# Updating TOPLATS with Fortran 2003

Nathaniel Chaney
Colby Fisher
Amanda Siemann
Ruolan Xu
Wang Zhan

Department of Civil and Environmental Engineering

January 18, 2013

## 1 Introduction

Land surface hydrologic models are used to simulate the interaction between the atmosphere and land surface. Their main objective is to model the water and energy balances and also to accurately partition the surface fluxes. Their ability to simulate these fluxes from limited data is key for monitoring the hydrologic cycle at continental and global scales. One of the models used at a regional scale is TOPLATS which was developed at Princeton University during the 1990's. Time has shown that this model is an excellent bridge between coarse scale models and very fine resolution models, making it an excellent candidate to replace coarse scale models to study global hydrology. This is also due, in part, to the large increase in computational power over the past two decades.

The model was originally written in Fortran 77 and it falls along the lines of "spaghetti code". It was never developed with the intent of making it easy to switch components of the models. Some of the inherent deficiencies are the lack of proper decoupling between the I/O and the model through an appropriate interface and the fact that the parameterizations of physical processes (evaporation, infiltration, and runoff) were not appropriately modularized, making it practically impossible to quickly and efficiently update the model with new research. With a desire to now update this model and run it over much larger spatial scales than what it was originally intended for, we see this as a perfect opportunity to gravitate towards object-oriented programming in Fortran 2003.

Main goals of this project include: 1) Modularize the code based on the physics so that it will be easy to make changes within compartments; 2) Improve data structures by introduce derived data types; 3) Add parallel computing using OpenMP; 4) Make tests to ensure subroutines as well as the whole model produce same results as the original model.

The following table concludes the programming tools we have used.

Table 1: Tools used in the project

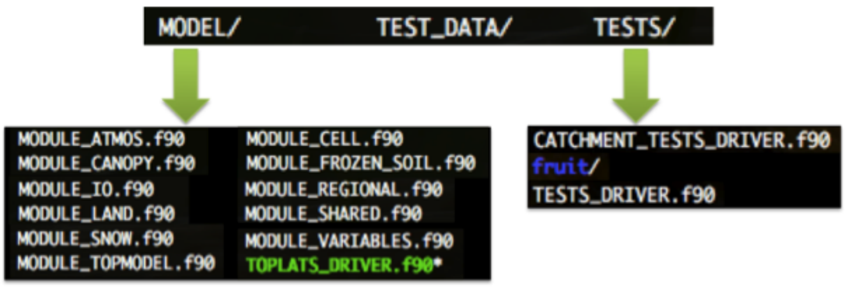
| Purpose | Tools |
|---|---|
| Language | Fortran 2003 |
| Testing | FRUIT |
| Version control | github |
| Profiling | Valgrind |
| Documentation | Doxygen |

## 2 Interface

(Nate)
Driver
Separate I/O from actual model
Simplified input files

## 3 Modularization

One of the tasks in the project is modularization. We set up 11 modules for the model and split tests from its own interface. Each module is written in a separate file. By modularization, we are able to reduce the total number of files from more than 50 to 11. The new TOPLATS model files are shown in Figure 3. The model are consist of two main parts: MODULE_CELL and MODULE_CATCHMENT. MODULE_CELL uses MODULE_LAND, MODULE_ATMOS, MODULE_CANOPY and MODULE_SNOW. The program IO is separated into MODULE_IO and variables and data structures defined in MODULE_VARIABLES. With modularization, swapping physics in the model becomes much easier since we are able to replace the subroutine or module instead of making changes to every related variables and functions throughout the model.



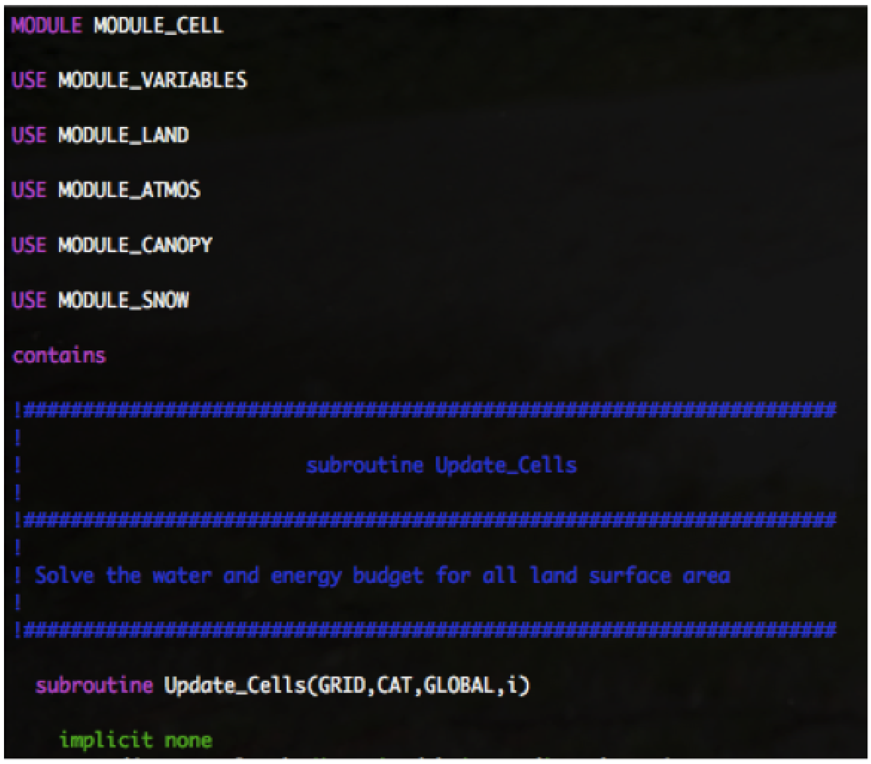
```
MODEL/            TEST_DATA/        TESTS/

MODULE_ATMOS.f90      MODULE_CELL.f90
MODULE_CANOPY.f90     MODULE_FROZEN_SOIL.f90
MODULE_IO.f90         MODULE_REGIONAL.f90
MODULE_LAND.f90       MODULE_SHARED.f90
MODULE_SNOW.f90       MODULE_VARIABLES.f90
MODULE_TOPMODEL.f90   TOPLATS_DRIVER.f90*

CATCHMENT_TESTS_DRIVER.f90
fruit/
TESTS_DRIVER.f90
```

Figure 1: New TOPLATS model

Within each module, as shown in Figure 3, we first listed shared variables and structures from other modules, defined structures being shared by subroutines in the module if there is any, and then listed the subroutines that are related to the module. Routines are being regrouped by their function.



```
MODULE MODULE_CELL

USE MODULE_VARIABLES

USE MODULE_LAND

USE MODULE_ATMOS

USE MODULE_CANOPY

USE MODULE_SNOW

contains

!######################################################################
!
!                        subroutine Update_Cells
!
!######################################################################
!
! Solve the water and energy budget for all land surface area
!
!######################################################################

  subroutine Update_Cells(GRID,CAT,GLOBAL,i)

    implicit none
```

Figure 2: Snapshot of MODULE_CELL.f90

## 4 Variables

In the original version of TOPLATS, the variables were stored in one lengthy list. As previously mentioned, the hundreds of variables were passed into subroutines individually, causing extremely large subroutine calls. For the updated version of TOPLATS developed in this project, these hundreds of variables were organized into derived data types (structures) based upon function or purpose. In this way, structures could be passed into the modules and subroutines in relevant groups, thereby cleaning up the otherwise unmanageable code. In each larger subroutine, such as Atmos, the variables used in the code were replaced with the corresponding member of the structure. For each smaller subroutine, the members of the structures were passed into the subroutine individually to avoid passing in more variables than necessary.

Not only was the original version of TOPLATS crippled by the massive subroutine calls, but it was also impaired by the unreasonable number of help files storing the initialized variables with their corresponding types. Each subroutine was associated with a help file which initialized new variables in each subroutine but which also re-initialized all variables which were passed into the subroutine. This process of re-initializing variables over and over was cumbersome and unnecessary. In the updated version of TOPLATS, the variables in structures are initialized within the structure, and the rest of the variables are only initialized within the larger subroutine in which they are used. In this way, hundreds of files were removed.

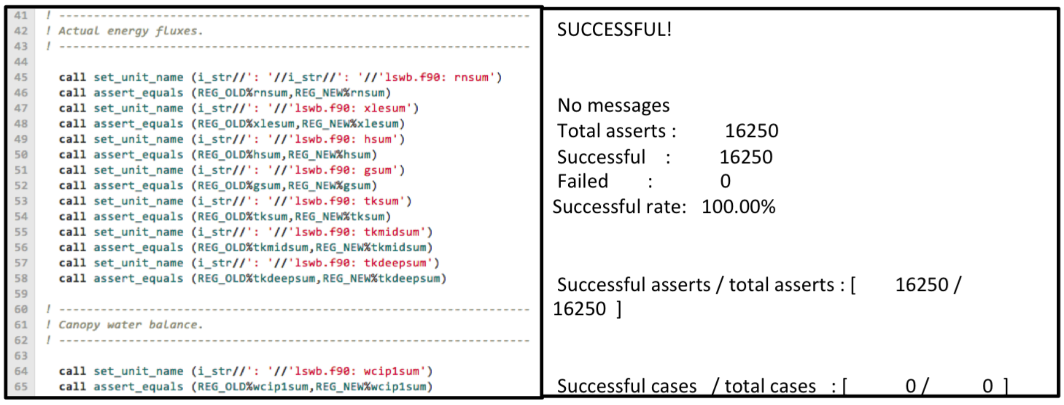
Another limitation of the original version of TOPLATS was the use of static memory allocation. Using this method, the size of all arrays needed to be provided at compile time, meaning that the users needed to alter the source files every time they used the model for a different domain. In order to improve upon this, the dynamic memory allocation features implemented in Fortran 90 were used. By dynamically allocating the arrays, the user is able to specify dimensions at run time, allowing for the model to be used across multiple domains without maintaining and compiling multiple versions of the model code.

## 5 Tests

FORTRAN Unit Test Framework (FRUIT) is utilized for testing. We have two test programs - CATCHMENT_TESTS_DRIVER and TESTS_DRIVER, each serving a different purpose.

The CATCHMENT_TESTS_DRIVER runs the model and compares all the output variables with the original model. Although written in a messy way, the original model is validated in its physics and results. We have validated output from a sample input dataset, all we need in the catchment tests is to make sure our modified model reproduces the output at every time step. These tests were the first thing we wrote, which we ran every time before we made a commit to ensure we did not change any essential physics or algorithms.

The TESTS_DRIVER runs tests on various subroutines. Considering the amount of subroutines and functions and that the model was previously validated, we only tested about 50 of them. Random values were given to the subroutine or function being tested, and the result value was compared with the true value. The true value was previously calculated by the original subroutine or function.



```
! ------------------------------------------------------------------
! Actual energy fluxes.
! ------------------------------------------------------------------

  call set_unit_name (i_str//': '//i_str//': '//'lswb.f90: rnsum')
  call assert_equals (REG_OLD%rnsum,REG_NEW%rnsum)
  call set_unit_name (i_str//': '//'lswb.f90: xlesum')
  call assert_equals (REG_OLD%xlesum,REG_NEW%xlesum)
  call set_unit_name (i_str//': '//'lswb.f90: hsum')
  call assert_equals (REG_OLD%hsum,REG_NEW%hsum)
  call set_unit_name (i_str//': '//'lswb.f90: gsum')
  call assert_equals (REG_OLD%gsum,REG_NEW%gsum)
  call set_unit_name (i_str//': '//'lswb.f90: tksum')
  call assert_equals (REG_OLD%tksum,REG_NEW%tksum)
  call set_unit_name (i_str//': '//'lswb.f90: tkmidsum')
  call assert_equals (REG_OLD%tkmidsum,REG_NEW%tkmidsum)
  call set_unit_name (i_str//': '//'lswb.f90: tkdeepsum')
  call assert_equals (REG_OLD%tkdeepsum,REG_NEW%tkdeepsum)

! ------------------------------------------------------------------
! Canopy water balance.
! ------------------------------------------------------------------

  call set_unit_name (i_str//': '//'lswb.f90: wcip1sum')
  call assert_equals (REG_OLD%wcip1sum,REG_NEW%wcip1sum)
```

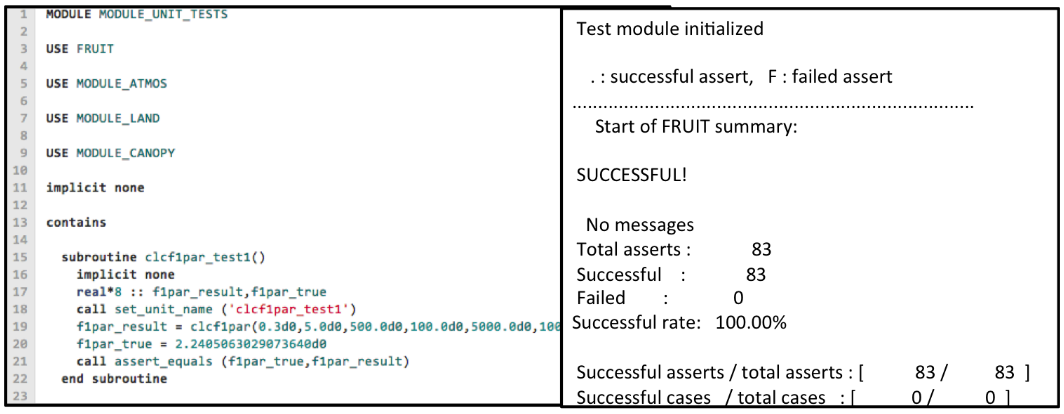
```
SUCCESSFUL!


No messages
Total asserts :       16250
Successful   :       16250
Failed       :           0
Successful rate:   100.00%


Successful asserts / total asserts : [       16250 /
16250  ]


Successful cases   / total cases   : [          0 /          0  ]
```

Figure 3: Catchment tests example code(left) and success message (right)



```
MODULE MODULE_UNIT_TESTS

USE FRUIT

USE MODULE_ATMOS

USE MODULE_LAND

USE MODULE_CANOPY

implicit none

contains

  subroutine clcf1par_test1()
    implicit none
    real*8 :: f1par_result,f1par_true
    call set_unit_name ('clcf1par_test1')
    f1par_result = clcf1par(0.3d0,5.0d0,500.0d0,100.0d0,5000.0d0,100
    f1par_true = 2.2405063029073640d0
    call assert_equals (f1par_true,f1par_result)
  end subroutine
```

```
Test module initialized

   . : successful assert,   F : failed assert

.........................................................................
    Start of FRUIT summary:

SUCCESSFUL!

  No messages
Total asserts :          83
Successful   :          83
Failed       :           0
Successful rate:   100.00%

Successful asserts / total asserts : [          83 /          83  ]
Successful cases   / total cases   : [           0 /           0  ]
```

Figure 4: Subroutines tests example code(left) and success message (right)

## 6 Parallelization

Parallelization is currently implemented using OpenMP and uses up to 8 cores on a single node, depending on the preference of the user. In order to do this, the domain is split up into equal sized blocks and then is distributed among the threads. Calculations are then performed for each cell before the domain is reassembled and the regional processes are calculated. For the domain size that we are using the parallelization does not have a significant impact on the calculation time. It should be noted though that for future work, as the domain size increases, we believe that we will find that parallelization becomes increasingly important. In order to run the model on a global scale, we plan to implement parallelization through MPI in order to maximize the computational power of each node.

## 7 Valgrind

(Colby)
Valgrind detects uninitialized variables

## 8 Profiling

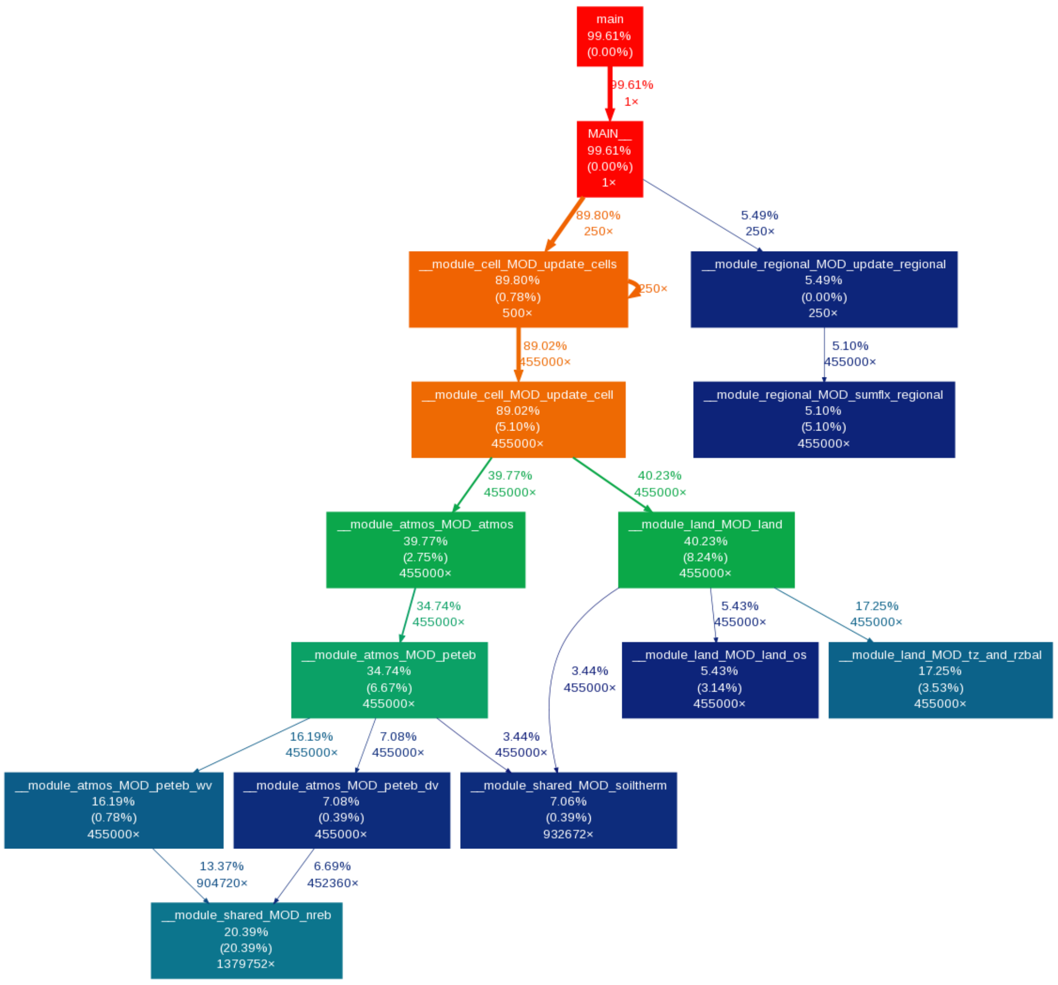

main 99.61% (0.00%)
99.61% 1×
MAIN__ 99.61% (0.00%) 1×
89.80% 250×
5.49% 250×
__module_cell_MOD_update_cells 89.80% (0.78%) 500×
250×
__module_regional_MOD_update_regional 5.49% (0.00%) 250×
89.02% 455000×
5.10% 455000×
__module_cell_MOD_update_cell 89.02% (5.10%) 455000×
__module_regional_MOD_sumflx_regional 5.10% (5.10%) 455000×
39.77% 455000×
40.23% 455000×
__module_atmos_MOD_atmos 39.77% (2.75%) 455000×
__module_land_MOD_land 40.23% (8.24%) 455000×
34.74% 455000×
5.43% 455000×
17.25% 455000×
3.44% 455000×
__module_atmos_MOD_peteb 34.74% (6.67%) 455000×
__module_land_MOD_land_os 5.43% (3.14%) 455000×
__module_land_MOD_tz_and_rzbal 17.25% (3.53%) 455000×
16.19% 455000×
7.08% 455000×
3.44% 455000×
__module_atmos_MOD_peteb_wv 16.19% (0.78%) 455000×
__module_atmos_MOD_peteb_dv 7.08% (0.39%) 455000×
__module_shared_MOD_soiltherm 7.06% (0.39%) 932672×
13.37% 904720×
6.69% 452360×
__module_shared_MOD_nreb 20.39% (20.39%) 1379752×

Figure 5:

## 9 Conclusion

We have accomplished most of our goals. Derived data types are implemented; modules are set up making it possible to swap physics; I/O is separated from other routines and enhanced; parallelization is added; main program is replaced with a clean interface.

Incredible changes were made to TOPLATS, which enable us to put the model from two decades ago to new uses. Nathaniel Chaney will continue to add MPI parallelization and run it on Blue Waters on high resolution global grids.

This project and the course has been beneficial to everyone in the project. We have learned how to effectively collaborate on the same program with great help form github. Testing has become an important part of our programming practices, which was almost absent before taking this course. All the tools we have used, github, Doxygen, Valgrind will continue to serve as our programming companions.