Overview of the CHILD Source-Code Structure

Greg Tucker

June 2008 (Last updated August 2009)

Contents

1 Introduction

1.1 About CHILD

CHILD is a numerical landscape evolution model that was originally developed at MIT in the mid to late 1990s by members of Rafael Bras' geomorphology group (Rafael Bras, Nicole Gasparini, Stephen Lancaster, and Greg Tucker). Since then, the code has continued to grow, with new capabilities and contributions from researchers around the world. The main version of the code is currently maintained in a networked source-code repository at the University of Colorado. A release version is available through the Community Surface Dynamics Modeling System (CSDMS): http://csdms.colorado.edu.

1.2 Source-Code Structure

The CHILD (version 2008) source code is divided among a number of different source files. These source files represent, more or less, the various modules and utility classes that together make up CHILD. This document gives a brief overview of the module and file structure. It is intended for developers who wish to add capabilities, create linking software, or couple CHILD with other models. I assume you are basically familiar with C++. File names and file sets are given in **bold** and class names are given in *italics*. By "file set" I mean a group of related files that reside in the same folder (for example, **tUplift.h** and **tUplift.cpp**).

The code consists of several different file sets, most of which reside in their own sub-folders and usually consist of a header (.h) and source (.cpp) file. Sometimes there will be only a header file, and sometimes more than one header and/or source file.

The high-level structure, including initialization and the implementation of a time loop, is handled by "Child Interface" code. This provides a simple interface through which CHILD can be invoked by another application, with methods that initialize the model, run one storm, run for a specified amount of time, and finalize. The main driver is found in **childDriver.cpp**, which calls the Initialize, Run, and CleanUp functions in the interface. The interface routines for the development version are found in childInterface; for the release version, the corresponding code lives in the childRInterface folder.

Most of the main modules, which handle processes such as water routing, erosion, and tectonic uplift, are implemented as classes (such as the tStreamNet, tErosion, and tUplift classes). (The file **toddlermain.cpp** implemented a now-obsolete release version).

2 General Header Files: Classes, compiler, Definitions, Inclusions, Template_Model, and trapfpe

The header files handle various things. **Classes.h** tells the compiler what classes to expect. **Template_model.h** and **compiler.h** are meant to handle differences between compilers (they aren't foolproof). **Definitions.h** provides a list of defined constants and macros. **trapfpe.h**, which I think was written by Arnaud Desitter, enables floating-point-exception traps in Linux.

3 Basic Utilities

3.1 Error Handling: errors

The small **errors** file set takes care of reporting errors and warnings.

3.2 Command-Line Options: tOption

The **tOption** file set parses and handles command-line options.

3.3 Mathematics: Geometry, Mathutil, and Predicates

The **mathutil** file set handles random-number generation through the *tRand* class. Thanks to Arnaud Desitter, it includes an implementation of a Mersenne Twister pseudorandom number generator. The **geometry.h** file defines classes for 2D and 3D points with constructors and assignment operators. The **Predicates** file set handles arbitrary precision floating-point arithmetic, using a subset of a public domain code modified by Stephen Lancaster.

3.4 Vectors and Matrices: tArray and tMatrix

The first version of CHILD was written before the Standard Template Library was created, so it uses its own array (vector) handling routines in the **tArray** file set. This implements the *tArray* class, which handles 1D arrays with bounds-checking, constructors, etc. It is widely used throughout the code. The **tMatrix** file set implements 2D arrays, or matrices, using the *tMatrix* class. It is not used especially frequently in the code, however.

3.5 Linked List Management: tList, tPtrList, and tMeshList

CHILD makes heavy use of linked lists, which come in several flavors. Basic two-way linked lists of objects are handled by the **tList** file set. It includes several classes: *tList* implements the list itself, *tListNodeBasic*, *tListable* and *tListNodeListable* implement the nodes that make up the list, and *tListIter* implements iterators that move up and down a list to access its nodes. Lists of pointers to objects, as opposed to lists of the objects themselves, have to be handled a little differently, and are implemented by the **tPtrList** file set.

In addition to these, **tMeshList** handles a derived type of list designed specifically for handling lists of mesh elements (nodes and edges; triangles don't need special handling). A *tMeshList* is a *tList* that is divided into two sections: an "active" section (at the "top" of the list) representing nodes and edges that fall inside the mesh boundaries. In other words, nodes that are part of the computed domain and whose elevations change, etc., are placed in the *active* portion of the list. Boundary nodes (whether open or closed to water and sediment) are placed in the *inactive* ("bottom") portion of the list. This makes it easy to access a list of active nodes. Edges are handled in a similar fashion. The *tMeshListIter* class provides iterators to access a *tMeshList*.

3.6 Time Management: tRunTimer

tRunTimer is a small class that keeps track of the current time in the simulation.

4 Mesh Construction and Management: tMesh, tLNode and MeshElements

CHILD includes a large and fairly complex body of code to generate, maintain, and update the Delaunay triangulation and related Voronoi diagram. The *tMesh* class manages the overall mesh, and includes lists of three basic objects that make up the mesh: nodes (*tNode* and *tLNode* classes), directed edges (*tEdge* class), and triangles (*tTriangle* class). The *tNode*, *tEdge*, and *tTriangle* classes are defined in the **MeshElements** fileset. The *tNode* class is designed to be generic and adaptable; its data include 3D coordinates and a pointer to one of the edges ("spokes") connected to it, but little else. The *tLNode* class, which is inherited from *tNode*, contains most of the data related to landscape evolution (drainage area, shear stress, etc.). Data structures are described in Tucker et al. (2001 *Computers and Geosciences*).

5 Parameter and Data Input: tInput and tTimeSeries

Input of parameters and options is handled by the *tInputFile* class, which opens and reads a specified file, and delivers the necessary parameters to the parts of the code that request them. The class is generic and self-contained, and can easily be used by other codes (I have used it so). It implements a simple "two-line" format in which each parameter is represented by a unique tag on one line of the input file and the value is specified on the next line. The order of parameters is arbitrary, and comments can be included in the input file by placing a hash mark at the beginning of the line.

Initial topography data, if desired, is not handled by **tInputFile** but rather by the mesh routines (in the **tMesh** fileset). Some parameters can vary through time, and the time variation is handled using the *tTimeSeries* class written by Patrick Bogaart (currently at Wageningen University, Netherlands).

6 Output: tOutput, tLOutput

Output of data is handled by two related classes (and some helpers): *tOutput* and *tLOutput*. The *tOutput* class handles output of triangulated mesh data to files. The class handles only output of mesh data (nodes, edges, and triangles); output of additional data (e.g., water or sediment flow) is handled by the derived class *tLOutput*.

tOutput provides functions to open and initialize output files and write output at a specified time in a simulation. The class is templated in order to allow for a pointer to a templated *tMesh* object.

To handle output of application-specific data, one can create a class inherited from *tOutput* and overload its virtual WriteNodeData function to output the additional data. In the present version of CHILD, the files **tOutput.h/.cpp** contain the inherited class *tLOutput*, which handles output for the CHILD model. In the future, such inherited classes could be kept in separate files to preserve the generality of *tOutput*.

7 Rainfall and Runoff: tStorm and tStreamNet

The **tStorm** module generates storm events and interstorm periods using pseudo-random numbers (or with constant intensity and duration, if the user desires). The **tStreamNet** turns this into stream flow by computing drainage directions, infiltration, and discharge. It also computes channel geometry and sorts the node list according to network position.

The stochastic rainfall model is discussed in Tucker and Bras (2000 *Water Resources Research*) and Tucker (2004 *Earth Surface Processes and Landforms*). Runoff and drainage routing algorithms are described in Tucker et al. (2001a *Computers and Geosciences* and 2001b Chapter in *Landscape Erosion and Evolution Modeling* by R. Harmon and W. Doe).

8 Erosion and Transport: Erosion

The **Erosion** module is the geomorphic workhorse. It implements various options for sediment transport capacity and "bedrock" detachment. It includes the DetachErode function (and its counterpart, DetachErode2, written by Nicole Gasparini for the sediment-flux-dependent bedrock erosion laws). This module also includes hillslope transport. The erosion and transport algorithms are discussed in Tucker et al. (2001b), Gasparini et al. (2004 *Earth Surface Processes and Landforms*), and Gasparini et al. (2006 in *Tectonics, climate and landscape evolution*). As of this writing, DetachErode2 isn't fully integrated; one has to hack **childmain.cpp** to make it call DetachErode2 instead of DetachErode.

9 Stratigraphy

9.1 Layers beneath Nodes

Stratigraphy is normally handled using a stack of "layers" beneath each node. A layer is implemented using class *tLayer*, which was created by Nicole Gasparini and is defined in **tLNode**. Layer input (when a run is re-started) is handled in **tMesh**, while output is handled in **tOutput**. For discussion of implementation and an example application, see Gasparini et al. (2004).

9.2 High-Resolution Gridded Stratigraphy: tStratGrid

Handling layers as stacks beneath individual nodes can introduce problems when the nodes are moving significantly, as in the case of channel meandering simulations (see Clevis et al., 2006a *Computers and Geosciences* and Clevis et al., 2006b *Geoarchaeology*).). To deal with this problem, Quintijn Clevis developed an alternative method for handling stratigraphy based on a static raster grid that underlies the landscape. The approach is implemented in **tStratGrid**. As of this writing, it has only been used for a project on fluvial stratigraphy and geoarchaeology described by Clevis et al. (2006b).

10 Uplift, Subsidence and Baselevel Change: tUplift

Vertical motion of the landscape relative to a baselevel are handled by **tUplift**. The *tUplift* class provides a variety of different "uplift" geometries and methods (including some that have lateral motion).

11 Special Processes: Channel Meandering, Floodplain Deposition, Eolian Deposition, Vegetation

11.1 Stream Meandering: tStreamMeander

Stephen Lancaster developed a module for stream meandering in which nodes representing “large” streams (those with drainage area larger than some threshold) undergo meandering, using a model described by Lancaster and Bras (2002 *Hydrological Processes*). As channel nodes migrate, the mesh is constantly updated, with some nodes being added (representing point-bar deposits) and others deleted (representing bank erosion). The code to implement this, in **tStreamMeander**, is quite tricky, and has gone through a number of iterations and bug fixes. As of this writing, the most recent published implementation is Clevis et al. (2006b) on geoarchaeology simulation. Nate Bradley is also using the meander module to examine floodplain sediment residence-time distributions (Bradley and Tucker, 2007 AGU abstract). Applications that use the meander module have so far focused on a domain consisting of an idealized segment of a large river valley, rather than an entire drainage basin.

11.2 Floodplain Deposition: tFloodplain

For the floodplain evolution and geoarchaeology projects, a **tFloodplain** module was created that deposits “overbank” sediment on a landscape based on a modified version of Howard’s (1992) floodplain-sedimentation model. So far, this module has only been used in the context of the geoarchaeology simulation project (Clevis et al., 2006b).

11.3 Vegetation Dynamics: tVegetation

Around 1997, stimulated in part by simplifications used in climate-response modeling by Tucker and Slingerland (1997 *Water Resources Research*), we started thinking about the role of vegetation in controling channel network extent. The result was a simple dynamic model of vegetation growth and erosion, which is implemented in **tVegetation**. Daniel Collins explored this model for his MS thesis, and the results appear in Collins et al. (2004 *Journal of Geophysical Research*). A zero-dimensional version is presented (not coupled in CHILD) in Tucker et al. (2006 *GSA Bulletin*). Lee Arnold also used the **tVegetation** model in a chapter of his D.Phil. thesis

(Arnold, 2006 Oxford D.Phil.), which I hope will see the light of publication some day.

11.4 Dust Accumulation: tEolian

When the development team in Rafael Bras' geomorphology group (Bras, Gasparini, Lancaster and Tucker) began building CHILD at MIT in the mid to late 1990s, the effort was supported by a Corps of Engineers grant that was geared in part toward geoarchaeology modeling at Fort Riley, Kansas (see Zeidler et al., 1999 CEMML Technical Report). Because the geomorphology of that setting includes a strong component of glacial loess, we wrote a very simple **tEolian** module to drape dust on the landscape at a specified rate. To the best of my knowledge, it has never been published outside of technical reports.

12 Notes on the CHILD Interface Routines

As of academic year 2009-2010, CHILD has a set of interface functions. These interface functions make it possible for CHILD to be treated as a callable component by other code, so that it can be coupled with other models or simply driven from an external program. This interface is motivated by the CSDMS project. As of this writing (April 2010), most of the interface functions are only available in the Development version; the Release version has a more limited set of interface routines. The plan is to incorporate the full interface routines into the Release version once they are mature.

The interface routines are declared in `childInterface.h` (note that the Release version uses a different file, `childRInterface.h`).

12.1 How to Tell CHILD to Erode or Deposit Sediment

Sometimes it might be useful for an external program or module to modify CHILD's topography and deposits. For example, if CHILD is coupled to SedFlux, that model might calculate a certain amount of erosion or sedimentation. The interface function `ExternalErosionAndDeposition` supports this. It takes as an argument a vector of nodes indicating the depth. Depths are positive for deposition and negative for erosion. The vector must be in order by permanent ID number. As of now, this function does not handle multiple grain-size fractions, and will probably fail if it is used in a run with multiple sizes. Fixing this is on the "to do" list. Note also that this function should not be used to implement external tectonics, because it alters the layering (that is, deposition can actually add new layers rather than simply raising the elevation, and likewise erosion can remove layers).

12.2 How to Track Water and Sediment Fluxes

Coupling CHILD with other process models, like SedFlux, will often require information about CHILD's fluxes of water and sediment. One might think that this would simply involve querying CHILD's current values of water and sediment discharge at each node. However, in the most general case, it is not quite so straightforward. Imagine for example that one wants to run CHILD for 1000 years, then use its water and sediment fluxes over that time interval as input to another model. At the end of 1000 years, each CHILD node will have a value for water and sediment flux, but these values do not necessarily represent the average flux over the 1000-year period. Suppose the run in question was configured to use CHILD's stochastic rainfall module. In that case, the discharge could fluctuate widely over the course of the 1000-year period, so that the value at $t = 1000$ does not necessarily reflect the values at any other time during the 1000-year period. Likewise, the sediment fluxes will vary widely in concert with the discharges.

This means that in order to properly record CHILD's water and sediment fluxes over a given period of time, another approach is needed. To that end, an interface function called `TrackWaterAndSedFluxAtNodes` is provided. This function is used to activate tracking of water and sediment flux at specified nodes (if it is not already activated), or to re-set the list of nodes to track. The water and sediment fluxes at the tracked nodes is recorded in a set of files. Each node has one file.

Tracking of fluxes is implemented with help from the *tWaterSedTracker* class, which is defined in a pair of files `tWaterSedTracker.h` and `tWaterSedTracker.cpp`. The sequence of events works like this:

1. When the CHILD interface is instantiated at the beginning of a run, it automatically creates a *tWaterSedTracker*.

2. When the interface's `Initialize` method is called, it checks the main input file to see whether the user wants to tracker water and sediment fluxes (via the tag `OPT_TRACK_WATER_SED_TIMESERIES`). If yes, then the *WaterSedTracker's* `InitializeFromInputFile` method is called to set things up.

3. The `InitializeFromInputFile` method reads in the number of nodes to track and their (x, y) coordinates, and sets up an internal list of tracking nodes. It also creates a set of output files for water and sediment fluxes, one per node. The name for a typical flux output file looks like:

`myrun_node23_t0.water_sed`

Here, "myrun" is the base name, the node is 23, and the "t0" indicates that the records begin from time 0. (The start time is included because the list of nodes can change, and when that happens new files are created).

4. The *Erosion* module is told to activate tracking. This simply sets a flag inside *Erosion* and hands it a pointer to the *tWaterSedTracker*.

5. When tracking is switched on, the cumulative sediment volume at each tracking node is recorded at the end of each sub-time step. This is implemented in *Erosion's* `DetachErode` method via a call to the *tWaterSedTracker's* method `AddSedVolumeAtTrackingNodes`. Caveats: this only works when (1) detachment-limited mode is OFF (otherwise there are no defined sediment fluxes), and (2) `DetachErode`, rather than Nicole's `DetachErode2` (which implements the "$f(Q_s)$" algorithms), is used.

6. At the end of each storm, the *tWaterSedTracker's* `WriteAndReset...` method is called. This records the average flux of sediment (volume divided by elapsed time) at each of the tracking nodes, along with the fluvial discharge (which is constant during each storm).

To make use of the flux data, the output files need to be opened and read. An advantage is that the full time-series is provided (i.e., one data point per node per storm). A disadvantage is that file I/O is required.