_f + V_{mn} + V_{in} \tag{19.2}$$

When you use the "volume" command to set the bulk volume V_b, the program scales the fluid volume V_f and optionally the mineral and inert volumes, V_{mn} and V_{in}, so the system fills the specified space.

The command

```
volume = 2 m3
```

for example, causes the program to scale the fluid volume V_f and optionally the mineral and inert volumes, V_{mn} and V_{in}, so the system occupies 2 m^3. Default behavior, in which no scaling is performed, can be restored at any time with the command

```
volume = ?
```

The unit you use to constrain the amount of each mineral in the system controls whether or not the program adjusts that mineral's volume. If you constrain Quartz to "10 cm3", for example, Quartz will occupy 10 cm^3 after scaling. Setting Quartz in relative units such "volume%" or "mmol/m3", in contrast, causes the program to fix the mineral's volume relative to V_b.

Similarly if you set inert volume in an absolute unit, such as

```
inert = 50 cm3
```

the program will honor that constraint and hold V_{in} steady. The inert portion of the system in this example will occupy 50 cm^3 after scaling. The command

```
inert = 10 volume%
```

however, causes the program to scale the inert volume to 10% of the system's bulk volume.

C.2.110 Xstable

Xstable = <value>

Use the "Xstable" command to control how the stability criterion for dispersive transport and thermal conduction is applied. A value of one sets the theoretically largest stable time step for an ideal situation. ChemPlugin simulations are not necessarily ideal (for example, the solute may react and the medium may not be uniform), so this limiting time step may in fact be too large to be stable. Setting "Xstable" to a value smaller than one results in a more stringent constraint on the time step, and hence greater stability. The default value for this variable is 1.0, the theoretical limit. See also the "Courant" command.

Appendix: Report Function

Tables in this Appendix list the keywords recognized by the “Report()” member function and, for each keyword, any arguments recognized, a description of the keyword’s meaning, the units by which values are returned by default, and the type of value returned.

Keyword	Arguments	Description
activity	<**aqueous** | **surf_species** | **end_members**> <name(s) | index> . . .	Species, end member activities
alkalinity		Carbonate alkalinity
aqueous	<index> . . .	Names of aqueous species
basis	<**original** | **current**> <index> . . .	Names of basis entries
biomass	<reactant(s) | index> . . .	Biomass concentration
boltzman	<surf_species | index> . . .	Boltzman factors for surface species
bulk_volume		Bulk volume of nodal block
cat_area	<reactant(s) | index> . . .	Areas of catalyzing surfaces
charge	<**original** | **current** | **aqueous** | **surf_species**><name(s) | index> . . .	Charge on components or species
chlorinity		Chlorinity
colloids	<index> . . .	Names of mobile colloids
concentration	**original** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **current** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **elements** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **aqueous** **surf_species** **minerals** <**equilibrium** | **kinetic** | **both** | **all**> **end_members** <**equilibrium** | **kinetic** | **both** | **all**> <name(s) | index> . . .	Concentration of components, aqueous or surface species, minerals, end members, or elements
configuration	<**basis** | **swap** | **type** | **unit** | **scale** | **as**> <names(s) | index> . . .	Basis configuration, including entry, swap species, constraint type, unit, scale, and "as" unit
constraints	<name(s) | index> . . .	Values constraining each basis entry
contact_area	<reactant(s) | index> . . .	Contact areas for kinetic gases
couples	<index> . . .	Names of redox couples

Default units	Return
	double
mg/kg sol'n as $CaCO_3$	double
	strings
	strings
mg/kg	double
	double
cm^3	double
cm^2	double
	double
molal	double
	strings
molal	double
	strings
	double
cm^2	double
	strings

Keyword	Arguments	Description
database	<**elements** | **basis** | **redox** | **aqueous** | **electron** | **minerals** | **end_members** | **solid_solutions** | **gas** | **oxide**>	Names of entries in the thermo dataset, whether included in the current simulation or not
Deltat		Length of current time step
EC		Electrical conductivity
Eh	<**system** | **couples**> <name(s) | index> . . .	The system Eh or Nernst Eh values for redox couples
elements	<index> . . .	Names of elements
end_members	<index> . . .	End member names
exchange_capacity	<surface_type(s) | index> . . .	Capacity of ion exchange surface
freeflowing		Volume of free-flowing zone in nodal block
FA	<reactant(s) | index> . . .	Kinetic factor for electon acceptance by microbes
FD	<reactant(s) | index> . . .	Kinetic factor for electon donation by microbes
fugacity	<gas(es) | index> . . .	Gas fugacities
gamma	<**aqueous** | **surf_species** | **end_members**> <name(s) | index> . . .	Species, end member activity coefficients
gas_pressure	<gas(es) | index> . . .	Gas partial pressures
gases	<index> . . .	Names of gases
hardness		Hardness
hardness_carb		Carbonate
hardness_ncarb		Non-carbonate
imbalance		Charge imbalance
imbalance_error		Error percentage
inert_volume		Inert volume in system
IS *or* **Tionst**		System ionic strength
isotopes	<**symbols**>	Names of or symbols for isotope systems

Default units	Return
	strings
s	double
μS/cm or umho/cm	double
volts	double
	strings
	strings
eq	double
cm^3	double
	double
	double
	double
	double
bar	double
	strings
mg/kg sol'n as $CaCO_3$	double
mg/kg sol'n as $CaCO_3$	double
mg/kg sol'n as $CaCO_3$	double
eq/kg	double
% error	double
cm^3	double
molal	double
	strings

Keyword	Arguments	Description
iterations		Iterations required for Newton-Raphson to converge
Kd	<name(s) | index> . . .	Net K_d for sorption of original basis entries onto all surfaces
logfO2		Log fugacity of O_2
logQoverK *or* **SI**	<**minerals** | **solid_solutions** | **reactants**> <name(s) | index> . . .	Saturation index for minerals, solid solutions, or reactants
mass	**original** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **current** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **elements** <**fluid** | **system** | **rock** | **sorbed** | **stagnant** | **colloid**> **aqueous** **surf_species** **minerals** <**equilibrium** | **kinetic** | **both** | **all**> **end_members** <**equilibrium** | **kinetic** | **both** | **all**> <name(s) | index> . . .	Mass of components, aqueous or surface species, minerals, end members, or elements
mass_reacted	<reactant(s) | index> . . .	Mass of a reactant that has reacted
mass_remaining	<reactant(s) | index> . . .	Mass of a reactant remaining to react
minerals	<**equilibrium** | **kinetic** | **both** | **all**> <index> . . .	Names of minerals
mineral_mass		Mass of minerals in system or block
mineral_volume		Volume of minerals in system or block (excludes inert volume)
mixing_fraction		Mixing fraction in flash model
mobility	<surface_type(s) | index> . . .	Mobility of colloidal surfaces
mv	<mineral(s) | index> . . .	Mineral molar volume
mw	<**original** | **current** | **aqueous** | **surf_species** | **elements** | **minerals** | **end_members** | **gases**> <name(s) | index> . . .	Mole weight of components, species, or elements

Default units	Return
	int
liter/kg	double
log fugacity	double
log Q/K	double
moles	double
moles	double
moles	double
	strings
kg	double
cm^3	double
	double
	double
cm^3/mol	double
g/mol	double

Keyword	Arguments	Description
naqueous		Number of aqueous species
nbasis		Number of basis entries
ncolloids		Number of mobile colloids
ncouples		Number of redox couples
nelements		Number of elements
nend_members		Number of end members
ngases		Number of gases
nisotopes		Number of isotope systems
nminerals	<**equilibrium** | **kinetic** | **both** | **all**>	Number of minerals
nreactants	<**simple** | **fixed** | **sliding** | **kin_mineral** | **kin_endmember** | **kin_redox** | **microbial** | **kin_aqueous** | **kin_surface** | **kin_gas** | **all**>	Number of reactants, kinetic reactions
nsolid_solutions		Number of solid solutions
nsorbed		Number of original basis species that sorb
nsorbing_surfaces		Number of sorbing surface types
nstagnant		One for dual porosity, else zero
nsurf_species		Number of surface species
options		List of keywords for the report command
pe	<**system** | **couples**> <name(s) | index> . . .	The system pe or theoretical pe for redox couples
permeability		Sediment permeability
pH		System pH
porosity		Porosity
pressure		Pressure
PV		Pore volumes displaced
QoverK	<**minerals** | **solid_solutions** | **reactants**> <name(s) | index> . . .	*Q/K* for a mineral, soild solution, or reactant

Default units	Return
	int
	int
	int
	int
	int
	int
	int
	int
	int
	int
	int
	int
	int
	strings
	double
darcy $\approx \mu m^2$	double
	double
volume fraction	double
bars	double
	double
	double

Keyword	Arguments	Description
rate_con	<reactant(s) | index> . . .	Rate constants for kinetic reactions
ratecon_unit	<reactant(s) | index> . . .	Units of kinetic rate constants
reactant_area	<reactant(s) | index> . . .	Surface areas of kinetic minerals
reactant_type	<reactant(s) | index> . . .	`simple`, `fixed`, `kin_mineral`, `microbial`, and so on
reactants	<**simple** | **fixed** | **sliding** | **kin_mineral** | **kin_endmember** | **kin_redox** | **microbial** | **kin_aqueous** | **kin_surface** | **kin_gas** | **all**> <index> . . .	Names of reactants and kinetic reactions
rxn_rate	<reactant(s) | index> . . .	Reaction rates
SIS *or* **Sionst**		Stoichiometric ionic strength
solid_solutions	<index> . . .	Names of solid solutions
soln_compressibility		Solution compressibility
soln_density		Solution density
soln_expansivity		Solution thermal expansivity
soln_mass		Solution mass
soln_viscosity		Viscosity of fluid
soln_volume		Volume of fluid
sorb_area	<surface_type(s) | index> . . .	Areas of sorbing surfaces
sorbed	<index> . . .	Names of original basis species that sorb
sorption_capacity	<surface_type(s) | index> . . .	Capacity of a Langmuir surface
stagnant		Volume of stagnant zone in nodal block
success		Returns a value of one if ChemPlugin has successfully completed a time step, zero if not
surf_charge *or* **surf_charge0**	<surface_type(s) | index> . . .	Electrical charge at 0-plane of a sorbing surface
surf_chargeb	<surface_type(s) | index> . . .	Electrical charge at β-plane
surf_charged	<surface_type(s) | index> . . .	Electrical charge at d-plane

Default units	Return
(varies)	double
	strings
cm^2	double
	strings
	strings
mol/s	double
molal	double
	strings
bar^{-1}	double
g/cm^3	double
$°C^{-1}$	double
kg	double
cp	double
cm^3	double
cm^2	double
	strings
mol	double
cm^3	double
	int
$\mu C/cm^2$	double
$\mu C/cm^2$	double
$\mu C/cm^2$	double

Keyword	Arguments	Description
surf_potential *or* **surf_potential0**	<surface_type(s) | index> . . .	Electrical potential at 0-plane of a sorbing surface
surf_potentialb	<surface_type(s) | index> . . .	Electrical potential at β-plane
surf_potentiald	<surface_type(s) | index> . . .	Electrical potential at d-plane
surf_species	<index> . . .	Names of surface species
surf_type		Types of reacting surfaces
surfaces		Names of reacting surfaces
TDS		Total dissolved solids
temperature *or* **T**		Temperature
Tend		Final time of simulation
Time		Current point in time
total_biomass		Biomass in system or block
total_reacted		Mass reacted into system or block
TPF	<reactant(s) | index> . . .	Thermodynamic potential factor
Tstart		Beginning time of simulation
Watact		Activity of water
watertype		Ion type of water
Wmass		Water mass
Xfree		Free-flowing fraction
Xi		Reaction progress
xsorbed	<name(s) | index> . . .	Sorbed fraction of an original basis entry
xss	<**equilibrium** | **kinetic** | **both** | **all**> <name(s) | index> . . .	End member mole fractions
<isotope | **Hydrogen-2** | **Carbon-13** | **Oxygen-18** | **Sulfur-34** | symbol | **2-H** | **13-C** | **18-O** | **34-S**>	<**fluid** | **rock** | **sorbate** | **system**> <**solvent** | **aqueous** | **minerals** | **end_members** | **gases** | **surf_species** | **reactants**> <name(s) | index> . . .	Isotopic compositions of various aspects of system

Default units	Return
mV	double
mV	double
mV	double
	strings
	strings
	strings
mg/kg	double
°C	double
s	double
s	double
mg/kg	double
g	double
	double
s	double
	double
	string
kg	double
	double
	double
	double
	double
δ (‰)	double

Appendix: Units Recognized

The following is a complete table of the unit names recognized by ChemPlugin. The qualifier “free” specifies that the constraint applies to the free rather than to the bulk entry. Use the “log” qualifier to set the variable on a logarithmic scale. Examples:

```
Cl- 4.1 mg/kg
Cl- 4.1 free mg/kg
Cl- 0.612784 log free mg/kg
```

Dimension	Units			
Mass and Concentration	mol	mmol	umol	nmol
	molal	mmolal	umolal	nmolal
	mol/kg	mmol/kg	umol/kg	nmol/kg
	mol/l	mmol/l	umol/l	nmol/l
	kg	g	mg	ug
	ng			
	g/kg	mg/kg	ug/kg	ng/kg
	wt fraction	wt%		
	g/l	mg/l	ug/l	ng/l
	eq	meq	ueq	neq
	eq/kg	meq/kg	ueq/kg	neq/kg
	eq/l	meq/l	ueq/l	neq/l
	cm3	m3	km3	l
	mol/cm3	mmol/cm3	umol/cm3	nmol/cm3
	kg/cm3	g/cm3	mg/cm3	ug/cm3
	ng/cm3			
	mol/m3	mmol/m3	umol/m3	nmol/m3
	kg/m3	g/m3	mg/m3	ug/m3
	ng/m3			
	vol. fract.	volume%		

Dimension	Units			
Activity	activity	ratio		
Fugacity	fugacity			
Electrical Potential (Eh)	V	mV	pe	
pH	pH			
Percentage	%			
Time	s	min	hr	day
	mon	yr	m.y.	
Reaction Rate	mol/s	mmol/s	umol/s	nmol/s
	kg/s	g/s	mg/s	ug/s
	ng/s			
	cm3/s	m3/s	l/s	gal/s
	ft3/s			
	mol/min	mmol/min	umol/min	nmol/min
	kg/min	g/min	mg/min	ug/min
	ng/min			
	cm3/min	m3/min	l/min	gal/min
	ft3/min			
	mol/hr	mmol/hr	umol/hr	nmol/hr
	kg/hr	g/hr	mg/hr	ug/hr
	ng/hr			
	cm3/hr	m3/hr	l/hr	gal/hr
	ft3/hr			
	mol/day	mmol/day	umol/day	nmol/day
	kg/day	g/day	mg/day	ug/day
	ng/day			
	cm3/day	m3/day	l/day	gal/day
	ft3/day			
	mol/yr	mmol/yr	umol/yr	nmol/yr
	kg/yr	g/yr	mg/yr	ug/yr
	ng/yr			
	cm3/yr	m3/yr	l/yr	gal/yr
	ft3/yr			

Dimension	Units			
Reaction Rate	mol/m.y.	mmol/m.y.	umol/m.y.	nmol/m.y.
	kg/m.y.	g/m.y.	mg/m.y.	ug/m.y.
	ng/m.y.			
	cm3/m.y.	m3/m.y.	l/m.y.	gal/m.y.
	ft3/m.y.			
	mol/cm3/s	mmol/cm3/s	umol/cm3/s	nmol/cm3/s
	kg/cm3/s	g/cm3/s	mg/cm3/s	ug/cm3/s
	ng/cm3/s			
	cm3/cm3/s	volume%/s		
	mol/cm3/min	mmol/cm3/min	umol/cm3/min	nmol/cm3/min
	kg/cm3/min	g/cm3/min	mg/cm3/min	ug/cm3/min
	ng/cm3/min			
	cm3/cm3/min	volume%/min		
	mol/cm3/hr	mmol/cm3/hr	umol/cm3/hr	nmol/cm3/hr
	kg/cm3/hr	g/cm3/hr	mg/cm3/hr	ug/cm3/hr
	ng/cm3/hr			
	cm3/cm3/hr	volume%/hr		
	mol/cm3/day	mmol/cm3/day	umol/cm3/day	nmol/cm3/day
	kg/cm3/day	g/cm3/day	mg/cm3/day	ug/cm3/day
	ng/cm3/day			
	cm3/cm3/day	volume%/day		
	mol/cm3/yr	mmol/cm3/yr	umol/cm3/yr	nmol/cm3/yr
	kg/cm3/yr	g/cm3/yr	mg/cm3/yr	ug/cm3/yr
	ng/cm3/yr			
	cm3/cm3/yr	volume%/yr		
	mol/cm3/m.y.	mmol/cm3/m.y.	umol/cm3/m.y.	nmol/cm3/m.y.
	kg/cm3/m.y.	g/cm3/m.y.	mg/cm3/m.y.	ug/cm3/m.y.
	ng/cm3/m.y.			
	cm3/cm3/m.y.	volume%/m.y.		
	mol/m3/s	mmol/m3/s	umol/m3/s	nmol/m3/s
	kg/m3/s	g/m3/s	mg/m3/s	ug/m3/s
	ng/m3/s			
	m3/m3/s			
	mol/m3/min	mmol/m3/min	umol/m3/min	nmol/m3/min
	kg/m3/min	g/m3/min	mg/m3/min	ug/m3/min
	ng/m3/min			
	m3/m3/min			

Dimension	Units			
Reaction Rate	mol/m3/hr	mmol/m3/hr	umol/m3/hr	nmol/m3/hr
	kg/m3/hr	g/m3/hr	mg/m3/hr	ug/m3/hr
	ng/m3/hr			
	m3/m3/hr			
	mol/m3/day	mmol/m3/day	umol/m3/day	nmol/m3/day
	kg/m3/day	g/m3/day	mg/m3/day	ug/m3/day
	ng/m3/day			
	m3/m3/day			
	mol/m3/yr	mmol/m3/yr	umol/m3/yr	nmol/m3/yr
	kg/m3/yr	g/m3/yr	mg/m3/yr	ug/m3/yr
	ng/m3/yr			
	m3/m3/yr			
	mol/m3/m.y.	mmol/m3/m.y.	umol/m3/m.y.	nmol/m3/m.y.
	kg/m3/m.y.	g/m3/m.y.	mg/m3/m.y.	ug/m3/m.y.
	ng/m3/m.y.			
	m3/m3/m.y.			
	molal/s	mmolal/s	umolal/s	nmolal/s
	mol/kg/s	mmol/kg/s	umol/kg/s	nmol/kg/s
	g/kg/s	mg/kg/s	ug/kg/s	ng/kg/s
	cm3/kg/s			
	molal/min	mmolal/min	umolal/min	nmolal/min
	mol/kg/min	mmol/kg/min	umol/kg/min	nmol/kg/min
	g/kg/min	mg/kg/min	ug/kg/min	ng/kg/min
	cm3/kg/min			
	molal/hr	mmolal/hr	umolal/hr	nmolal/hr
	mol/kg/hr	mmol/kg/hr	umol/kg/hr	nmol/kg/hr
	g/kg/hr	mg/kg/hr	ug/kg/hr	ng/kg/hr
	cm3/kg/hr			
	molal/day	mmolal/day	umolal/day	nmolal/day
	mol/kg/day	mmol/kg/day	umol/kg/day	nmol/kg/day
	g/kg/day	mg/kg/day	ug/kg/day	ng/kg/day
	cm3/kg/day			
	molal/yr	mmolal/yr	umolal/yr	nmolal/yr
	mol/kg/yr	mmol/kg/yr	umol/kg/yr	nmol/kg/yr
	g/kg/yr	mg/kg/yr	ug/kg/yr	ng/kg/yr
	cm3/kg/yr			
	molal/m.y.	mmolal/m.y.	umolal/m.y.	nmolal/m.y.
	mol/kg/m.y.	mmol/kg/m.y.	umol/kg/m.y.	nmol/kg/m.y.
	g/kg/m.y.	mg/kg/m.y.	ug/kg/m.y.	ng/kg/m.y.
	cm3/kg/m.y.			
Flow Rate	cm3/s	m3/s	l/s	gal/s
	ft3/s			
	cm3/min	m3/min	l/min	gal/min
	ft3/min			

Dimension	Units			
Flow Rate	cm3/hr	m3/hr	l/hr	gal/hr
	ft3/hr			
	cm3/day	m3/day	l/day	gal/day
	ft3/day			
	cm3/yr	m3/yr	l/yr	gal/yr
	ft3/yr			
	cm3/m.y.	m3/m.y.	l/m.y.	gal/m.y.
	ft3/m.y.			
Density	kg/cm3	g/cm3	mg/cm3	ug/cm3
	ng/cm3			
	kg/m3	g/m3	mg/m3	ug/m3
	ng/m3			
Titration Alkalinity	eq_acid	meq_acid	ueq_acid	neq_acid
	eq_acid/kg	meq_acid/kg	ueq_acid/kg	neq_acid/kg
	eq_acid/l	meq_acid/l	ueq_acid/l	neq_acid/l
	g/kg_as_CaCO3	mg/kg_as_CaCO3	ug/kg_as_CaCO3	ng/kg_as_CaCO3
	wt%_as_CaCO3			
	g/l_as_CaCO3	mg/l_as_CaCO3	ug/l_as_CaCO3	ng/l_as_CaCO3
	mol/kg_as_CaCO3	mmol/kg_as_Ca...	umol/kg_as_Ca...	nmol/kg_as_CaCO3
	mol/l_as_CaCO3	mmol/l_as_CaCO3	umol/l_as_CaCO3	nmol/l_as_CaCO3

Dimension	Units			
Sorption Capacity	mol/grock	mmol/grock	umol/grock	nmol/grock
Exchange Capacity	eq/grock	meq/grock	ueq/grock	neq/grock
Surface Charge	uC/cm2			
Pore Volumes	pore_volumes			
Dynamic Viscosity	cp	poise		
Compressibility	/Pa /psi	/MPa	/atm	/bar
Thermal expansivity	/C	/F	/K	/R
Pressure	Pa psi	MPa	atm	bar
Permeability	m2 darcy	cm2 mdarcy	um2 udarcy	
Distribution Coefficients (KDs)	l/kg	ml/g	ml/mg	
Activity Coefficients	act. coef.			
Electrical Conductivity	uS/cm	umho/cm		
Heat Capacity	J/g/C	J/kg/K	cal/g/C	
Internal Heat Source	J/cm3/s cal/cm3/s W/cm3	J/cm3/yr cal/cm3/yr W/m3	J/m3/s cal/m3/s	J/m3/yr cal/m3/yr
Thermal Transmissivity	W/C cal/s/C	W/K cal/s/K	J/s/C	J/s/K
Saturation	Q/K			
Temperature	C	F	K	R
Number	number			

Index