BioRT++ Report

Andrew Vierbicher

*Note

I used Doxygen for documentation of the code, and these PDF files are present in the **doc** folder of the source code.

Project Description

My project is an update to the core functionality of the catchment biogeochemical model BioRT-HBV developed in the research group under Dr. Li Li with whom I work. BioRT-HBV is a numerical model that simulates chemical reactions in a watershed. With this model, my research group and I use the numerical model along with in-situ chemical observations to model biotic and abiotic chemical reactions in watersheds to quantitatively understand how elements cycle throughout the natural environment.

In the real, world, this model attempts to determine the concentrations of chemical species when there are equilibrium relationships and kinetic reactions taking place in the environment, as these concentrations have implications for the humans in how we use water.

BioRT-HBV is a simple bucket model coupled to the conceptual hydrologic model HBV[1], yet it has proven to be effective at modeling organic carbon reactions, mineral weathering, and nitrogen cycling all for using much less computational power than spatially explicit models. Furthermore, there are relatively few parameters involved in calibrating this model and the data requirements are minimal, meaning that this model is relatively much easier to calibrate than large, expensive spatially explicit models. Because the model has a simple structure and relatively few parameters, it has the potential to be much more efficient at revealing biogeochemical processes than more complex models.

However, the current version of BioRT-HBV is implemented in C with code that is tightly coupled and difficult to both read and modify. As such, I am undertaking the effort to update this model by rewriting it in C++ because it will allow cleaner code that is much easier to understand, modify, and maintain. For this project, I have rewritten the core numerical components for the model, which are the chemical equilibrium solver and the kinetic reaction solver. These two components are essentially the entire heart of the model and drive all of the chemical dynamics. The governing chemical equations are highly nonlinear and the differential equations are commonly stiff, so this problem requires a thorough consideration of the mathematics that the current version does not have. In addition, I have chosen to use YAML files as the format for the input files because the current version of BioRT-HBV uses a custom file format for each input file, which is difficult to learn to use and debug.

Design Strategy

For this project, I choose to use a functional programming pattern rather than object-oriented because of the nature of the model. At the core of our work in my group, we are solving ordinary differential equations, and an object-oriented programming style does not make sense for this type of problem. I attempted to make all function parameters immutable so that there would be no internal mutation of variables within a function. I choose this because the current version of BioRT-HBV is written in a procedural style in C and it is nearly impossible to debug, and I, as the maintainer of the model now, would like to be able to test the model easily, and immutability of function parameters makes this easier.

Program Walkthrough

BioRT++ is used through both command line parameters and input files. The structure of the input files is described below, but roughly, the chemistry file (chem.yaml) contains the chemical system of interest, including the aqueous species, equilibrium reactions, and mineral species present in the system. The database contains all of the static parameters for every type of species present. In my examples here, the database contains only the species that I use in the programs. In the current version of BioRT, however, the database contains thousands of entries and is not computer readable at present. Translating this database will be very time consuming and not necessary.

To begin, the program is called with the the input folder name, which also contains the run type, begin an equilibrium or kinetic problem. The program will search in the input directory for the input folder of the given name to load the input files. From here, the program splits based on whether it is an equilibrium or kinetic solution.

Equilibrium Solution

Solving for the equilibrium concentrations is a relatively simple numerical optimization problem. It involves solving two sets of equations: mass conservation and equilibrium constants. Mass conservation equations have the format:

$$C_{tot,i} = \sum_{j=1}^{n} a_{ij} C_j$$

Where a_{ij} is the total concentration matrix and $\{C_j\}$ is the set of species concentrations. In general, the total concentration represents how primary species can be used to make up different species like the basis vectors of a vector space. For every primary species, there will be one total species mass balance equation like above, and this problem is, in general, underdetermined, meaning that there must be more equations for this system to have closure.

The other type of equations are the equilibrium equations which have the following general formula:

$$K_{eq_i} = \prod_{j=1}^{n} C_j^{\nu_{ij}}$$

Where ν_{ij} is the matrix of stoichiometric coefficients and $\{C_j\}$ is the set of concentrations. Note that when taking the logarithm, this system can be linearized as:

$$\ln K_{eq_i} = \sum_{j=1}^{n} \nu_{ij} C_j$$

And as such, it is much easier to solve this equations on log-space. Therefore, since the mass conservation takes place in linear space and the equilibrium equations take place in log space, these equations are nonlinear and can only be solved through nonlinear iteration, which is what I implemented in this model.

Kinetic Solution

The kinetic solution is much more computationally expensive, though has simpler mathematics. Kinetic chemical reactions depend on concentrations of each species and independent of time, meaning that this is an autonomous ODE system:

$$\frac{\partial \vec{C}}{\partial t} = f(\vec{C})$$

There are numerous equations to represent the kinetic rate laws, and I choose one based on a Transition-State-Theory type rate law with the form:

$$\frac{dC_i}{dt} = A_j k_j \left(1 - \frac{1}{K_{eq}} \prod_{k=1}^{n} C_{k\,jk}^{\nu}\right)$$

However, I use to solve for the rate of change of each of the total concentrations instead of species, as the concentration depends on the total concentrations: $\vec{C} = \vec{C}(\vec{C}_{tot})$ due to equilibrium relationships. To solve these autonomous ODE, I implement Runge-Kutta methods, both explicit and implicit. In general, these chemical reaction rates can vary by orders of magnitude, meaning that these ODE systems may be stiff, requiring an implicit solver. For this reason, the solver I implemented in my library **ode** first attempts to solve the ODE with the much faster explicit Runge-Kutta method and only uses the much slower, yet stable, implicit method if the explicit method is unstable.

External Libraries

For this model, I use four external open-source libraries:

1. **armadillo**: This library provides linear algebra functionality. It includes common data types like vectors and matrices and common linear algebra functions like solving linear system, determining null space, and eigenvalue decomposition. Furthermore, these functions can be used in a functional style like the Python's **numpy.** This library is essential for all aspects of numerical computation in this project.
2. **Catch2:** This is the unit-testing framework that I use to test parts of my code in isolation. This library integrates very well with cmake to make writing and running tests very easy.

3. **matplotplusplus**: This library provides the plotting functionality to plot the kinetic simulation results. The interface for this library is nearly identical to Matlab of Matplotlib's plotting interface, so making a line plot is very easy.
4. **Yaml-cpp**: This library provides functions and datatypes to easily work with YAML files. With this library, I can deal with a YAML file as a code object rather than parsing a string object with Regex or any other type. This library is beneficial for the model because it removes the responsibility of us in our group working on the model to write code that parses the input files, and rather places it on the library that has been extensively tested for a very popular file format.

My Own Libraries

ode

I implemented my own ordinary differential equation library with just two solvers: the 4^{th} order explicit Runge-Kutta method and the 4^{th} order implicit Runge-Kutta-Gauss method for stiff systems. The documentation for this library is present in the **docs** folder of the source code. I chose these two methods so that the accuracy would be very good while also having a relatively good performance.

optimization

I implemented a very simple numerical optimization library with functions to compute the numerical Jacobian matrix and determine the root of a vector function using Newton-Raphson iteration. This function is templated in case I wish to use a different level of precision later on when adding this library to the main version of BioRT-HBV

Justifications

- **Compilable:** I have used Cmake as the build system for this project and included all libraries from source so that the model could be built on Windows and Mac in addition to Linux, which I developed it in.
- **Runtime Errors**: I included the Catch2 library to run unit test my code to catch errors on isolated code.
- **Readability**: To the best of my ability, I have used lambda functions to reduce the complexity of the code and reduce the number of parameters to think about. I used libraries like armadillo that use operator overloading on their containers that abstract many of the operations so that I can express the mathematical relationships as simply as possible.
- **Math/Technical knowledge**: This program includes numerous mathematical concepts including numerical solution of ordinary differential equations, stiff differential equations, numerical optimization, and linear algebra. It also includes concepts from chemistry including thermodynamic equilibrium and kinetic reactions.
- **Code organization**: I have organized my code into header and implementation files with each file containing the functionality of a single part of the code, namely the command line handling, reading the input files, the equilibrium solution, the kinetic solutions, and utility functions.
- **Efficiency**: Running on my computer, the model is able to run a simulation of ~1000 time steps for a fairly simple kinetic reactions in ~0.71 seconds.
- **Memory leaks**: I use memory safe containers and iterators where possible to reduce the possibility of memory leaks. While writing the code, I did have one segmentation fault, though I determined the cause of this using the GNU Debugger and fixed the problem so that it could not be a problem layer on.
- **Memory usage**: I use containers that manage their memory like vectors and maps and never raw pointers or C-style arrays so that these containers will be de-allocated automatically when they go out of scope and so that I do not have to deal with it.
- **CPU usage**: This program is single- threaded by default, and it uses only a single CPU core for all its runtime. This is a numerical model, and because of that, it uses 100% of the CPU core for that runtime. This is by design, because once this core functionality in my project is implemented into the new version of BioRT-HBV, much time will be spent on calibrating model parameters, which involves running the model for hundreds of thousands to millions of times, so being able to run efficiently ona single core is a necessity.
- **Complexity**: In this project, I have implemented the governing equations to chemical equilibrium and kinetic reactions in a new way compared to the current version of BioRT-HBV, and so I was on my own to implement these numerical methods and represent the governing equations.

- **Operators**: As this is numerical model, and computer math is limited to the 4 operations +, -, *, and /, I have used all extensively within all components of the model.
- **Conditional statements**: In the main function, I use a conditional statement to determine whether to run an equilibrium simulation or a kinetic simulation.
- **Iterative statements**: In the **optimization** library, I use Newton-Raphson iteration in a while loop to determine the root of a function, in addition to many other locations throughout the codebase.
- **Functions**: I use functions everywhere throughout my code, and the two most important functions are **solve_equilibrium** and **solve_kinetic_equilibrium**, which are the two core concepts that my model aims to simulate.
- **Preprocessor**: In each of my header files, I use **#pragma once** so that I do not have to ues include guards in the code.
- **References**: In all cases where I pass a vector to a function, I pass a vector by reference so that there is no memory copy when passing to the function.
- **Lambda functions**: I use lambda functions frequently in the numeric parts of my code. On example that is used frequently is to take the **jacobian** function that requires multiple parameters and turn it into just a function of a single variable to be used in numerical optimization.
- **Template functions**: I use templated functions in the **optimization** and **ode** libraries so that I could reuse those libraries in later versions of the model if I wanted a different precision.
- **Containers**: I use both vectors and maps extensively that come from the standard library. I use a vector as a container for all of the chemical species names and a map to store the indices of each species in the concentration vectors.
- **Iterators**: I use iterators when reading the input files in the **ModelInputs** class. When reading the YAML maps and arrays, I use the YAML iterators to parse each element in the map/node to make the code cleaner than if I had used a for loop.
- **Struct/Class**: I use both structs and classes througout my code. One important data structure is the **ChemicalState**, which is a simple struct to contain the concentration and total concentrations of each species.
- **Objects**: I instantiate my classes and structs often, namely that the ModelInputs is instantiated to contain the input data, and a ChemicalState object is created each time **solve_equilibrium** or **solve_kinetic_equilibrium** is called.
- **GUI**: This project uses the **matplotplusplus** library to plot the results, which contains a GUI that the user can manipulate to save plots of kinetic simulations.
- **Date and Time**: I use the **chrono** and **ctime** libraries to get the current date and time to create a time stamp for each model simulation run. This is used in the **get_timestamp** function.

- **File system handling**: My project reads files from the filesystem and also creates output files containing the results of a simulation.

Citations

1. Seibert, Jan. "HBV light." *User's manual, Uppsala University, Institute of Earth Science, Department of Hydrology, Uppsala* (1996).