18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

TELLUSIM: A Python Plug-in Based Computational Framework for Spatially Distributed Environmental and Earth Sciences Modelling

Willgoose, G.R. [1]

[1] *Centre for Climate Impact Management (C2IM), School of Engineering, The University of Newcastle, Callaghan, 2308 Australia*
Email: garry.willgoose@newcastle.edu.au

Abstract: TelluSim was designed as a new generation model to replace the well used, but old, code underpinning the SIBERIA landform evolution model. The SIBERIA code base dates back to the author's PhD work in the late 1980's. It is regarded as the first of the modern generation of physically based landform evolution models. The development of TelluSim had a three overriding objectives (1) to rid the code of the architectural limitations in SIBERIA and update the code base to modern standards, (2) to facilitate new science and applications, and (3) to leverage the original code base of SIBERIA where possible.

TelluSim is a Python-based computational framework for integrating and manipulating modules written in a variety of computer languages. TelluSim's specific application area is designed to be time evolving, spatially distributed systems, where each module simulates a physical process, and where the model is assembled from a combination of modules. TelluSim handles, using simple data from the modules, the interaction between the modules (i.e. where the physical processes interact). TelluSim consists of a main program that dynamically, at run time, assembles a series of modules. These modules can be written in any language that can be accessed by Python. Currently we have modules in Fortran and Python, with C to be supported soon. New modules are incorporated as plug-ins like done for a browser or Photoshop, simply by copying the module binary into a plug-in directory. TelluSim automatically generates a GUI for parameter and state I/O, and automatically creates the intermodule communication mechanisms needed for the computations. A decision to use Python was arrived at after detailed trials using other languages including C, Tcl/Tk and Fortran. An important aspect of the design of TelluSim was to minimise the overhead in interfacing the modules with TelluSim, and minimise any requirement for recoding of existing software, so eliminating a major disadvantage of more complex frameworks (e.g. JAMS, openMI). Several significant Fortran codes developed by the author have been incorporated as part of the design process and as proof of concept. In particular the SIBERIA landform evolution code (a high performance F90 code, including parallel capability) has been broken up into a series of TelluSim modules, so that the SIBERIA model now consists of a Python script of 20 lines. These 20 lines assemble and run the underlying modules (about 50,000 lines of Fortran code). The presentation will discuss in more detail the design of TelluSim, and experiences of the advantages and disadvantages of using Python relative to other approaches (e.g. Matlab, R).

We plan to release TelluSim V1.0 as open source in mid 2009. This version will ship with the TelluSim implementation of SIBERIA and our digital elevation model analysis tools. This version will, at a minimum, support both LINUX and OSX. The Windows implementation schedule is less clear because the current version of Python for Windows relies on a superseded version of Visual Studio to develop Python extension modules (i.e. the Fortran modules).

Keywords: *modelling framework, python, landform evolution modelling, erosion, hydrology.*

1. INTRODUCTION

The original objective for the development of TelluSim was relatively simple. It was to replace the well used, but old, code underpinning the SIBERIA landform evolution model (Willgoose et al, 1991; Willgoose, 2005). The SIBERIA code base dates back to the author's PhD work in the late 1980's. It is regarded as the first of the modern generation of physically based landform evolution models. Since its development it has found wide use world wide in both the science community and in applications in mine rehabilitation design (Willgoose and Riley, 1998) and hazardous and nuclear waste containment structure design (Crowell et al, 2005). In these latter applications it is used to examine the long term evolution, under erosion, of constructed landforms to examine whether the containment structure will be breached. The myriad of applications and pressures for customization had led to a degree of code forking. The code is also written in FORTRAN, originally F77 but updated to F95 with dynamic data structures. However, there are architectural limitations dating from its original development that have made code maintenance time consuming and significant extensions difficult to implement. On the other hand the code is extremely fast compared to more recent landform evolution models, it supports parallel computation, and because of the extensive debugging in application, relatively bug free.

TelluSim was designed as a new generation model to replace to SIBERIA. The development of TelluSim had a three overriding objectives (1) to rid the code of the architectural limitations in SIBERIA and update the code base to modern standards, (2) to facilitate new science and applications, and (3) to leverage the original code base of SIBERIA where possible. Specifically the objectives included:

- Open source, platform independent numerics, and platform independent graphical user interface. SIBERIA has a GUI but it is platform dependent (Windows only). Many SIBERIA applications must be run on high performance non-Windows computers that requires running without the GUI.
- As far as possible the code base should be compiler language independent. While the SIBERIA code base is in F95, extension in other languages should be possible so researchers with skills in other languages can be accommodated. On the other hand there should be no presumption of any expertise in any particular language, so that interfacing a researcher's code should not require a deep knowledge of a 2nd language.
- It should be modular so that extensions by other researchers can be easily incorporated.
- These modules should be easily assembled into complete self-contained models. Models are how the modules interact. This model construction process should be interactive and scripting skills required to assemble a model should be minimised.
- There should be minimal refactoring of code necessary to incorporate models, modules or capabilities from existing science models. This requirement underpinned the requirement to be

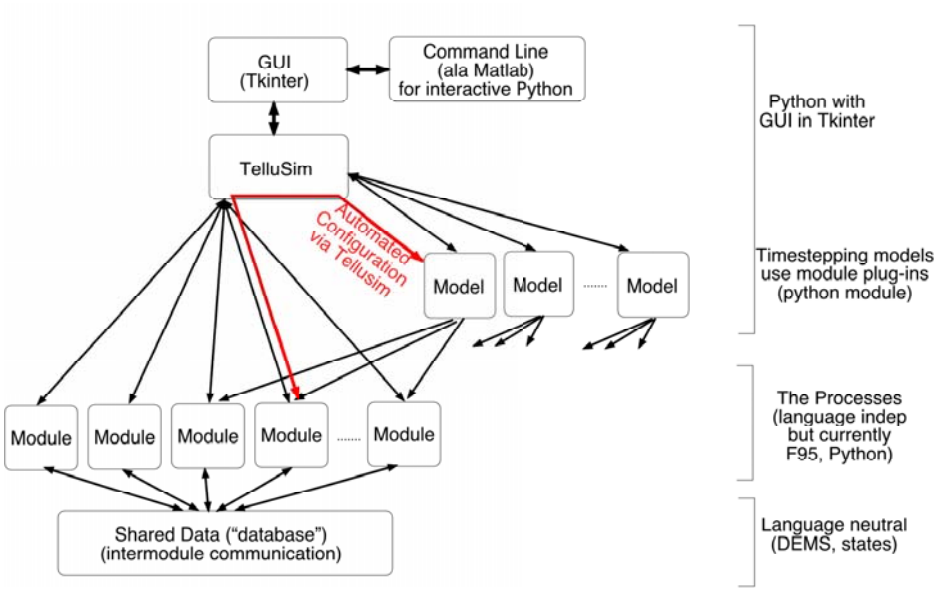

Figure 1: TelluSim Schematic

compiler language independent.

- The framework should be computationally efficient since existing runs in SIBERIA can take CPU days, and some quantitative risk assessments can take CPU months. The ability to run in parallel (either with openMP or MPI-2) is thus crucial. As far as possible this parallel capability should be hidden from the user because most users are unfamiliar (and thus inexperienced) with parallel tools.
- The framework should be memory efficient because SIBERIA bumps against 32 bit memory limitations for many current problems. Support for 64 bit memory highly desirable.
- An ability to plug and play with processes and parameters should be supported for research users while practical users should see a more structured environment that protects the user from unnecessary complexity.

As this paper demonstrates TelluSim has largely met these objectives. The task of replacing SIBERIA is mostly complete. However, during the development of TelluSim it became clear that TelluSim had wider applicability in the community of researchers who do dynamical modelling within GIS (i.e. time evolving physics that is spatially distributed across a time evolving landform), and it is to these types of applications and the extensions to TelluSim motivated by this broader need that this paper is largely devoted.

2. TELLUSIM ARCHITECTURE

TelluSim is written entirely in Python using Python tools to provide the user interface through a GUI and an optional command line. We evaluated (partially implemented TelluSim on some cases) a number of enabling platforms and decided that Python provided the best set tool set to achieve our objectives (Table 1). This Python code runs the models via the GUI. The models themselves assemble and do the computations using the modules. Models are typically scripts written in Python (though this is not essential) and the modules that the models call can be a mixture of executable code (e.g. compiled code written in FORTRAN, C, etc) or scripts in Python (Figure 1). A close analogy is how models are developed in R/S and Matlab. Models are the scripts or M-files, while the modules are (loosely) the commands on the command line.

Table 1. A comparison of candidate platforms upon which TelluSim could have been implemented.

	Python	R & S	Octave	Tcl/Tk	Java	Matlab
Open Source	Y	R only	Y	Y	Y	-
Scripting	Y	Y	Y	Y	Y	Y
Platform independent GUI Support	Y	-	?	Y	Y	Y
Science Support	Y	Y	?	-	Y	Y
Language Independence	Y	Y	-	-	-	Y
Parallel Computation	V2.6	-	?	-	-	Y
Computational Efficiency	Y	Y	?	-	?	Y

Typically the computation bound parts of the computations are performed with compiled executable code, will less compute intensive components implemented in Python. The modules communicate between each other via shared data that are in data structures shared between all modules (e.g. states, parameters, partial results). TelluSim manages how data is stored in the data structure. If data is only required by a single module and is not used by another module then data can be stored within the module as local data that is either (1) local data to the module that is created and destroyed each time the module is called (e.g. local variables created within a subroutine) or (2) saved between invocations of the module (e.g. global module storage in Fortran). If, however, the data is required by another module then TelluSim creates global storage in the shared data structure and instructs the module where the data is stored. Administration routines are provided by TelluSim that simply allow the modules to (1) copy the data into the module at the start and copy it out at the end of the module (if the copying overhead is not too large), or (2) directly access the data structure with pointers. TelluSim determines what states need to be shared through two subroutines potentially defined in the module interface (states_info() and set_shared_states() in Figure 2).

The data structures that are used in the global data structure are similar to those used in openMI but modified so that they are equally easy to access with Fortran and C. No objects are defined in the interface and all data

is stored in a multidimensional array included scalar variables, which are treated as 0D arrays. Python (i.e. the TelluSim GUI, and models and modules written in Python) sees all data as NumPy arrays, while C and Fortran see them as native arrays (though with index ordering defined as in Fortran).

The use of a shared data structure for passing information between the modules avoids much of the complexity of message passing in openMI, and the simple data structures mean that the user does not need a background in C#, which is required to understand the data interfacing documentation in openMI since almost all documentation available to date is written for C# objects.

3. TELLUSIM COMPONENTS

3.1. Modules

The modules have a standard interface designed to make it easy to write wrappers around existing code. Computations are all done in one subroutine (compute() in Figure 2). All the subroutines are designed to allow TelluSim to assemble the modules in the most memory and CPU efficient mode (Figure 2). Not all of the subroutines are needed as Python has the ability to examine a module to see if a routine exists in the module. If the module doesn't exist then TelluSim makes sensible assumptions. For instance if the parameter_info() routine does not exist in the module then TelluSim assumes that the module doesn't have any parameters.

Modules can themselves be a combination of modules. For instance, there may be some CPU and memory advantage in merging two modules.

3.2. Models

A model is the main basis of running simulations. It's a self-contained set of calculations. By default models are written in Python but there is no restriction within TelluSim that requires this.

A model typically consists of 5 components

- Module initialisation.
- Any data preprocessing and/or data/file input.
- A timestep loop. Within the loop each process is executed once. This loop typically calls the modules. Not all modules need to do computation. For instance a graphics model might output visualisation while the model is calculating. Likewise another module might be some form of controller to stop and start the run, change parameters, explore the data, etc.
- Any data post-processing and/or data/file output.
- Module finalisation.

Based on the Python code in the model and the routines in the modules used by that model TelluSim knows what the parameters for the model are and automatically constructs a parameter input user interface for the model. Each model has a parameter database for storing its parameters, and the parameter input interface configures and manages this database. The existence of the model initialisation and finalisation stages is a key part of a TelluSim model. They are needed to set shared states and allow state and module initialisation. The highly structured nature of the model will also allow us to develop simple means of constructing model writing tools at a later stage in TelluSim development.

3.3. TELLUSIM Frontend

Parameters
Parameters are scalar variables (integer, real, complex, string, filenames). In addition, any parameter can be a Python expression (involving other parameter names), which provides a way that parameter values can be made a complex function of any other parameter or set of parameters. The Python expression can be arbitrarily complex, with the only restriction being that it must be able to be evaluated with the Python command eval().

DEM Management
In this paper we use the term DEM here in a slightly different way to normal terminology. Normally a DEM is defined as both the elevations and the positions at which those elevations are defined. A TelluSim DEM is

only the locations at which states (e.g. elevation, soil depth, surface slope) are defined. The corresponding elevations are defined as a state. A DEM in TelluSim is one of a (1) regular grid (either square or rectangular), (2) triangular irregular network of x,y node positions with or without an associated triangulation or (3) path of ordered x,y data along a line. The rectangular grid has not been commonly used in environmental applications but allows for raw data from satellites when the position is defined in latitudes and longitudes.

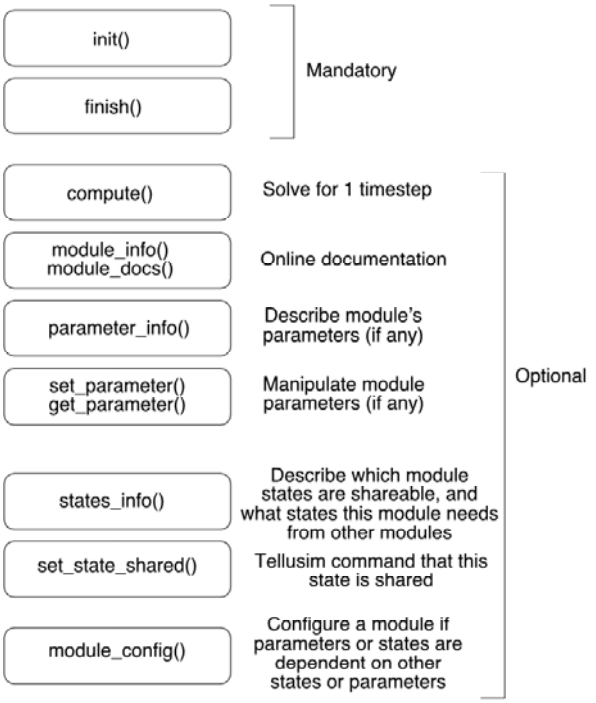

Figure 2: TelluSim module schematic

Shareable States

The shared states are 0D (i.e. scalar), 1D, 2D, 3D and 4D arrays of integer, real, strings or complex. These states can be cross-referenced to a digital elevation model (DEM) or an irregularly spaced scale (e.g. a time scale). This means that the position of a node is stored separately from the states at that node allowing for efficiency of memory usage. Different states may be based on different DEMs (e.g. rainfall may be on one grid while geology and elevation might be on others). At model initialisation TelluSim determines which states must be shared on the basis of what input other modules require. If another module requires a state then it must be shared. TelluSim creates the shared storage and instructs the modules how to access these shared data.

4. A MODEL CASE STUDY: SIBERIA LANDFORM EVOLUTION MODEL

4.1. SIBERIA Background

The initial project objective was that TelluSim would replace SIBERIA. The main rationale for the wholesale architectural change was that SIBERIA had become overly complex to maintain. It was a significant time sink for the author simply to keep it going, let along to do any non-trivial extensions for new science. TelluSim was designed to be the new foundation for science in the area of soil pedogenesis modeling and ecohydrology problems. Both of these areas had been attacked in SIBERIA in the past but the compromises in the original design of SIBERIA in the 1980's made it difficult to meet all the science objectives.

SIBERIA is 50,000 lines of highly optimized Fortran 95 code, with parallel capability. The current version of SIBERIA uses openMP for parallel computations while a pre-2000 version uses Parallel Virtual Machine (PVM) for workstation clusters. PVM is roughly functionally equivalent to MPI-2. SIBERIA was written in 1989 and has many architectural limitations dating from that time. Dynamic data structures from Fortran 90 have been bolted on, the code has been heavily modularized with Fortran 90 modules. Many design compromises remain. Chief among them is the lack of any form of platform independent GUI, though we have one written in GLUT for Windows dating from the late 1990's and one written by others in ARC-GIS (called Arc-Evolve).

In addition to SIBERIA, there is large suite of related analysis programs that are central to SIBERIA calibration and testing. These codes are a mixture of Fortran and C, with and without GUIs that use SIBERIA data file formats.

Finally, despite SIBERIA's numerical efficiency it is not unknown for runs of CPU days and Monte-Carlo runs can take CPU months so computational efficiency is of paramount importance. In recent years with our soil pedogenesis research we regularly hit 32 bit memory limits (particularly for Windows) so memory efficiency and support of 64 bit memory is important.

4.2. SIBERIA Refactoring Experience

About 90% of SIBERIA has been running in TelluSim since 2008. Originally, as part of the TelluSim development and testing, this implementation used a makeshift shared data structure but the shared data structures and supporting code are now largely complete. The main part of SIBERIA that has not been ported is our multilayer soil/geology and sediment tracking module. This has been delayed because it interacts with other work currently underway on a more advanced pedogenesis model (Cohen, et al 2009).

A module takes about 2-3 hours to refactor for TelluSim provided it is already reasonably modularised (SIBERIA is already heavily modularised). The Python interfacing of the F95 code is done automatically with the f2py tool in NumPy. The tool f2py is limited to interfacing simple FORTRAN 90 without dynamic data structures in the interface and this sometimes requires layering a second Fortran module around the first to hide from f2py any dynamic data structures in the global storage of the Fortran module.

CPU usage is unchanged because the compiled Fortran modules do the bulk of the compute intensive work, and the only overhead of TelluSim is in assembling the model from the modules (a one off cost at the start of the run), and in the GUI which is largely inactive during the computations. Memory usage is a little higher because of overheads in the abstraction required by the shared data structures. The TelluSim implementation of SIBERIA is about 20 lines of Python code. Of course all the original Fortran code still exists but as modules that the model calls rather in the model itself.

5. ACHIEVING THE TELLUSIM DEIGN OBJECTIVES

5.1. Open Source

Python and its standard libraries is an excellent platform for open source development. Documentation for the some of the extension libraries can be a bit patchy (e.g. wxPython). To date the only non-standard extension modules used by TelluSim are NumPy and wxPython. For international geosciences the recently funded Community Surface Dynamics Modelling System (CSDMS) will support Python code. We are currently collaborating on incorporating TelluSim into CSDMS (Syvitski, et al, 2004).

5.2. Platform Independence

The main reasons for platform dependencies in science codes are (1) a platform dependent GUI library, and (2) system calls that are operating system dependent (e.g. directory open/close/create/delete). Python has excellent tools in both of these categories and the only a platform dependencies in TelluSim are some minimal setting of font sizes and types at the beginning of the code to make the GUI look native.

Currently TelluSim is operational for:

- Linux: Intel CPU, both 32 bit and 64 bit.
- Apple OSX 10.4 and 10.5: both PowerPC and Intel CPUs, 32 bit only.

A Windows implementation was deferred until TelluSim was fully up and running because the current version of Python requires a superseded version of Visual Studio to develop extension libraries (TelluSim modules are implemented as Python extension libraries).

5.3. Module Language Independence

To date we have focused on the porting of our Fortran code using the f2py tool in NumPy. This has been an excellent tool for interfacing Fortran to Python. It does have some limitations. It won't support passing of structures, nor will it pass pointers. But overall our experience with it has been highly satisfactory.

Similar tools exist for interfacing other languages to Python including C and Java.

5.4. Modularity

The dynamic loading of modules by Python means that the plug-in architecture of TelluSim has been easy implement, has required little in the way of refactoring of existing code, and is mostly compiler language independent. The flexibility of Python's ability to interrogate a loaded module to discover what routines are implemented means that a module can be as minimal as needed, reducing the learning curve for a new module developer. This also provides flexibility into the future. If extra capabilities are required in modules for future unanticipated applications routines can be added to the interface as needed without having to recode existing modules.

5.5. Parallel Computation

Python's parallel capabilities are currently rather disappointing. Python V2.5 has a 'threads' module but these threads are executed sequentially. This is because of the way the Python interpreter works internally. Thus even on a multi-core processor the maximum performance is only the equivalent of a single core. Python V2.6 (released in late 2008) introduced a 'multiprocessing' module that is truly parallel. We have been unable to test it because at the time of paper writing.

The most mature support for external parallel libraries is for MPICH, an open source implementation of MPI-1. Our experience with PVM indicates that MPI-2 is needed for our applications. Recently a new library supporting open-MPI (which is MPI-2) has been released but we have not had time to test it.

Finally we can do parallel computation internally within a single modules using openMP if the compiler supports it. Just how efficient this is depends on the application. The parallel computation must be started each time the module is called (for each timestep) and finished at the end of the module (for each timestep). How successful this is depends on the parallel startup costs. However, it doesn't meet our needs for model-level Monte-Carlo simulation, which for TelluSim's structure must be done in Python.

6. CONCLUSIONS AND FUTURE PLANS

We plan to release TelluSim V1.0 as open source in mid 2009. This version will ship with the TelluSim implementation of SIBERIA and our digital elevation model analysis tools. This version will, at a minimum, support both LINUX and OSX. The Windows implementation schedule is unclear because the current version of Python for Windows relies on a superseded version of Visual Studio to develop Python extension modules (i.e. the Fortran modules). Accordingly we have judged this as having a lower priority than getting TelluSim running.

More work is needed on the interactive command input option. The current GUI is built with the Tkinter library. While Tkinter has proven to be simple to program and robust in its performance it lacks inbuilt support for a command line interpreter. We are currently recoding the GUI in wxPython, which has a widget that provides a full command interpreter. This will allow us to provide robust support for command line input, but does require us to recode the GUI. In addition, there are some subtle issues to do with shared states and module initialisation that we still need to resolve to allow unrestricted support of command line input.

The current version of TelluSim is not parallel at the TelluSim level. This limits our Monte-Carlo modeling (e.g. Willgoose, et al, 2003). Implemented parallel will require moving to Python 2.6.

ACKNOWLEDGMENTS

The author is funded by an Australian Professorial Fellowship from the Australian Research Council.

REFERENCES

Cohen, S., G. R. Willgoose, and G. R. Hancock. 2009. A computationally efficient spatially distributed soil pedogenesis model: modelling framework and analysis of hillslope catena. *Journal of Geophysical Research (Surface Processes)*:in press.

Crowell, K. J., C. J. Wilson, L. J. Lane, B. D. Newman, and T. G. Schofield. 2005. *Impact of Extreme Events and Soil Hydraulic Conductivity on the Evolution of a Mesa-top Waste Repository Cover*. Pages H43D-0523 in Fall Meeting of the American Geophysical Union. AGU, San Francisco.

Syvitski, J., C. Paola, R. Slingerland, D. J. Furbish, P. L. Wiberg, and G. E. Tucker. 2004. *Building a Community Surface Dynamics Modeling System: Rationale and Strategy*. http://instaar.colorado.edu/deltaforce/workshop/csdms.html.

Willgoose, G. R. 2005. Mathematical Modeling of Whole-Landscape Evolution. *Annual Review of Earth and Planetary Sciences* 33:443-459.

Willgoose, G. R., and S. J. Riley. 1998. An assessment of the long-term erosional stability of a proposed mine rehabilitation. *Earth Surface Processes and Landforms* 23:237-259.

Willgoose, G., R. L. Bras, and I. Rodriguez-Iturbe. 1991. A Coupled Channel Network Growth and Hillslope Evolution Model .1. Theory. *Water Resources Research* 27:1671-1684.

Willgoose, G. R., G. R. Hancock, and G. A. Kuczera. 2003. A framework for the quantitative testing of landform evolution models. Pages 195-216 in P. R. Wilcock and R. M. Iverson, editors. *Predictions in geomorphology*. American Geophysical Union, Washington DC.