Geosci. Model Dev. Discuss., 1, S128–S132, 2008
www.geosci-model-dev-discuss.net/1/S128/2008/

Geoscientific
Model Development
Discussions

Interactive comment on "qtcm 0.1.2: A Python Implementation of the Neelin-Zeng Quasi-Equilibrium Tropical Circulation model" *by* J. W.-B. Lin

J. W.-B. Lin

Received and published: 21 December 2008

Many thanks to the referee for his constructive and helpful review! His comments are greatly appreciated. Below I address his comments in the order given. My itemization follows his bullet points. In this reply, I use the past tense "changed," "added," etc. to describe my corrections. However, I will not upload a revised draft of the paper with these changes until after the comment period. Thus, I have copied the most substantial of those changes into this reply, so the referee can review them.

General comments:

- I agree with the reviewer that a complete use case in Fig. 10 would make stronger my case regarding the potential for changing the manner of addressing scientific problems. At the same time, I think that the examples I give do make the case I claim, that the tool types prototyped in the `qtcm` package offer the *potential* of automating parts of the analysis step in scientific modeling. In this way, I believe Fig. 10 is a reasonable inference from the previous examples, and in the conclusion make explicit reference to those examples as justification (see the parenthetical statement "see Sect. 4's examples as illustrations" in the Fig. 10 discussion).

- I think the reviewer's question regarding the possibilties of applying the framework to a comprehensive GCM is a great one, and the logical next step. But how to make a simple framework work in a GCM is, I think, a difficult question. Large infrastructure projects like ESMF are one approach, but despite their benefits, are quite difficult to use. The current work represents another approach, and by prototyping it on an intermediate-level model, hopes to contribute something to the larger problem. It is, however, only a first step, and a meaningful assessment of applying the methods in the current work to GCMs is an entire paper (or set of papers) in themselves.

- A detailed description of the structure of the Fortran code is available in Neelin et al. (2002), referenced in the text. I've added the following sentence in the last paragraph of the section describing the Neelin-Zeng QTCM1:

  > A detailed manual (Neelin et al. 2002) describes the structure of the Fortran code.

  Model performance, and the penalty due to the wrapping, is discussed in the final subsection of section 3, and Table 2 gives wall-clock values for running the pure-Fortran QTCM1 and the `qtcm` package.

- This is a great suggestion, and adding multi-threading capabilities is one of my priorities for package development. Parallelism via MPI, however, is something intermediate-level developers want to avoid using (Fortran QTCM1, for instance, has no MPI calls); that design choice makes the code much easier to understand and diagnose. Intermediate-level models can get away with eschewing parallelism because they run so fast; a model year runs on a PC running Linux in 45 seconds. To help clarify, however, I added the sentence "all runs are executed as single threads" to the Table 2 caption.

Specific comments:

- I changed the sentence mentioned to:

  > Being mainly a procedural language, Fortran has traditionally lacked the default programming structures to organize a model into truly self-contained units, thus limiting modularity.

  in order to make the sentence more accurate.

- I've added a short introduction to object-oriented programming as section 3.1. The text is below:

  > Because Python is an object-oriented language, the fundamental programming unit is not the subroutine, but rather is the "object." In a procedural language, data and functions that operate on data are two separate entities. In an object-oriented language, these two entities are bound together in a single construct, the object. Because of this framework, functions are automatically considered in context with the data they operate on, and vice versa. This lessens the risk of errors that

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Discussion Paper

occur when data is manipulated by functions that were never intended to be used on that kind of data.

Data bound to an object are called “attributes” of that object, and functions that operate on that data are called “methods” of that object. In Python, the attributes of an object are specified by a name that comes after a period at the end of the object name. Thus, `model.runname` refers to the `runname` attribute of the `model` object. Methods are similarly named; however, to call a method, a parameter list (even if empty) must be specified. Thus, `model.run_session()` calls the `run_session` method bound to the `model` object.

In general, Python objects consist of two types of attributes and methods: public and private. Public attributes and methods are accessible to the general user. Private attributes and methods, on the other hand, are designed to be accessed only by developers. In Python, private attributes and methods have names prepended by one or two underscores.

Objects are created from a “template” that defines the attributes and methods that go into that object. The template is known as a “class,” and individual objects that are derived from a class are called “instances” of that class. Creating an object that is an instance of a class is known as “instantiating” the object. In the example above, `model` is an instance of the `Qtcm` class, which defines the `runname` attribute and `run_session` method. There is no limit to the number of instances of a class, and all instances of a class have equal access to the attributes and methods defined by the class.

Python’s highest level of organization is the package, a library of related modules. Modules, in Python, are individual files that define related objects, functions, and variables, and thus a package is a directory of module files. A single module can contain an unlimited number of objects, functions, and variables.

Regarding metadata, I've added the parenthetical statement "e.g., units, long name, etc." after "metadata" to help clarify.

- Yes, you can read Fortran-level variables, process them at the Python-level using CDAT and other tools, and send values back to the Fortran-level, all at run time. I retitled (the old) section 3.8 to include discussion about this kind of analysis and have added this paragraph:

> Because the `qtcm` package makes the Fortran-level variables accessible from the Python level, the user can use any analysis tools at the Python-level on data from those Fortran-level variables, in addition to the netCDF output, and send the values as desired back to the Fortran-level, all during run time. This enables the user to utilize the powerful analysis tools provided by the Climate Data Analysis Tools, SciPy (van der Walt 2008), and other Python packages, during as well as after run time.

Thanks again to the reviewer for all his help! In appreciation, I've added his name to the acknowledgments section.

Interactive comment on Geosci. Model Dev. Discuss., 1, 315, 2008.