Geoscientific Model Development Discussions

Geoscientific Model Development Discussions is the access reviewed discussion forum of *Geoscientific Model Development*

A Python implementation of QTCM1

J. W.-B. Lin

`qtcm` 0.1.2: A Python Implementation of the Neelin-Zeng Quasi-Equilibrium Tropical Circulation model

J. W.-B. Lin

Physics Department, North Park University, 3225 W. Foster Ave., Chicago, Illinois 60625, USA

Correspondence to: J. W.-B. Lin (jlin@northpark.edu)

A Python implementation of QTCM1

J. W.-B. Lin

Abstract

Historically, climate models have been developed incrementally and in compiled languages like Fortran. While the use of legacy compiled languages results in fast, time-tested code, the resulting model is limited in its modularity and cannot take advantage
5 of functionality available with modern computer languages. Here we describe an effort at using the open-source, object-oriented language Python to create more flexible climate models: the package `qtcm`, a Python implementation of the intermediate-level Neelin-Zeng Quasi-Equilibrium Tropical Circulation model (QTCM1) of the atmosphere. The `qtcm` package retains the core numerics of QTCM1, written in Fortran to optimize
10 model performance, but uses Python structures and utilities to wrap the QTCM1 Fortran routines and manage model execution. The resulting "mixed language" modeling package allows order and choice of subroutine execution to be altered at run time, and model analysis and visualization to be integrated in interactively with model execution at run time. This flexibility facilitates more complex scientific analysis using less com-
15 plex code than would be possible using traditional languages alone, and provides tools to transform the traditional "formulate hypothesis → write and test code → run model → analyze results" sequence into a feedback loop that can be executed automatically by the computer.

1   Introduction

20 Although early weather and climate models, beginning with Richardson's "Forecast Factory" in 1922 (Edwards, 2000), led the development of the field of scientific computing, over the past few decades, climate models have not, in general, kept up with advances in computing languages and structures. Many climate models are still written in compiled languages (primarily Fortran), and utilize the same programming structures
25 familiar to a Fortran programmer of the 1970's. On the positive side, this continued reliance on Fortran results in very fast code that runs on almost all platforms, the ability

to reuse legacy code, and the availability of well-tested libraries, which have been optimized over decades of use.

At the same time, the continued development of climate models in Fortran has made it difficult to take advantage of advances in programming languages that increase the modularity and robustness of scientific code. Being mainly a procedural language, Fortran lacks the default programming structures to organize a model into truly self-contained units, thus limiting modularity. Fortran subroutine function calls may utilize long and unwieldy argument lists, its default variables are not self-describing, and variables exist in a loosely controlled namespace; this can result in brittle code where undetectable errors easily propagate. Finally, as a compiled language, Fortran is non-interactive, which hinders informal small-scale testing and prevents users from interacting with the model at run time. Recent versions of Fortran have added some of these modern features to the language, but scientific programs, in general, make limited use of these new features.

Modern computer languages have constructs that overcome many of these difficulties, though at a penalty in performance. These languages possess the tools to manage the variable namespace that older procedural languages lack, and thus modern languages can avoid lengthy hard-wired argument lists through the use of dictionaries and the creation of specialty data structures and classes that ensure the right variables are available and used when needed. Modern object-oriented frameworks bind metadata to variables, as well as the functions that act on the variables. Such contextualized variables make possible additional levels of modular decomposition. Object-oriented programming can also produce code of higher quality (e.g., Johnson, 2002), that more closely emulates real-world entities (e.g., Pennington et al., 1995). Some modern languages are also interpreted; in those languages, source code is directly executed without a separate compile step, thereby enabling interactive debugging and execution.

One such modern language is Python (van Rossum, 2008), an interpreted, object-oriented, multi-platform, open-source language used in a variety of software appli-

⏮ | ⏭

◀ | ▶

Back | Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

cations, including as a robust scientific computing platform (Oliphant, 2007). In climate studies, Python has been used as the core language for data analysis (e.g., PCMDI, 2006), visualization (e.g., Hunter and Dale, 2008), and modeling (e.g., Py-CCSM, 2008). Python's object-orientation and higher-level data structures and tools

5   (e.g., dictionaries, string and file utilities) permits numerous ways of decomposing a model into modular units. Its extensive suite of higher-level analysis tools (e.g., statistics, visualization), accessible at run time, enables modeling and analysis to occur concurrently. As an interpreted language, Python's lack of a separate compile step greatly simplifies debugging and testing, and permits changes in the program to be

10 made at run time.

Thus, while it is more difficult to write robust code in compiled languages, the code is usually very fast. Modern languages, however, while producing much more robust and stable code, exact a cost in performance. Naturally, we want the best of both worlds, both speed and simplicity: "mixed language" environments (Oliphant, 2007) are the

15 solution. In such an environment, the user-interface and calling infrastructure of the model is written in a modern language while the performance sensitive code is written in a compiled language. A wrapper generator automatically creates extension modules (as shared object libraries) of the compiled language modules, making them accessible to the modern language. A number of wrapper generator packages exist for Python,

20 including `f2py` (Peterson, 2005) which wraps Fortran modules, and SWIG (Beazley, 1997) which wraps C/C++ code.

In the present work, we describe a Python implementation of an intermediate-level atmospheric circulation model originally written in Fortran. By wrapping the Fortran code within a Python object structure, the package, `qtcm`, provides a modular and

25 interactive model where the user can alter order and choice of subroutine execution, and analyze and visualize model results, all dynamically at run time. The result is a climate modeling environment that can transform parts of the "formulate hypothesis → write and test code → run model → analyze results" sequence into a feedback loop that can be executed automatically by the computer.

A Python implementation of QTCM1

J. W.-B. Lin

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Section 2 briefly describes the the Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (QTCM1). In Sect. 3, we describe the construction of a Python implementation of QTCM1, the `qtcm` package. Section 4 gives examples of the use of the `qtcm` package, which illustrate the benefits of a mixed language environment for climate
5 modeling. We finish with discussion and conclusions in Sect. 5.

2 The Neelin-Zeng QTCM1

The QTCM1 is a primitive equation-based intermediate-level atmospheric model that focuses on simulating the tropical atmosphere (Neelin et al., 2002). Being more complicated than a simple model, the model retains full non-linearity with a basic representation of baroclinic instability, includes a radiative-convective feedback package, and
10 includes a simple land soil moisture routine (but does not include topography). The QTCM1 has been used in a variety of studies, including investigations of Madden-Julian oscillation maintenance mechanisms (Lin et al., 2000), stochastic convective parameterization (Lin and Neelin, 2000, 2002), El Niño-Southern Oscillation teleconnection patterns (Gushchina et al., 2006), and vegetation-atmosphere interactions (Zeng et al.,
15 1999).

QTCM1 differs from most full-scale general circulation models (GCMs) in that the vertical temperature, humidity, and velocity structures of the atmosphere are represented by a truncated Galerkin expansion in the vertical, instead of finite-differenced
20 pressure levels. The vertical basis functions of the expansion are chosen based on analytical solutions under convective quasi-equilibrium conditions, and thus in the tropics, where convective quasi-equilibrium effects dominate, the solution is asymptotically exact. Away from the tropics, the model behaves as a two-layer model. In principle, a Galerkin model can have any number of baroclinic basis functions accompanying its
25 barotropic basis: QTCM1 has a single baroclinic mode, and hence the "1" in its name. In the horizontal, the model discretizes the domain using a staggered Arakawa C-grid (Mesinger and Arakawa, 1976) and a default resolution of $5.625°$ longitude by $3.75°$

J. W.-B. Lin

latitude.

By using tailored vertical profiles, the QTCM1 delivers reasonable simulations of the tropics at a fraction of the computational cost of a full-scale GCM. Its relative simplicity also makes it far easier to diagnose than a full-scale GCM, potentially result-
₅ ing in greater understanding and comprehension of model results. Neelin and Zeng (2000) presents a comprehensive description of the model's formulation, and Zeng et al. (2000) describes the model's climatology. Neelin and Zeng (2000) is based upon v2.0 of QTCM1 and Zeng et al. (2000) is based on QTCM1 v2.1. The `qtcm` package is based on QTCM1 v2.3.

₁₀ **3 The Python `qtcm` package**

The `qtcm` package is an implementation of QTCM1 in a Python-based object-oriented modeling framework, using `f2py` to create extension module versions of the Fortran modules (as shared object libraries). At the package home page http://www.johnny-lin.com/py_pkgs/qtcm/, the most recent version of the full source code
₁₅ and a comprehensive user's guide is available for download. The User's Guide and source code for the version of the model described in the present work is available as a supplement at http://www.geosci-model-dev-discuss.net/1/315/2008/gmdd-1-315-2008-supplement.zip, and are covered by licenses separate from the present work. The User's Guide (Lin, 2008) provides detailed information regarding
₂₀ installing, using, troubleshooting, and adding code to the package. In the present work, we provide an overview of `qtcm`'s structure and function. Parts of this section are copied and/or adapted from Lin (2008).

3.1 Package and model structure

The `qtcm` package consists of two shared object libraries and four main submodules.
₂₅ The two shared object libraries are compiled and generated by `f2py` during pack-

A Python implementation of QTCM1

J. W.-B. Lin

age installation, and will not need to be recompiled prior to model execution. The `defaults` submodule defines various defaults for the model, the `field` submodule defines class `Field` (described later), the `plot` submodule defines routines used for quick visualization of model results, and the `qtcm` submodule defines the class `Qtcm`

5 (also described later).

A model in the `qtcm` package is defined as an instance of the class `Qtcm`. Because the `qtcm` package wraps Fortran routines with a Python layer, there are two types of variables associated with `Qtcm` model instances: those defined at the Python-level and those defined at the Fortran-level. Some variables, while defined separately at both the

10 Python and Fortran levels (i.e., they do not share the same memory space), have the same names and functions in both levels of the model. Those variables are known as "field variables" and are considered to be defined at both the Python-level and Fortran-level (an example of such a variable is Q_c, the precipitation). `Qtcm` instances have public methods (`get_qtcm1_item` and `set_qtcm1_item`) for passing the values of

15 field variables back-and-forth between the Python and Fortran levels.

All field variables and most model parameters (such as time step, input and output directory names, etc.) are instances of the `Field` class. A `Field` instance stores the value of the variable in an attribute named `value`, and metadata related to the variable as other instance attributes. If the value of the field variable is an array, the

20 value stored in the attribute `value` is a NumPy (van der Walt, 2008) array. Only the value of a `Field` instance can be passed to its Fortran counterpart (when it exists), because standard Fortran variables cannot hold metadata. All model parameters are attributes of `Qtcm` instances. Field variables, at the Python-level, are also `Qtcm` instance attributes. Model parameter and field variable values can be passed into the

25 model instance on instantiation via the input keyword parameter list, or set after instantiation by changing the instance attribute. If these parameters and variables are not set manually, they are set to default values given in the `defaults` submodule. Table 1 lists the key instance attributes and methods for the `Field` and `Qtcm` classes.

3.2 Creating a model instance and running the model

Figure 1 shows a simple example of a model instance being created and run. Model instances are created using standard Python syntax; in Fig. 1, `model = Qtcm(**inputs)` creates a model instance `model`. In this example, we make use of a feature in Python where keyword parameter argument lists can be passed in as a dictionary, where the dictionary's keys correspond to the names of the keyword parameters, and the associated value in the dictionary corresponds to the input value of the keyword parameter; the variable `inputs` is such a dictionary. Based on the values of `inputs` shown in Fig. 1, the model instance is configured to make an aquaplanet run (set by `landon`), starting from November 1, Year 1 (set by `year0`, `month0`, and `day0`), running for 30 days (set by `lastday`) from a newly initialized model state (set by `mrestart`). The model's netCDF (Unidata, 2007) output filenames will contain the string given by `runname`. By default, the model uses climatological sea-surface temperatures (SST) for the lower-boundary forcing over the ocean.

The keyword `compiled_form` defines which of the two types of Fortran extension modules, derived from the Fortran QTCM1 code, the model instance will link to. The first type permits very little control over the compiled Fortran routines at the Python level, and is selected by setting `compiled_form = 'full'`. The second allows a user, from the Python-level, to control model execution in the Fortranlevel all the way down to the atmospheric timestep level. This extension module is selected by setting `compiled_form = 'parts'`. In general, most users will set `compiled_form = 'parts'`, and thus we assume this setting for the rest of the present work. See Lin (2008) for details about this keyword.

Once the model has been instantiated, running the model requires just a call of the `run_session` method. In most cases, no input parameters need to be passed at this call. In Fig. 1, this is given in the last line. At the beginning and end of the `run_session` call, the values of all field variables at the Python and Fortran levels are synchronized to match each other.

3.3 Run sessions

Once we instantiate and configure a model instance, we can use the instance for any number of runs. We call each of these runs using the same model instance a "run session." In a run session, the model is run from day 1 of simulation to the day specified by the `lastday` attribute. A run session is a "complete" model run, at the beginning of which all Fortran-level field variables are set to the values given at the Python-level, and at the end of which restart files are written, the values at the Python-level are overwritten by the values from the Fortran-level, and a Python-accessible snapshot is taken of the model variables that were written to the restart file.

Before and after a run session, model variables are easily accessed from the Python level, and can be changed at will just by changing the value of the pertinent model instance attribute. The new values can then be used at the next run session of the model instance. To continue a second run session after an initial run session, set the keyword parameter `cont` in the input list of a `run_session` method call.

Figure 2 gives an example of two run sessions, where the second run session is a continuation of the first, and with changes made to a field variable between the two run sessions. The first run session lasts 10 days, and is given by the setting of the `lastday` keyword parameter. Between these run sessions, the value of field variable `u1` (the zonal wind associated with the first baroclinic mode) is doubled, and this doubled value is used in the second run session. The second run session lasts 30 days.

The change between the two run sessions in the simple example given in Fig. 2 is uninteresting, but the example illustrates how the `qtcm` Python framework opens up possibilities of interactive analysis with the model. Because Python is an interpreted language, the code in Fig. 2 does not have to be written in a file, compiled, linked, and executed; the code can be typed in during run time. Between run sessions, we can conduct and visualize more complex analyses of the model, and use the results of those analyses to change the model configuration for the next run session. And since Python is a complete programming language, we can also automate these analyses,

A Python implementation of QTCM1

J. W.-B. Lin

|◄   ►|

◄   ►

Back   Close

Full Screen / Esc

[Printer-friendly Version](#)

[Interactive Discussion](#)

without leaving the modeling environment. The important benefits of this feature are described in Sect. 4.

3.4 Passing restart snapshots between run sessions

Sometimes, we want to branch a number of model runs from the same starting point. The QTCM1 writes restart files for that purpose, and a `Qtcm` instance can also make use of those files by setting the `mrestart` attribute accordingly. This restart mechanism is straightforward to use, but becomes difficult to manage when many restart files are involved.

The `qtcm` package provides a way to take, store, and apply restart snapshots at the Python-level, by storing a snapshot as a dictionary. At the end of a run session, a snapshot of the model state is automatically taken and stored as the instance attribute `snapshot`. You can store this attribute as another Python variable for later use. Figure 3 shows an example of saving the model snapshot as the variable `mysnapshot`, and using that snapshot to initialize a later run session. The method `sync_set_py_values_to_snapshot` initializes the model to the values of `mysnapshot`, and setting the attribute `init_with_instance_state` to `True` prior to calling `run_session` the second time will force the model to use the current instance state as the run session's initial values.

3.5 Creating multiple models

Creating multiple QTCM1 Fortran models requires maintaining and operating on different sets of source code, as well as compiling each set of source code separately to obtain the desired multiple executables. With the `qtcm` package, creating multiple models is as easy as instantiating multiple `Qtcm` instances. For instance, to create two Python QTCM1 models, `model1` and `model2`, just enter in the following:

```
from qtcm import Qtcm
model1 = Qtcm(**inputs1)
model2 = Qtcm(**inputs2)
```

where `inputs1` and `inputs2` are separate dictionaries specifying the input keyword parameters. Note that `model1` and `model2` do *not* have any variables in common, including in the extension modules holding the Fortran code, and thus the two
5 instances are two truly independent models. Each instance automatically links to a separate copy of the extension modules, which are saved in temporary directories.

3.6 Passing restart snapshots between multiple models

In Sect. 3.4, we saw how a model snapshot can be saved to a separate variable and used to initialize a later run session. Of course, since `mysnapshot` is an independent
10 dictionary, we are not limited to using it only with the model instance the snapshot originally came from. Figure 4 shows an example of using a snapshot to initialize run sessions in multiple models.

3.7 Run lists

Of all the features the Python infrastructure enables us to create in our wrapping of the
15 QTCM1 model, run lists may be the most valuable. A run list in the `qtcm` package is a Python list that specifies a series of Python or Fortran methods, functions, subroutines (or other run lists) that will be executed when the list is passed into a call of the `Qtcm` instance method `run_list`. Since routines in run lists are identified by strings (instead of, for instance, as a memory pointer to a library archive object file), and Python lists are mutable, run lists are full changeable at run time. As a result, what routines the
20 model executes are also fully changeable at run time.

Run lists are stored in a dictionary set to the `Qtcm` instance attribute `runlists`. The dictionary key for the run list's entry is the run list name. Figure 5 shows an interactive Python session that prints out the contents of run list `'atm_physics1'`. This run list

|◄   ►|

◄   ►

Back   Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

specifies the set of routines used to execute to calculate atmospheric physics at one instant in time. Each entry of the list is a string and refers to the name of the wrapped Fortran routine that calculates moist convection, cloud effects, shortwave radiative flux, longwave radiative flux, and surface fluxes, respectively. To change the order of the
5 calculation, or to add, delete, or replace the routines being called, just change the elements of the list using any of the list methods provided by Python (e.g., `append`).

In Fig. 5, all the routines given in the run list are Fortran subroutines and require no parameters to be passed in via an argument list. Run lists can, however, specify Python functions and methods and other run lists. For both Python and Fortran routines, the
10 run list feature can also accommodate routines that have argument lists. Figure 6 shows the run list for initializing the atmospheric portion of the model. The first two routines executed by the run list are Fortran subroutines without any input parameters. The third is the `Qtcm` instance method `varinit`, also without input parameters in the calling argument list. The fourth element of the run list is a Fortran subroutine, but with
15 one input parameter in its calling argument list. The final routine is not a routine at all, but another run list. Regardless of what kind of routine or run list is specified, the specification is still the same: a string or a one-element dictionary with a string as the key. Lin (2008) gives details about run lists.

3.8 Output and visualization

20 The `Qtcm` model instance writes instantaneous and mean output to netCDF files. The netCDF data format is a platform independent binary format that permits metadata to be saved with the data. There are a number of packages for Python that can read and manipulate netCDF data, such as the Climate Data Analysis Tools (PCMDI, 2006).

The Matplotlib package (Hunter and Dale, 2008) for Python generates 1-D and 2-D
25 plots using Matlab-like syntax. `Qtcm` instances have a method `plotm` which reads the netCDF output files and uses Matplotlib to create line or contour plots of user-specified slices of the data. Figure 7 shows an interactive modeling session with the `qtcm` package where the user has created visualizations of a variety of parameters at

A Python implementation of QTCM1

J. W.-B. Lin

Printer-friendly Version

Interactive Discussion

run time.

3.9 Model performance

Because the model's core numerics are written in Fortran, with Python providing a sophisticated programmer/user-interface, the performance penalty of the `qtcm` package (with `compiled_form = 'parts'`), compared to the pure-Fortran QTCM1 is approximately 4–9% (the penalty for `compiled_form = 'full'` is less). Table 2 gives wall-clock values for `qtcm` running on two platforms, Mac OS X and Ubuntu GNU/Linux.

4 Example uses of the `qtcm` package

By wrapping the Fortran QTCM1 with a Python layer, the `qtcm` package permits us to accomplish science tasks that would otherwise require a labyrinthine set of shell scripts, temporary input and output files, and source code versions. In this section, we describe a few such science tasks to illustrate what the Python wrapping buys us. The examples in this section are taken from Lin (2008).

4.1 Conditionally explore parameter space

Figure 8 provides an example of code that explores different values of mixed-layer depth (`ziml`) over a set of 30 day runs, as a function of maximum zonal wind associated with the first baroclinic mode (`u1`) magnitude, until it finds a case where the maximum of `u1` is greater than 10 m/s. (The relationship between `ziml` and the maximum of the speed of `u1`, where `ziml = 0.1 * maxu1`, is made up.) With each iteration, the new run uses the snapshot from a previous run as its initialization (as well as the new value of `ziml`); the `try` statement is used to ensure the model works even if `mysnapshot` is not defined (which is the case the first time around).

If we implemented this science task using the pure-Fortran QTCM1 and shell scripts, we would probably have to write a separate program (possibly in a separate data anal-

⏮ | ⏭

◀ | ▶

Back | Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

ysis language like IDL, Matlab, or NCL) to analyze model output. Required parameters might be passed through an operating system pipe, or through namelists and temporary files. Automating modeling with analysis in such an environment can be difficult, limited, and error prone. The `qtcm` package allows us to take advantage of Python's numerical computing capabilities so that we can embed our traverse of parameter space within a `while` loop, thus automating the analysis task within the modeling environment.

4.2 Test alternative parameterizations

Figure 9 demonstrates the following scenario: Assume we have nine different cloud physics schemes we wish to test in nine different runs. The easiest way to do this is to take advantage of Python's object-oriented inheritance capabilities, creating a new class `NewQtcm` that inherits everything from `Qtcm`, and to which we add the additional cloud schemes (`cloud0`, `cloud1`, etc.). In the `for` loop in Fig. 9, we change the cloud model run list entry in the `'atm_physics1'` run list to whatever the cloud model is at this point in the loop.

Of course, we could do the same thing by running the nine models separately, but this set-up makes it easy to do hypothesis testing between these nine models as the models are running. For instance, we can create a test by which we will choose which of the nine models to use: Within this framework, the selection of those models can be altered by changing a string. If the same task were implemented with shell scripts and makefiles, we would have to write our own selector routines (perhaps using file system functions) for selecting model(s) from amongst the possible executables. It is much easier to use Python's built-in string manipulation routines.

5   Discussion and conclusions

In the present work, an intermediate-level atmosphere model written in Fortran is wrapped with an object-oriented structure written in Python, which makes modern data abstraction utilities available to a model written in a traditional procedural language.
The result is a model that can be used dynamically at run time, with the user able to change the order of subroutine execution at will, and able to analyze model results within the modeling environment.

This flexibility, however, provides more than just convenience for the user. The `qtcm` package's run time-interactive tools, and tools like them, can transform the traditional analysis sequence used in modeling studies into a sequence with more capabilities. The traditional sequence begins with formulation of a hypothesis, then leads to implementing a test of the hypothesis in model code, making model runs using the coded test, and ends with analyzing the model results using various statistical and visualization packages (Fig. 10a). Some transitions between the various steps mainly make use of human input (e.g., from hypothesis to code), while others combine human reasoning with computational tools (e.g., we can mostly automate the transition from code to model runs through the use of makefiles combined with shell scripts). The feedback part of the cycle, where analysis of the results modifies the original hypothesis, usually requires human input.

In contrast, the tools provided by `qtcm` and similar packages open up the potential to automate substantially larger portions of the analysis sequence. Figure 10b shows a schematic of how model analysis might be transformed. Instead of being limited to a few hypotheses, the transformed sequence makes additional types of hypothesis accessible without changing the complexity of the code required (see Sect. 4's examples as illustrations). Most importantly, the Fig. 10b sequence enables model output analysis to automatically control future model runs. Instead of requiring human intervention to determine future model runs, the computer can make that evaluation, and as a result, for the same complexity of code, we can more intelligently explore the problem's

solution space.

Thus, though the use of mixed language programming environments for climate modeling has a modest cost in performance, these environments have the potential the pay back substantial dividends in code simplicity, reliability, and ease-of-use. More importantly, such an environment, by providing a robust programming interface with capabilities traditional languages cannot easily support, gives researchers the tools to investigate previously inaccessible (or difficult to access) questions. The wrapping techniques illustrated in the present study for the Neelin-Zeng QTCM1 may be fruitfully deployed to other climate models, increasing their flexibility and scientific usefulness.

References

Beazley, D. M.: SWIG 1.1 Users Manual, http://www.swig.org/Doc1.1/HTML/Contents.html, 1997. 318

Edwards, P. N.: A brief history of atmospheric general circulation modeling, in: General Circulation Development, Past Present and Future: The Proceedings of a Symposium in Honor of Akio Arakawa, edited by: Randall, D. A., Academic Press, New York, 67–90, 2000. 316

Gushchina, D., Dewitte, B., and Illig, S.: Remote ENSO forcing versus local air-sea interaction in QTCM: A sensitivity study to intraseasonal variability, Adv. Geosci., 6, 289–297, 2006, http://www.adv-geosci.net/6/289/2006/. 319

Hunter, J. and Dale, D.: The Matplotlib User's Guide, http://matplotlib.sourceforge.net/users/index.html, 2008. 318, 326

A Python implementation of QTCM1

J. W.-B. Lin

|◄ | ►|

◄ | ►

Back | Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Johnson, R. A.: Object-oriented analysis and design — What does the research say?, J. Comput. Inform. Syst., 42, 11–15, 2002. 317

Lin, J. W.-B.: qtcm User's Guide, http://www.johnny-lin.com/py_pkgs/qtcm/doc/manual.pdf, 2008. 320, 322, 326, 327, 334

5 Lin, J. W.-B. and Neelin, J. D.: Influence of a stochastic moist convective parameterization on tropical climate variability, Geophys. Res. Lett., 27, 3691–3694, 2000. 319

Lin, J. W.-B. and Neelin, J. D.: Considerations for stochastic convective parameterization, J. Atmos. Sci., 59, 959–975, 2002. 319

Lin, J. W.-B., Neelin, J. D., and Zeng, N.: Maintenance of tropical intraseasonal variability:
10 Impact of evaporation-wind feedback and midlatitude storms, J. Atmos. Sci., 57, 2793–2823, 2000. 319

Mesinger, F. and Arakawa, A.: Numerical Methods Used in Atmospheric Models, Vol. 1, GARP Publications Series No. 17, World Meteorological Organization, 1976. 319

Neelin, J. D. and Zeng, N.: A quasi-equilibrium tropical circulation model – formulation, J.
15 Atmos. Sci., 57, 1741–1766, 2000. 320

Neelin, J. D., Zeng, N., Chou, C., Lin, J., Su, H., Munnich, M., Hales, K., and Meyerson, J.: The Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (QTCM1), Version 2.3, UCLA Department of Atmospheric Sciences, Los Angeles, http://www.atmos.ucla.edu/~csi/qtcm_man/v2.3/qtcm_manv2.3.pdf, 2002. 319

20 Oliphant, T. E.: Python for scientific computing, Comput. Sci. Eng., 9, 10–20, 2007. 318

PCMDI: Climate Data Analysis Tools, http://cdat.sf.net, 2006. 318, 326

Pennington, N., Lee, A. Y., and Rehder, B.: Cognitive activities and levels of abstraction in procedural and object-oriented design, Hum.-Comput. Interact., 10, 171–226, 1995. 317

Peterson, P.: F2PY Users Guide and Reference Manual, http://cens.ioc.ee/projects/f2py2e/
25 usersguide/index.html, 2005. 318

PyCCSM: pyccsm: A Python version of the CCSM coupler, http://code.google.com/p/pyccsm/, 2008. 318

Unidata: The NetCDF Tutorial, Boulder, CO, http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial.html, 2007. 322

30 van der Walt, S.: Documentation: NumPy and SciPy, http://www.scipy.org/Documentation, 2008. 321

van Rossum, G.: Python Tutorial: Release 2.5.2, Python Software Foundation, http://www.python.org/doc/2.5.2/tut/tut.html, 2008. 317

A Python implementation of QTCM1

J. W.-B. Lin

Title Page

Abstract | Introduction

Conclusions | References

Tables | Figures

◄◄ | ►►

◄ | ►

Back | Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Zeng, N., Neelin, J. D., Lau, K.-M., and Tucker, C. J.: Enhancement of interdecadal climate variability in the Sahel by vegetation interaction, Science, 286, 1537–1540, 1999. 319

Zeng, N., Neelin, J. D., and Chou, C.: A quasi-equilibrium tropical circulation model— implementation and simulation, J. Atmos. Sci., 57, 1767–1796, 2000. 320

A Python implementation of QTCM1

J. W.-B. Lin

|◄ | ►|

◄ | ►

Back | Close

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

A Python implementation of QTCM1

J. W.-B. Lin

Table 1. Key public instance attributes and methods for `Field` and `Qtcm` instances. Note that for `Qtcm` instances, field variables are also attributes, with attribute names corresponding to the ids of the fields.

Class	Type	Name and Description
`Field`	Attributes	`id`: A string naming the model parameter or field variable (e.g., "Qc", "mrestart").
		`value`: The value of the field.
		`units`: A string giving the units of the field.
		`long_name`: A string giving a description of the field.
	Methods	`rank`: Returns the rank of `value`.
		`typecode`: Returns the typecode of `value`.
`Qtcm`	Attributes	`compiled_form`: Describes the form of the compiled Fortran version of the QTCM1 model.
		`coupling_day`: Current value of the atmosphere-ocean coupling day.
		`init_with_instance_state`: Initialize run session with the `Qtcm` model instance state.
		`runlists`: Lists of methods and other run lists that can be executed by the `run_list` method.
		`sodir`: Name of temporary directory containing shared object files for this `Qtcm` instance.
	Methods	`get_qtcm1_item`: Get field from the compiled QTCM1 model.
		`make_snapshot`: Make copy of the current state of the run session's variables.
		`plotm`: Plot mean output for a given model field.
		`qtcm`: Run the atmosphere over a coupling interval step.
		`run_list`: Run run list(s) and/or instance methods.
		`run_session`: Run a model run session.
		`set_qtcm1_item`: Set Python-accessible compiled QTCM1 model fields.
		`sync_set_py_values_to_snapshot`: Set Python attributes to a previous `make_snapshot` output.
		`varinit`: Initialize model variables in a run session.

J. W.-B. Lin

Table 2. Wall-clock times (sec) for the average of three 365 day aquaplanet runs using climatological sea surface temperature as the lower boundary forcing (Lin, 2008). The "Pure" column refers to runs using the pure-Fortran QTCM1, while "Wrap" refers to the Python wrapped `qtcm` package (v0.1.1) with `compiled_form = 'parts'`.

System	Pure	Wrap
Mac OS X: MacBook 1.83 GHz Intel Core Duo running Mac OS X 10.4.10.	152.59	158.94
Ubuntu GNU/Linux: Dell PowerEdge 860 with 2.66 GHz Quad Core Intel Xeon processors (64 bit) running Ubuntu 8.04.1 LTS.	43.73	47.45

A Python implementation of QTCM1

J. W.-B. Lin

```
from qtcm import Qtcm
inputs = {}
inputs['runname'] = 'test'
inputs['landon'] = 0
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 30
inputs['mrestart'] = 0
inputs['compiled_form'] = 'parts'
model = Qtcm(**inputs)
model.run_session()
```

Fig. 1. A simple `qtcm` run.

```
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 10
inputs['mrestart'] = 0
inputs['compiled_form'] = 'parts'

model = Qtcm(**inputs)
model.run_session()
model.u1.value = model.u1.value * 2.0
model.init_with_instance_state = True
model.run_session(cont=30)
```

Fig. 2. An example of two `qtcm` run sessions where the second run session is a continuation of the first. Assume `inputs` is a dictionary as in Fig. 1, and that earlier in the script the run name and all input and output directory names were added to the dictionary.

```
model.run_session()
mysnapshot = model.snapshot
model.sync_set_py_values_to_snapshot(snapshot=mysnapshot)
model.init_with_instance_state = True
model.run_session()
```

Fig. 3. An example of using a snapshot from one `qtcm` run session as the restart for a second run session.

```
model.run_session()
mysnapshot = model.snapshot
model1.sync_set_py_values_to_snapshot(snapshot=mysnapshot)
model2.sync_set_py_values_to_snapshot(snapshot=mysnapshot)
model1.run_session()
model2.run_session()
```

Fig. 4. An example of using a snapshot from one `qtcm` run session as the restart for run sessions in multiple other model instances.

```
>>> from qtcm import Qtcm
>>> model = Qtcm(compiled_form='parts')
>>> print model.runlists['atm_physics1']
['_qtcm.wrapcall.wmconvct', '_qtcm.wrapcall.wcloud',
'_qtcm.wrapcall.wradsw',
'_qtcm.wrapcall.wradlw', '_qtcm.wrapcall.wsflux']
```

Fig. 5. Contents of run list `'atm_physics1'`, the set of routines to execute to calculate atmospheric physics at one instant in time, as displayed during a Python interpreter session.

```
>>> from qtcm import Qtcm
>>> model = Qtcm(compiled_form='parts')
>>> print model.runlists['qtcminit']
['__qtcm.wrapcall.wparinit', '__qtcm.wrapcall.wbndinit', 'varinit',
{'__qtcm.wrapcall.wtimemanager':  [1]}, 'atm_physics1']
```

Fig. 6. Contents of run list `'qtcminit'`, the set of routines to execute to initialize the atmospheric portion of the model, as displayed during a Python interpreter session.
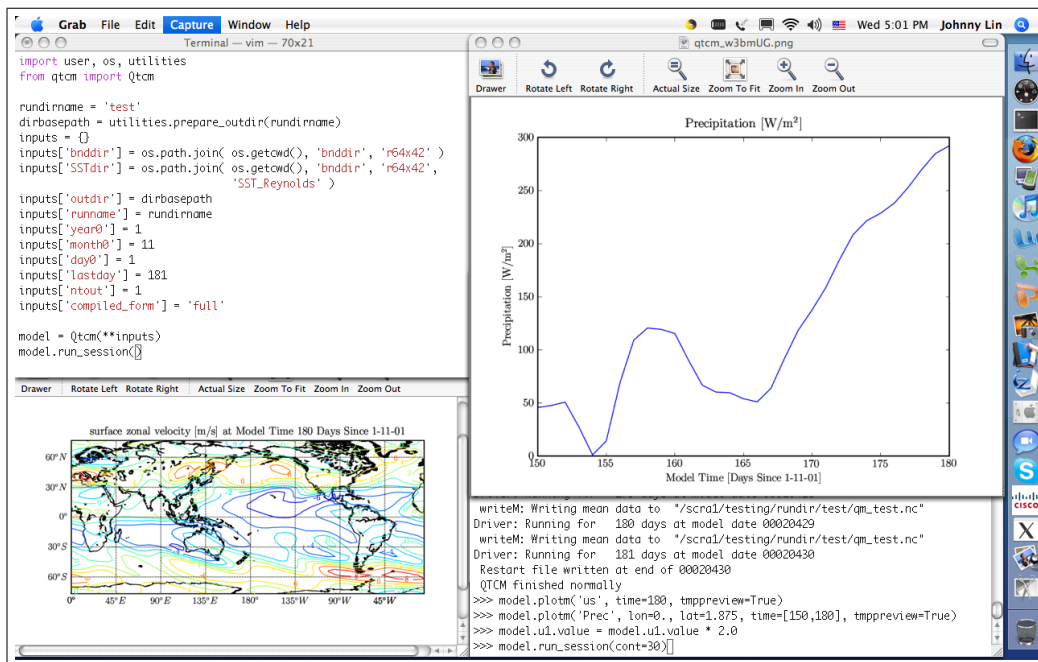
Fig. 7. Screenshot of an interactive modeling session using the `qtcm` package. The upper-left panel shows the source code file specifying the run. The lower-right panel shows the Python interpreter session making the run. The two plot windows display the plots generated by the `plotm` calls from the Python interpreter command line.

```
import os
import numpy as N
maxu1 = 0.0
while maxu1 < 10.0:
    iziml = 0.1 * maxu1
    iname = 'ziml-' + str(iziml) + 'm'
    ipath = os.path.join('proc', iname)
    os.makedirs(ipath)
    model = Qtcm(**inputs)
    try:
        model.sync_set_py_values_to_snapshot(snapshot=mysnapshot)
        model.init_with_instance_state = True
    except:
        model.init_with_instance_state = False
    model.ziml.value = iziml
    model.runname.value = iname
    model.outdir.value = ipath
    model.run_session()
    maxu1 = N.max(N.abs(model.u1.value))
    mysnapshot = model.snapshot
    del model
```

Fig. 8. Example of an exploration of the effects of different values of mixed-layer depth. The `inputs` dictionary is initialized similarly as in Fig. 1.

```
import os

class NewQtcm(Qtcm):
    def cloud0(self):
        [...]
    def cloud1(self):
        [...]
    def cloud2(self):
        [...]
    [...]

inputs['init_with_instance_state'] = False
for i in xrange(10):
    iname = 'cloudroutine-' + str(i)
    ipath = os.path.join('proc', iname)
    os.makedirs(ipath)
    model = NewQtcm(**inputs)
    model.runlists['atm_physics1'][1] = 'cloud' + str(i)
    model.runname.value = iname
    model.outdir.value = ipath
    model.run_session()
    del model
```

Fig. 9. Example of using inheritance in Python to explore the effects of multiple cloud physics schemes in multiple runs. The `[...]` denote the code of the different (hypothetical) cloud physics schemes. The `inputs` dictionary is defined similarly as in Fig. 1.
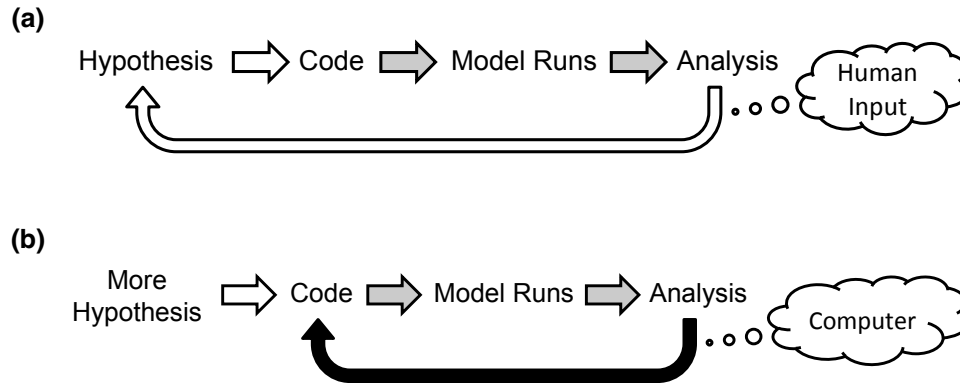
(a)

(b)

Fig. 10. Schematic of **(a)** the traditional analysis sequence used in modeling studies, and **(b)** the transformed analysis sequence using `qtcm`-like modeling tools. Outlined arrows with no fill represent mainly human input. Gray-filled arrows represent a mix of human and computer-controlled input. Completely filled (black)-arrows represent purely computer-controlled input.