# qtcm User's Guide

Johnny Wei-Bing Lin[1]

September 12, 2008

[1]Physics Department, North Park University, 3225 W. Foster Ave., Chicago, IL 60625, USA

# Contents

# Chapter 1

# Introduction

## 1.1 How to Read This Manual

**Most users:** Just read (1) the installation instructions in Chapter 2, (2) Chapter 3, which tells you all you need to get started using `qtcm`, and (3) examples in Section 4.10 that give a feel for how you can use the model.

**Users having problems:** Chapter 5 provides troubleshooting tips for a few problems. The detailed description of how the package functions, in Chapter 4, will probably be more useful.

**Developers:** If you want to change the source code, please read Chapter 6. Chapter 7 describes all the things I'd like to do to improve the package, but haven't gotten to yet.

## 1.2 About the Package

The single-baroclinic mode Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (QTCM1)[1] is a primitive equation-based intermediate-level atmospheric model that focuses on simulating the tropical atmosphere. Being more complicated than a simple model, the model has full non-linearity with a basic representation of baroclinic instability, includes a radiative-convective feedback package, and includes a simple land soil moisture routine (but does not include topography). A brief, but more detailed, description of QTCM1 is given in Section 1.6.

Python[2] is an interpreted, object-oriented, multi-platform, open-source language that is used in a variety of software applications, ranging from game development to bioinformatics. In climate studies, Python has been used as the core language for data analysis (e.g., Climate Data Analysis Tools[3]), visualization (e.g., Matplotlib[4]),

[1]http://www.atmos.ucla.edu/∼csi

[2]http://www.python.org

[3]http://cdat.sf.net

[4]http://matplotlib.sf.net

and modeling (e.g., PyCCSM[5]).

In comparison to traditional compiled languages like Fortran, Python's lack of a separate compile step greatly simplifies the debugging and testing phases of development, because code snippets can be testing as code is written. Python's extensive suite of higher-level tools (e.g., statistics, visualization, string and file manipulation) accessible at runtime enables modeling and analysis to occur concurrently.

The `qtcm` package is an implementation of the Neelin-Zeng QTCM1 in a Python object-oriented environment. The conversion package `f2py`[6] is used to wrap the QTCM1 Fortran model routines and manage model execution using Python datatypes and utilities. The result is a modeling package where order and choice of subroutine execution can be altered at runtime. Model analysis and visualization can also be integrated with model execution at runtime.

## 1.3 Conventions In This Manual

### 1.3.1 Audience

In this manual I assume you have a rudimentary knowledge of Python. Thus, I do not describe basic Python data types (e.g., dictionaries, lists), object framework and syntax (e.g., classes, methods, attributes, instantiation), module and package importing. If you need to brush up (or learn) Python, I'd recommend the following resources:

- Python Tutorial:[7] This tutorial was written by Guido van Rossum, Python's original author.
- Instant Hacking:[8] Learn how to program with Python.
- Dive Into Python:[9] Reasonably complete and cohesive. The entire book is available for free online.
- Handbook of the Physics Computing Course:[10] Written for a science audience. Recommended.
- CDAT/Python Tips for Earth Scientists:[11] This web site is a FAQ of sorts for people using Python and the Climate Data Analysis Tools (CDAT) in the earth sciences, and thus focuses on using Python to do science rather than the computer science aspects of the language.

[5]http://code.google.com/p/pyccsm/
[6]http://cens.ioc.ee/projects/f2py2e/
[7]http://docs.python.org/tut/
[8]http://www.hetland.org/python/instant-hacking.php
[9]http://diveintopython.org/index.html
[10]http://www.pentangle.net/python/handbook/
[11]http://www.johnny-lin.com/cdat_tips/

The purpose of this package is to make the QTCM1 model easier to use. In order to interpret the results, however, you still need to have a robust sense of what climate models can and cannot tell you. A starting point for the QTCM1 model is the brief description of the model in Section 1.6. After that, I would read the original papers describing the model formulation and results [3, 5], and papers citing the model formulation work.[12] Being an intermediate-level model using the quasi-equilibrium assumption, QTCM1 (and thus `qtcm`) has distinct strengths and limitations; please be aware of them.

## 1.3.2 Typographic Conventions

| | |
|---|---|
| `commands` | to be typed at the command-line are rendered in a blue, serif, fixed-width typewriter font (e.g., `make _qtcm_full_365`). |
| *dummy arguments* | coupled with code or screen display is rendered in a serif, proportional, italicized font (e.g., `Error-Value too long in` *variable*). |
| *file and directory names* | are rendered in a sans-serif, italicized font (e.g., *setbypy.F90*). |
| `screen display` | is rendered in a green, serif, fixed-width typewriter font. |
| `module, method, and subroutine names` | are rendered in a blue, serif, fixed-width typewriter font. |
| `variable and attribute names` | are rendered in a blue, serif, fixed-width typewriter font. |
| `class names` | are rendered in a blue, serif, fixed-width typewriter font. |

Blocks of code (usually commands, screen display, and module, variable, and class names) are displayed in a blue, serif, fixed-width typewriter font.

## 1.3.3 Terminology

**attribute and method references:** If there is any possibility of confusion, I will give the class that an attribute or method comes from when that attribute or method is referenced. If no class is mentioned by name or context, assume that the attribute/method comes from the `Qtcm` class.

**"compiled QTCM1 model":** This usually is used to denote when I'm talking about compiled Fortran QTCM1 routines and variables therein, as an extension module to the Python `qtcm` package.. Thus, "compiled QTCM1 model `u1`" is

[12]http://scholar.google.com/scholar?hl=en&lr=&cites=14217886709842286738

the value of variable `u1` in the Fortran model, not the value at the Python-level. Sometimes I refer to the compiled QTCM1 model as just "QTCM1" or as "compiled QTCM1 Fortran model".

**"pure-Fortran QTCM1":** This refers to the Neelin-Zeng QTCM1 model in it's original Fortran form, not as an extension module to the Python `qtcm` package.

**"Python-level":** This usually denotes the value of a variable as an attribute of a `Qtcm` instance. This variable is stored at the Python interpreter level.

**`Qtcm`:** This refers to the Python `Qtcm` class (note the capitalized first letter).

**`qtcm`:** This refers to the Python `qtcm` package.

**QTCM1 vs. QTCM:** Although the QTCM1 is currently the only version of a quasi-equilibrium tropical circulation model (QTCM), in principle one can construct a QTCM with any number of baroclinic modes. In anticipation of this, the names of the Python package and class do not end in a numeral. In this manual and the `qtcm` docstrings, QTCM and QTCM1 are used interchangably. Usually either of these phrases mean the quasi-equilibrium tropical circulation model in a generic sense, regardless of its form of implementation.

## 1.4 Current Version Information and Acknowledgments

This manual describes version 0.1.2 (dated September 12, 2008), of package `qtcm`. Johnny Lin is the primary author of the package.

The `qtcm` package is built upon the pure-Fortran QTCM1 model, version 2.3 (August 2002), with a few minor changes. Those changes are described in detail in Section 6.2.

The homepage for the `qtcm` package is http://www.johnny-lin.com/py_pkgs/qtcm. All Python code in this package, and the Fortran files *setbypy.F90* and *wrapcall.F90*, are © 2003–2008 by Johnny Lin[13] and constitutes a library that is covered under the GNU Lesser General Public License (LGPL):

> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License[14] as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

[13] http://www.johnny-lin.com

[14] http://www.gnu.org/copyleft/lesser.html

All other Fortran code in this package, as well as the makefiles, are covered by licenses (if any) chosen by their respective owners.

Transparent copies of this document are located online in PDF[16] and HTML[17] formats. The LaTeX source files are distributed with the qtcm distribution. While the HTML version is nearly identical to the PDF and LaTeX versions, not every feature in the manual was successfully converted. This is particularly true with figures, which are unnumbered in the HTML version and may be formatted differently than the authoritative PDF version. Thus, please consider the LaTeX version as the authoritative version.

**Acknowledgements:** Thanks to David Neelin and Ning Zeng and the Climate Systems Interactions Group at UCLA for their encouragement and help. On the Python side, thanks to Alexis Zubrow, Christian Dieterich, Rodrigo Caballero, Michael Tobis, and Ray Pierrehumbert for steering me straight. Early versions of some of this work was carried out at the University of Chicago Climate Systems Center, funded by the National Science Foundation (NSF) Information Technology Research Program under grant ATM-0121028. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

[15]http://www.gnu.org/licenses/fdl.html
[16]http://www.johnny-lin.com/py_pkgs/qtcm/doc/manual.pdf
[17]http://www.johnny-lin.com/py_pkgs/qtcm/doc/

## 1.5 Summary of Release History

- 2008 Sep 12: Version 0.1.2 released. Summary of changes:
  - Create `Qtcm` method `get_qtcm1_item`. This method is effectively an alias of method `get_qtcm_item`.
  - Create `Qtcm` method `set_qtcm1_item`. This method is effectively an alias of method `set_qtcm_item`.
  - Update User's Guide to phase out references to the `get_qtcm_item` and `set_qtcm_item` methods. Adding the "1" to the method names makes the purpose of the methods clearer.
  - Add unit tests to cover methods `get_qtcm1_item` and `set_qtcm1_item`.
- 2008 Jul 30: Updates to the User's Guide (just the online versions, not the copies released with the tarball).
- 2008 Jul 15: First publicly available distribution released (v0.1.1).

## 1.6 A Brief Description of The QTCM1

This description is copied from Ch. 3 of Lin [2], with minor revisions. Model formulation is fully described in Neelin & Zeng [3] and model results are described in Zeng et al. [5]. Neelin & Zeng [3] is based upon v2.0 of QTCM1 and Zeng et al. [5] is based on QTCM1 v2.1. The QTCM1 manual[18] [4] describes the details of model implementation.

QTCM1 differs from most full-scale GCMs primarily in how the vertical temperature, humidity, and velocity structure of the atmosphere is represented. First, instead of representing the vertical structure by finite-differenced levels, the model uses a Galerkin expansion in the vertical. The vertical basis functions are chosen according to analytical solutions under convective quasi-equilibrium conditions, so only a few need be retained. Temperature and humidity are each described by separate vertical basis functions ($a_1$ and $b_1$, respectively). Low-level variations in the humidity basis are larger than in the temperature basis. For velocity, QTCM1 uses a single baroclinic basis function ($V_1$) defined consistently with the temperature basis function, as well as a barotropic velocity mode ($V_0$). The vertical profiles of $a_1$, $b_1$, and $V_1$ are given in Figure 1.1. Currently, QTCM1 does not include a separate vertical degree of freedom describing the PBL. The horizontal grid spacing of the model is 5.625° longitude by 3.75° latitude.

These modes are chosen to accurately capture deep convective regions. Outside deep convective regions the mode is simply a highly truncated Galerkin representation. The system is much more tightly constrained than a full-scale GCM, yet

[18] http://www.atmos.ucla.edu/~csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

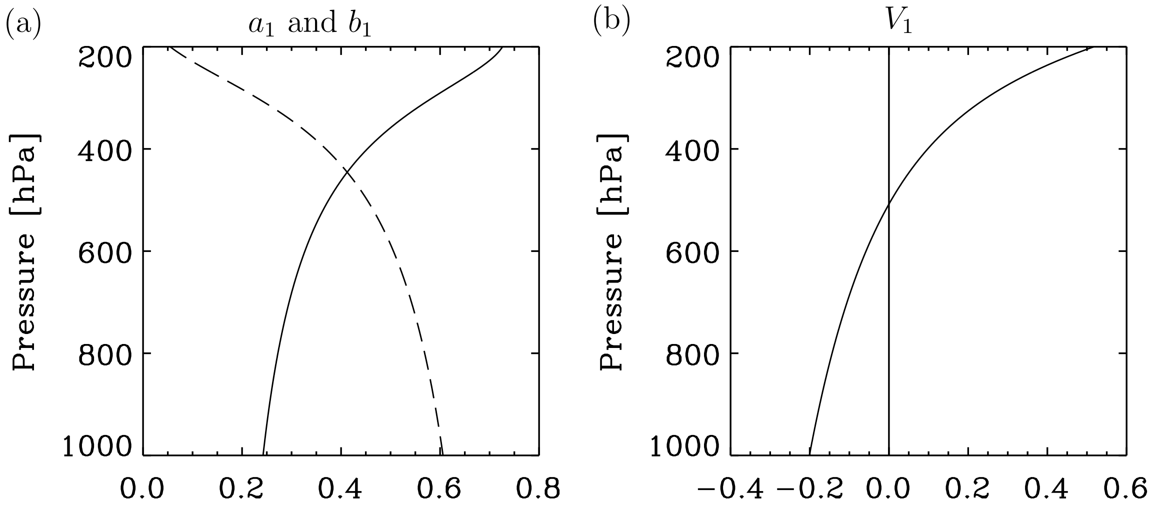

Figure 1.1: Vertical profiles of basis functions for (a) temperature $a_1$ (solid) and humidity $b_1$ (dashed) and (b) baroclinic component of horizontal velocity $V_1$.

hopefully retains the essential dynamics and nonlinear feedbacks. The result is that QTCM1 is easier to diagnose than a GCM, and is computationally fast (about 8 minutes per year on a Sun Ultra 2 workstation). Zeng et al. [5] show results indicating this intermediate-level model does a reasonable job simulating tropical climatology and ENSO variability.

Below is a summary of the main model equations [3]:

$$\partial_t \mathbf{v}_1 + \mathcal{D}_{V1}(\mathbf{v}_0, \mathbf{v}_1) + f\mathbf{k} \times \mathbf{v}_1 = -\kappa \nabla T_1 - \epsilon_1 \mathbf{v}_1 - \epsilon_{01} \mathbf{v}_0 \tag{1.1}$$

$$\partial_t \zeta_0 + \mathrm{curl}_z(\mathcal{D}_{V0}(\mathbf{v}_0, \mathbf{v}_1)) + \beta v_0 = -\mathrm{curl}_z(\epsilon_0 \mathbf{v}_0) - \mathrm{curl}_z(\epsilon_{10} \mathbf{v}_1) \tag{1.2}$$

$$\widehat{a_1}(\partial_t + \mathcal{D}_{T1})T_1 + M_{S1}\nabla \cdot \mathbf{v}_1 = \langle Q_c \rangle + (g/p_T)(-R_t^{\uparrow} - R_s^{\downarrow} + R_s^{\uparrow} + S_t - S_s + H) \tag{1.3}$$

$$\widehat{b_1}(\partial_t + \mathcal{D}_{q1})q_1 - M_{q1}\nabla \cdot \mathbf{v}_1 = \langle Q_q \rangle + (g/p_T)E \tag{1.4}$$

where (1.1) describes the baroclinic wind component, (1.2) describes the barotropic wind component, (1.3) is the temperature equation, and (1.4) is the moisture equation.

In the simplest formulation, the vertically integrated convective heating and moisture sink are assumed to be equal and opposite, so:

$$-\langle Q_q \rangle = \langle Q_c \rangle = \epsilon_c^*(q_1 - T_1) \tag{1.5}$$

For its convective parameterization for $Q_c$, this model uses the Betts-Miller [1] moist convective adjustment scheme, a scheme that is also used in some GCMs. In the convective parameterization, the coefficient $\epsilon_c^*$ is defined as:

$$\epsilon_c^* \equiv \widehat{a_1}\widehat{b_1}(\widehat{a_1} + \widehat{b_1})^{-1}\tau_c^{-1}\mathcal{H}(C_1) \tag{1.6}$$

where $\mathcal{H}(C_1)$ is zero for $C_1 \leq 0$, and one for $C_1 > 0$, and $C_1$ is a measure of the convective available potential energy (CAPE), projected onto the model's temperature and moisture basis functions.

Sensible heat ($H$) and evaporation ($E$) are given as bulk-aerodynamic formulations:

$$H = \rho_a C_D \mathrm{V}_s(T_s - (T_{rs} + a_{1s}T_1)) \tag{1.7}$$

$$E = \rho_a C_D \mathrm{V}_s(q_{sat}(T_s) - (q_{rs} + b_{1s}q_1)) \tag{1.8}$$

Longwave radiation components are denoted by $R$, and net shortwave radiation is denoted by $S$. The terms $\mathcal{D}_{V1}$ and $\mathcal{D}_{V0}$ are the advection-diffusion operators for the momentum equations (projected onto $V_0$ and $V_1(p)$, respectively). The terms $\mathcal{D}_{T1}$ and $\mathcal{D}_{q1}$ are the advection-diffusion operators for the temperature and moisture equations, respectively, using a vertical average projection. The $\langle X \rangle$ and $\widehat{X}$ operators are equivalent and denote vertically integration over the troposphere. Please see Neelin & Zeng [3] and Zeng et al. [5] for a more complete description of equations and coefficients.

# Chapter 2

# Installation and Configuration

## 2.1 Summary and Conventions

This section provides a summary of the steps needed to install `qtcm`, and a description of the naming conventions used in this chapter. If you have had a decent amount of experience with Python and installing software on a Unix system, this section will probably be all you need to read. The installation steps are:

1. Install a Fortran compiler (see Section 2.2 for a list of compilers known to work). This compiler should be in a directory listed in your system path (e.g., */usr/bin*, etc.).

2. Install all required packages (see Section 2.3 for details): Python, `matplotlib` (plus the `basemap` toolkit), NumPy (which includes `f2py`), Scientific Python, LaTeX, and netCDF.

   Python packages are required to be installed on your system in a directory listed in your `sys.path`, and the other packages/libraries are required to be in standard directories listed in your system path (e.g., */usr/bin*, */sw/include*, etc.).

   Make sure the executable for Python can be called at the Unix command line by typing both `python`. You might need to define a Unix alias that maps `python2.4` (or whichever version of Python you are using) to `python`.

3. Download[1] the `qtcm` tarball and extract the distribution into a temporary directory for building purposes. *qtcm-0.1.2* is the name of the `qtcm` distribution directory; the number following the hyphen is the version number of the distribution.

[1]http://www.johnny-lin.com/py_pkgs/qtcm/

In this manual, the path to *qtcm-0.1.2* will be called the "`qtcm` build path" and be given as */buildpath*. When you see */buildpath*, please substitute the actual temporary directory you created for building purposes.

4. The `qtcm` distribution directory *qtcm-0.1.2* contains the following principal subdirectories: *doc*, *lib*, *src*, *test*. Documentation is in *doc*, all the package modules are in *lib*, building of extension modules will take place in *src*, and testing of the package is done in *test*.

5. Compile `qtcm` extension modules in *src*: Go to *src*, copy the makefile from *src/Makefiles* corresponding to your system into *src*, rename to *makefile*, make changes to the makefile as needed, and execute:

   ```
   make clean
   make _qtcm_full_365.so
   make _qtcm_parts_365.so
   ```

   If you executed the make commands in *src,*, the extension modules will be automatically placed in *lib* in the *qtcm-0.1.2* directory. See Section 2.4 for details.

6. Copy the entire contents of *lib* in *qtcm-0.1.2* (not *lib* itself) to a directory named *qtcm* that is on your `sys.path`. For instance, for Mac OS X using Fink, many Python packages are located in a directory named */sw/lib/python2.4/site-packages*, or something similar, and this directory is on the system `sys.path`. If this is the case for your system, copy the contents of *lib* into */sw/lib/python2.4/-site-packages/qtcm*. (For Unix systems, the equivalent directory is usually */usr/-local/lib/python2.4/site-packages*.)

7. Test the `qtcm` distribution in *test*: This step is optional and can take a while. Testing requires you to first generate a suite of benchmarks using the pure-Fortran QTCM1 model, then running the tests of `qtcm` by typing:

   ```
   python test_all.py
   ```

   at the Unix command line while in *test*. See Section 2.5 for details.

At some point, I will automate the installation using Python's `distutils`[2] utilities.

[2]http://docs.python.org/dist/dist.html

## 2.2 Fortran Compiler

You must have a Fortran compiler installed on your system in order to compile `qtcm`. The compiler must be able to interface with a pre-processor, as QTCM1 makes copious use of pre-processor directives. `qtcm` is known to work with the following Fortran compilers on the following platforms:

| Compiler | Compiler Web Site | Platform(s) |
|---|---|---|
| `g95` | http://www.g95.org/ | Mac OS X |

It will probably work with other platforms, but I haven't been able to test platforms besides those listed above. Note that `g95` is not GNU Fortran (`gfortran`), the Fortran 95 compiler included with the more recent versions of GCC.

## 2.3 Required Packages

The following Python packages are required to be installed on your system in a directory listed in your `sys.path`:

- Python[3]: The Python programming language and interpreter. Make sure you have a version recent enough to be compatible with all the needed Python packages.
- `matplotlib`[4]: Scientific plotting package, using Matlab-like syntax. The `basemap` toolkit for `matplotlib` must also be installed.
- NumPy[5]: The standard array package for Python. The module name of NumPy imported in a Python session is `numpy`.
- Scientific Python[6]: Has netCDF file operators, in addition to other routines of use in scientific computing. The module name of Scientific Python imported in a Python session is `Scientific`.

One other required Python package, `f2py`, is now a part of the NumPy package, and so installation of NumPy is sufficient to give you both.

The package SciPy[7], which includes several Python-accessible scientific libraries, also includes NumPy (and thus `f2py`), so if you install SciPy, you don't have to install NumPy again. Note that SciPy is not the same as Scientific Python; the names are confusing.

A few non-Python packages are also required:

[3]http://www.python.org/
[4]http://matplotlib.sourceforge.net/
[5]http://numpy.scipy.org/
[6]http://dirac.cnrs-orleans.fr/plone/software/scientificpython/
[7]http://www.scipy.org

- LaTeX: A scientific typesetting program used by the `Qtcm` instance method `plotm` to handle exponents and subscripts. The most common Unix distribution of LaTeX is teTeX[8].

- netCDF: This set of libraries enables one to write datasets into a platform independent, binary format, with metdata attached. The netCDF 3.6.2 library[9] source code can be downloaded from UCAR[10].

For most Unix installations, the easiest way to install all the above is via a package manager, for instance `apt-get` in Debian GNU/Linux, `aptitude` or `synaptic` in Ubuntu GNU/Linux, and `fink` in Mac OS X. Of course, you can also download a package's source code and build direct and/or install using Python's `distutils`[11] utilities.

## 2.4 Compiling Extension Modules

The extension modules (*.so* files) are imported and used by `qtcm` objects, and contain the Fortran QTCM1 model that is called by the `qtcm` Python wrappers. These extension modules are located in the *lib* directory of the `qtcm` distribution, and, in general, need to be created only when the `qtcm` package is installed.

Two extension modules are created: *_qtcm_full_365.so* and *_qtcm_parts_365.so*. Both modules define QTCM1 models where:

- A year is 365 days long (makefile macro `YEAR360` is off).

- Model output is written to netCDF files (makefile macro `NETCDFOUT` is on).

- The atmospheric boundary layer model is used (makefile macro `NO_ABL` is off).

- A global domain is used (makefile macro `SPONGES` is off).

- Topography effects due to induced divergence are not included (makefile macro `TOPO` is off).

- Coupling between atmosphere and ocean is through mean fluxes (makefile macro `CPLMEAN` is off).

- The mixed layer ocean model is not used (makefile macros `MXL_OCEAN` and `BLEND_SST` are both off).

[8]http://www.tug.org/teTeX
[9]http://www.unidata.ucar.edu/software/netcdf/
[10]http://www.unidata.ucar.edu/downloads/netcdf/netcdf-3_6_2/
[11]http://docs.python.org/dist/dist.html

(All other makefile macros not listed are also turned off.) The only difference between these two extension modules is that the "full" module is used by `Qtcm` instances where `compiled_form` is set to `'full'`, and the "parts" module is used by `Qtcm` instances where `compiled_form` is set to `'parts'`. See Section 4.4 for details about the `compiled_form` attribute.

The extension modules are created through the following steps:

1. Go to the `qtcm` distribution directory *qtcm-0.1.2* located in your build path */buildpath*. Go to the *src* sub-directory. This is where all the building of the extension modules will take place.

2. Copy the makefile that corresponds to your platform to the *src* directory, and rename it *makefile*. The *Makefiles* sub-directory of *src* contains makefiles for various platforms.

3. In *makefile*, make the following changes:

   (a) Change the `FC` environment variable as needed, if your Fortran compiler is different.

   (b) Change the `FFLAGSM` environment variable, if the compiler flags listed are not supported by your compiler.

   (c) Change the `-I` and `-L` parts of the `NCINC` and `NCLIB` environment variables so that the paths for the netCDF library and include files match your system's installation:

   ```
   NCINC=-I/yourpath/netcdf/include
   NCLIB=-L/yourpath/netcdf/lib -lnetcdf
   ```

   Set *yourpath* to the full path to the *netcdf* directory where the *include* and *lib* sub-directories are that hold the netCDF libraries and include files. (You shouldn't have to change the `-l` part of `NCLIB`, since it is standard to name the netCDF library *libnetcdf.a*. But if you have a non-standard installation, change the `-l` part too.)

4. At the Unix prompt, type:

   ```
   make clean && make _qtcm_full_365.so && make _qtcm_parts_365.so
   ```

   to clean up leftover files from previous compilations, and to compile the extension module shared object files *_qtcm_full_365.so* and *_qtcm_parts_365.so*.

The makefile will automatically move the shared object files into *../lib*, overwriting any pre-existing files of the same name. A detailed description of the makefile and using `f2py` is given in Section 6.6, if you wish to create a different extension module.

## 2.5 Testing the Installation

The `qtcm` distribution comes with a set of tests for the package, using Python's `unittest` package. Just to warn you, the tests take around an hour to run. The tests will not work if the contents of *lib* after you've finished building `qtcm` have not been copied to a directory named *qtcm* that is on your `sys.path` path, so make sure you've gone through all the install steps (summarized in Section 2.1) before you do these tests.

**NB:** For these tests to work, both `python` and `python2.4` must refer to the executable for the Python installation on your system that you are using for running `qtcm`.

The tests require a set of benchmark output files in the *test/benchmarks* directory in the *qtcm-0.1.2* directory (the output will be in directories whose names begin with "aquaplanet" or "landon"). These output files are not included with the `qtcm` distribution, and must be created, by doing the following:

1. Goto the directory *test/benchmarks/create/src* in the *qtcm-0.1.2* `qtcm` distribution directory, and copy the makefile from sub-directory *Makesfiles*, that corresponds to your system to the *test/benchmarks/create/src* directory. Rename the makefile in *test/benchmarks/create/src* to *makefile*.

2. In *makefile*, make the following changes:

   (a) Change the `FC` environment variable as needed, if your Fortran compiler is different.

   (b) Change the `FFLAGSM` environment variable, if the compiler flags listed are not supported by your compiler.

   (c) Change the `-I` and `-L` parts of the `NCINC` and `NCLIB` environment variables so that the paths for the netCDF library and include files match your system's installation:

   ```
   NCINC=-I/yourpath/netcdf/include
   NCLIB=-L/yourpath/netcdf/lib -lnetcdf
   ```

   Set *yourpath* to the full path to the *netcdf* directory where the *include* and *lib* sub-directories are that hold the netCDF libraries and include files. (You shouldn't have to change the `-l` part of `NCLIB`, since it is standard to name the netCDF library *libnetcdf.a*. But if you have a non-standard installation, change the `-l` part too.)

3. Go to the directory *test/benchmarks/create* in the *qtcm-0.1.2* `qtcm` distribution directory.

4. Type `python create_benchmarks.py` at the Unix command line to run the benchmark creation script.

The created benchmarks will be located in *test/benchmarks*, in directories with names related to the run that was done, as described earlier. The benchmarks are created using the pure-Fortran QTCM1 model code, version 2.3 (August 2002), with an altered makefile (described above) and the following code change: In all *.F90* files, occurrences of:

```
Character(len=130)
```

are changed to:

```
Character(len=305)
```

This enables the model to properly deal with longer filenames. The number "305" is chosen to make search and replace easier.

Once the benchmarks are created, you can test the `qtcm` package by doing the following:

1. Go to the *test* directory in the *qtcm-0.1.2* directory.
2. Type `python test_all.py` at the Unix command line.

If at the end of the test runs you see this message (or something similar):

```
----------------------------------------------------------------------
Ran 93 tests in 1244.205s

OK
```

then everything worked fine! If you get any other message, the test(s) have failed.

## 2.6 Model Performance

The wall-clock time values below give the mean over three separate 365 day aqua-planet runs, using climatological sea surface temperature for lower boundary forcing. NetCDF output is written daily, for both instantaneous and mean values. The time step is 1200 sec, and the version of `qtcm` used is 0.1.1. The horizontal grid spacing of all model versions is 5.625° longitude by 3.75° latitude. Values are in seconds:

| **System** | **Pure** | **Full** | **Parts** |
|---|---|---|---|
| Mac OS X: MacBook 1.83 GHz Intel Core Duo running Mac OS X 10.4.10 with 1 GB RAM (Python 2.4.3, NumPy 1.0.3, `f2py` 2_3816). | 152.59 | 153.63 | 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 (Python 2.5.2, NumPy 1.1.0, `f2py` 2_5237). | 43.73 | 44.79 | 47.45 |

"Pure" refers to the pure-Fortran version of QTCM1. "Full" refers to a `qtcm` run session with `compiled_form` set to `'full'`. "Parts" refers to a `qtcm` run session with `compiled_form` set to `'parts'`. (Section 4.4 has details about the difference between compiled forms.)

The `'parts'` version of `qtcm` gives Python the maximum flexibility in accessing compiled QTCM1 model subroutines and variables. The price of that flexibility is an increase in run time of approximately 4–9% over the pure-Fortran version. The difference in performance between the `'full'` version of `qtcm` and the pure-Fortran version of QTCM1 is between negligible and 3% longer.

To make a timing for the pure-Fortran model, go to *test/benchmarks/timing/work* in */buildpath* and run the *timing_365.sh* script in that directory. That script runs the QTCM1 model using `/usr/bin/time`, which at the end of the script will output the amount of time it took to make the model run. Run the timing script three times and average the values to obtain a time comparable to the above.

To make a timing for the `qtcm` model, type `python timing_365.py` while in the *test* directory in */buildpath*. Three run sessions will be made for `compiled_form` equal to `'full'` and `'parts'`, the times are averaged, and the value are output at the end of the script.

## 2.7 Installing in Mac OS X

### 2.7.1 Introduction

This section describes issues and a summary of the installation steps I followed to install `qtcm` on a Mac running OS X. It is a specific realization of the general installation instructions found in Sections 2.1–2.5. I first worked through these installation steps during June–July 2007, with updates during July 2008. The best way to go through this section is to go through the summary of the installation steps in Section 2.7.6, and looking back to other sections as needed.

### 2.7.2 Platform and Unix Dependencies

This work was done on a MacBook 1.83 GHz Intel Core Duo running Mac OS X 10.4.11. My machine has 1 GB RAM and 64 GB of disk in its main partition.

I recommend you turn-off your antivirus software before you do the installs. Problems have been reported by Fink users[12] using the Fink package manager with antivirus software enabled.

There are a variety of dependencies that are required to get your Mac up-and-running as a scientific computing platform. The most basic is installing Apple's

[12]http://finkproject.org/faq/usage-fink.php?phpLang=en#kernel-panics

XCode[13] developer tools.[14] This set of tools contains compilers and libraries needed to do anything further. You have to be a member of Apple's Developer Connection, but registration is free.

Besides XCode, there are a variety of Unix libraries and utilities that you need. I first tried installing them by myself, from scratch, into */usr/local*, but it was hard to keep track of all the dependencies. A few that did work, and that I installed from their disk images, are: MacTeX[15], MAMP[16], and Tcl/Tk Aqua BI (Batteries Included)[17].[18]

For everything else, thankfully, there's the Fink Project[19] which uses a package manager built upon Debian tools to install ports of Unix programs onto a Mac. I just downloaded[20] a binary version of the Fink 0.8.1 installer for Intel Macs, installed Fink, and used its package management tools to install (almost) everything else I needed.[21]

Although you do not need anything besides a Fortran compiler and the netCDF libraries to run QTCM1 in its pure-Fortran form, in order to manipulate the model and use this Python version `qtcm`, you need to have Python installed. The default Python that comes with the Mac is a little old, so I used Fink to also install Python 2.5 and related packages, including matplotlib[23], ScientificPython[24], and SciPy[25] (see Section **??** for details).

### 2.7.3 Fortran Compiler

There are a variety of high-quality, commercial Fortran compilers. Unfortunately, because I do not have a research budget, I am not able to use those compilers. The GNU Compiler Collection[26] (GCC) provides a suite of open-source compilers, some of

[13]http://developer.apple.com/tools/xcode/

[14]The package should work in Mac OS X 10.4 with XCode 2.4.1 and higher; I've tried it with both 2.4.1 and XCode 2.5. Note that XCode 3.1 only works on Mac OS X 10.5.

[15]http://www.tug.org/mactex/

[16]http://www.mamp.info/

[17]http://tcltkaqua.sourceforge.net/

[18]Theoretically you can use Fink to install the equivalent of these packages, but I like the specific collection found in these packages. For instance, Tcl/Tk Aqua BI runs natively on the Mac.

[19]http://www.finkproject.org/

[20]http://www.finkproject.org/download/index.php?phpLang=en

[21]The one drawback of Fink is that it sometimes has stability problems. In those cases, Fink provides command line suggestions to fix the problems, which sometimes will work. If not, sometimes deleting Fink and everything it installed,[22] and starting afresh, will do the trick. It also appeared to me that sometimes when I installed multiple packages via one `fink install` call, the installation did not work as well as when I installed only one package per call.

[23]http://matplotlib.sourceforge.net/

[24]http://dirac.cnrs-orleans.fr/plone/software/scientificpython/

[25]http://www.scipy.org

[26]http://gcc.gnu.org/

which are the standards of their language. Most of the GCC compilers are installed on your Mac when you install XCode.

GNU Fortran[27] (`gfortran`), is the Fortran 95 compiler included with the more recent versions of GCC. Unfortunately, I was not able to get it to compile QTCM1. There is a second open-source Fortran compiler, G95[28] (`g95`), which some feel is farther along in its development than `gfortran`. I was able to successfully compile QTCM1 with `g95` on my Mac. I used Fink to install G95 (see Section **??** for details).

### 2.7.4 NetCDF Libraries

For some reason, the netCDF libraries and include files installed by Fink didn't correspond to the files needed by the calling routines in `qtcm`. To solve this, I compiled my own set of netCDF 3.6.2 libraries[29] using the tarball downloaded from UCAR[30].

Once I uncompressed and untarred the package, and went into the top-level directory of the package, I built the package by typing the following at the Unix prompt:

```
./configure --prefix=/Users/jlin/extra/netcdf
make check
make install
```

This installed the netCDF binaries, libraries, and include files into sub-directories *bin*, *lib*, and *include* in the directory specified by `--prefix`. If you want to install the netCDF libraries in the default (usually */usr/local*), just leave out the `--prefix` option.

Note: When you build netCDF, make sure the build directory is not in the directory tree of `--prefix` (or the default directory */usr/local*).

### 2.7.5 Makefile Configuration

#### NetCDF

In the *src* directory in the `qtcm` distribution, there is a sub-directory *Makefiles* that contains the makefiles for a variety of platforms. Edit the file *makefile.osx_g95* so that the lines specifying the environment variables for the netCDF libraries and include files:

```
NCINC=-I/Users/jlin/extra/netcdf/include
NCLIB=-L/Users/jlin/extra/netcdf/lib -lnetcdf
```

[27] http://gcc.gnu.org/fortran/
[28] http://www.g95.org/
[29] http://www.unidata.ucar.edu/software/netcdf/
[30] http://www.unidata.ucar.edu/downloads/netcdf/netcdf-3_6_2/

are changed to the path where your *manually compiled* netCDF libraries and include files are.

Copy *makefile.osx_g95* from the *Makefiles* sub-directory in *src* into *src*. In other words, from the `qtcm` distribution directory (i.e., */buildpath*), at the Unix prompt execute:

```
cp src/Makefiles/makefile.osx_g95 src/makefile
```

**Linking Order**

Compilers in the GNU Compiler Collection (GCC) search libraries and object files in the order they are listed in the command-line, from left-to-right[31]. Thus, if routines in *b.o* call routines in *a.o*, you must list the files in the order *a.o b.o*.

For some reason, that isn't the case for `g95`. Thus, you will find `g95` makefile rules structured like the following (below is part of the rule to create an executable (*qtcm*) for benchmark runs):

```
qtcm:  main.o
       $(FC) -O $(NCINC) -o $@ main.o $(QTCMLIB) $(NCLIB)
```

even though *main.o* depends on the QTCM library (specified in macro setting `$(QTCMLIB)`), which in turn depends on the netCDF library (specified in macro setting `$(NCLIB)`).

### 2.7.6 Summary of Steps

The following summarizes all the steps I took to install `qtcm` in Mac OS X:

1. Install XCode 2.5[32].

2. Install MacTeX[33], MAMP[34], and TCL/Tk Aqua BI (Batteries Included)[35].

3. Install Fink 0.8.1[36]. Make sure you set up your environment to enable you to use the packages you install with Fink (e.g. `PATH` settings, etc.). Most of the time, that just means adding the line `source /sw/bin/init.csh` to your *.cshrc* file (or the equivalent in your *.bashrc*).

   Note that for many of the packages needed to run `qtcm`, you need to configure Fink to download packages from the unstable trees. To do that, add `unstable/main` and `unstable/crypto` to the `Trees:` line in */sw/etc/fink.conf*, and run:

[31] http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Link-Options.html#index-l-670
[32] http://developer.apple.com/tools/xcode/
[33] http://www.tug.org/mactex/
[34] http://www.mamp.info/
[35] http://tcltkaqua.sourceforge.net/
[36] http://www.finkproject.org/download/index.php?phpLang=en

```
fink selfupdate
fink index
fink scanpackages
fink update-all
```

When `selfupdate` runs, choose `rsync` for the self update method. If you do not, Fink will not look in the unstable trees for packages.

4. Use Fink to install the `g95` Fortran compiler. From a Unix prompt, type:

   ```
   fink --use-binary-dist install g95
   ```

5. Use Fink to install Python and the NumPy package (which `f2py` is a part of). From a Unix prompt, type:

   ```
   fink --use-binary-dist install python25
   fink --use-binary-dist install scipy-core-py25
   ```

   (Numpy used to be called SciPy Core.) If you want to install Python 2.4 instead, just change the "25" and "py25" above (and in later occurrences) to "24" and "py24", respectively. Note that Fink does not have a version of epydoc for Python 2.4, so if you wish to create documentation using epydoc, you will need to install Python 2.5.

6. Install teTeX and LaTeX2HTML using Fink. From a Unix prompt, type:

   ```
   fink --use-binary-dist install tetex
   fink --use-binary-dist install latex2html
   ```

   When prompted, choose ghostscript and ghostscript-fonts to satistfy the dependency (which should be the default options). I tried choosing system-ghostscript8, but Fink looks for ghostscript 8.51 and didn't recognize ghostscript 8.57 that was already installed in */usr/local* (via my MacTeX install). LaTeX2HTML has a package required by the `qtcm` manual LaTeX file.

7. Install additional programming and scientific packages and libraries using Fink. From a Unix prompt, type:

   ```
   fink --use-binary-dist install scientificpython-py25
   fink --use-binary-dist install matplotlib-py25
   fink --use-binary-dist install matplotlib-basemap-py25
   fink --use-binary-dist install matplotlib-basemap-data-py25
   fink --use-binary-dist install xaw3d
   fink --use-binary-dist install fftw fftw3
   fink --use-binary-dist install epydoc-py25
   fink --use-binary-dist install graphviz
   fink --use-binary-dist install scipy-py25
   ```

8. Manually install netCDF 3.6.2 (see Section 2.7.4).
9. From this point on, you can follow the general instructions given in Section 2.1, starting with step 3. Please do not ignore, however, Section 2.7's Mac-specific details.

## 2.8 Installing in Ubuntu

### 2.8.1 Introduction

This section describes installation issues I followed to install `qtcm` on my Dell PowerEdge 860 running Ubuntu GNU/Linux 8.04.1 LTS (Hardy). The machine has 2.66 GHz Quad Core Intel Xeon processors (64 bit), 4 GB RAM, and 677 GB of disk in its main partition. This section is a specific realization of the general installation instructions found in Sections 2.1–2.5. I worked through these installation steps during July 2008. The best way to go through this section is to go through the summary of the installation steps in Section 2.8.5, and looking back to other sections as needed.

### 2.8.2 Fortran Compiler

The easiest Fortran compiler to install in Ubuntu 8.04.1 is GNU Fortran[37] (`gfortran`), the Fortran 95 compiler included with the more recent versions of the GNU Compiler Collection (GCC); you can use any package manager (e.g., `apt-get`, `aptitude`) to install it. Unfortunately, I was not able to get it to compile QTCM1. I was, however, able to successfully compile QTCM1 using the second open-source Fortran compiler, G95[38] (`g95`), which some feel is farther along in its development than `gfortran`. G95, however, is not supported as an Ubuntu package, and so I had to manually install it.

I downloaded the binary version of G95 v0.91 (the Linux x86_64/EMT64 with 32 bit default integers) using the following `curl` command:[39]

```
curl -o g95.tgz http://ftp.g95.org/v0.91/g95-x86_64-32-linux.tgz
```

which saves the *.tgz* file as the local file *g95.tgz*. After that, I followed the G95 project's standard installation instructions[40] to finish the install.[41] The regular Linux

[37]http://gcc.gnu.org/fortran/

[38]http://www.g95.org/

[39]I use `curl` because I usually access my Ubuntu server via a terminal session.

[40]http://g95.sourceforge.net/docs.html#starting

[41]The G95 installation instructions say you can put *g95-install* anywhere, and make a link to the executable `g95` in *~/bin*. I put *g95-install* in */usr/local*, and while in */usr/local/bin*, I put a link to the G95 executable using the command:

```
sudo ln -s ../g95-install_64/bin/x86_64-suse-linux-gnu-g95 g95.
```

x86 version of G95 (in *g95-x86-linux.tgz* from the G95 website) did not work on my machine.

### 2.8.3 NetCDF Libraries

For some reason, the netCDF libraries and include files installed from the Ubuntu packages do not correspond to the files needed by the calling routines in `qtcm`. To solve this, I compiled my own set of netCDF 3.6.2 libraries[42] using the tarball downloaded from UCAR[43].

Once I uncompressed and untarred the package, and went into the top-level directory of the package, I built the package by typing the following at the Unix prompt:

```
export FC=g95
export FFLAGS="-O -fPIC"
export FFLAGS="-fPIC"
export F90FLAGS="-fPIC"
export CFLAGS="-fPIC"
export CXXFLAGS="-fPIC"
./configure
make check
sudo make install
```

(The `export` commands set environment variables for the Fortran compiler and Fortran and other compiler flags. The `-fPIC` flag enables the compilers to create position independent code, needed for shared libraries in Ubuntu on a 64 bit Intel processor.)

The above installs the netCDF binaries, libraries, and include files into subdirectories *bin*, *lib*, and *include* in */usr/local*, the default. The include files for this netCDF installation are thus located in */usr/local/include*, and the libraries for this netCDF installation are location in */usr/local/lib*. (If you want to specify a different installation location, use the `--prefix` option in `configure`.) While you don't have to have root privileges during the configuration and check steps, you do during the installation step if you're installing into */usr/local* (thus the `sudo` in the last step).[44]

### 2.8.4 Makefile Configuration

#### NetCDF

In the *src* directory in the `qtcm` distribution, there is a sub-directory *Makefiles* that contains the makefiles for a variety of platforms. Edit the file *makefile.ubuntu_64_g95*

[42]http://www.unidata.ucar.edu/software/netcdf/

[43]http://www.unidata.ucar.edu/downloads/netcdf/netcdf-3_6_2/

[44]Note that when you build netCDF, make sure the build directory is not in the directory tree of `--prefix` or the default directory */usr/local*.

so that the lines specifying the environment variables for the netCDF libraries and include files:

```
NCINC=-I/usr/local/include
NCLIB=-L/usr/local/lib -lnetcdf
```

are changed to the path where your manually compiled netCDF libraries and include files are.

Copy *makefile.ubuntu_64_g95* from the *Makefiles* sub-directory in *src* into *src*. In other words, from the `qtcm` distribution directory (i.e., */buildpath*), at the Unix prompt execute:

```
cp src/Makefiles/makefile.ubuntu_64_g95 src/makefile
```

### Linking Order

Compilers in the GNU Compiler Collection (GCC) search libraries and object files in the order they are listed in the command-line, from left-to-right[45]. Thus, if routines in *b.o* call routines in *a.o*, you must list the files in the order *a.o b.o*.

For some reason, that isn't the case for `g95`. Thus, you will find `g95` makefile rules structured like the following (below is part of the rule to create an executable (*qtcm*) for benchmark runs):

```
qtcm:  main.o
       $(FC) -O $(NCINC) -o $@ main.o $(QTCMLIB) $(NCLIB)
```

even though *main.o* depends on the QTCM library (specified in macro setting `QTCMLIB`), which in turn depends on the netCDF library (specified in macro setting `NCLIB`).

### Shared Object PIC

In order to compile the model in Ubuntu on a 64 bit Intel processor, the model and the netCDF library it is linked to needs to be compiled to be position independent code (PIC).[46] This is accomplished with the `-fPIC` flag[47].

In the `qtcm` makefiles, the `-fPIC` flag is introduced in the macro `FFLAGSM`, for instance:

```
FFLAGSM = -O -fPIC
```

For makefiles used in creating extension modules, `-fPIC` must be passed into the `f2py` call. To do so, put the flags:

[45] http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Link-Options.html#index-l-670
[46] http://www.gentoo.org/proj/en/base/amd64/howtos/index.xml?part=1&chap=3
[47] http://www.fortran-2000.com/ArnaudRecipes/sharedlib.html

```
--f90flags="-fPIC" --f77flags="-fPIC"
```

after the `--fcompiler` flag in the `f2py` calling line.

The `-fPIC` flag must also be used when compiling the netCDF libraries, as described in Section 2.8.3. Failure to create PIC libraries in 64 bit Ubuntu can result in errors like the following when creating the `qtcm` extension modules:

```
ld: /usr/local/lib/libnetcdf.a(fort-attio.o): relocation
R_X86_64_32 against 'a local symbol' can not be used
when making a shared object; recompile with -fPIC
/usr/local/lib/libnetcdf.a: could not read symbols: Bad value
```

### 2.8.5 Summary of Steps

The following summarizes all the steps I took to install `qtcm` in Ubuntu 8.04.1 LTS (Hardy) running on a Quad Core Intel Xeon (64 bit) machine. Note that while I use the `aptitude` package manager, you are free to use any manager of your choice (e.g., `apt-get`, `synaptic`, etc.):

1. Install the G95 Fortran compiler from the binary distribution. See Section 2.8.2 for details.

2. Use an Ubuntu package manager to install the following packages, by typing:

   ```
   sudo aptitude update
   sudo aptitude install curl
   sudo aptitude install python-epydoc
   sudo aptitude install python-matplotlib
   sudo aptitude install python-netcdf
   sudo aptitude install python-scientific
   sudo aptitude install python-scipy
   sudo aptitude install texlive
   ```

   Installing `python-scipy` will also install NumPy and `f2py`, so you don't have to install the `python-numpy` package separately.

   Early-on as I debugged my `qtcm` install on Ubuntu, I encountered errors that I thought came from an old version of NumPy[48], and thus I replaced Ubuntu's packaged NumPy with NumPy 1.1.0 built directly from source.[49] (Note, you shouldn't install your new NumPy in the default location, which may cause problems later-on with Ubuntu's package manager.) Later on, I concluded the errors I had encountered were not because of the NumPy version, but by then I didn't want to try to reinstall NumPy again. So strictly speaking, the version

[48]http://cens.ioc.ee/pipermail/f2py-users/2008-June/001617.html
[49]http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103

of Numpy I used is not the one bundled with `python-scipy`, but that shouldn't be a problem.

3. Manually install netCDF 3.6.2 from source (see Section 2.8.3).

4. Manually install the `basemap` package of `matplotlib`. The source for the `basemap` toolkit is available from Sourceforge[50] I obtained version 0.9.9.1 using the following `curl` command:

```
curl -o basemap.tar.gz \
http://voxel.dl.sourceforge.net/sourceforge/matplotlib/basemap-0.9.9.1.tar.gz
```

   The *README* file in the *basemap-0.9.9.1* directory has detailed installation instructions. Note that you have to install the GEOS library first (*README* has detailed directions on how to do that too). To be on the safe-side, I would set the `FC` environment variable to the G95 compiler (e.g., with `export FC=g95` in Bash).

5. From this point on, you can follow the general instructions given in Section 2.1, starting with step 3. Please do not ignore, however, Section 2.8's Ubuntu-specific details.

[50] http://sourceforge.net/project/showfiles.php?group_id=80706

# Chapter 3

# Getting Started With qtcm

## 3.1 Your First Model Run

Figure 3.1 shows an example of a script to make a 30 day seasonal, aquaplanet model run, with run name "test", starting from November 1, Year 1.

```
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()
```

Figure 3.1: An example of a simple `qtcm` run.

The class describing the QTCM1 model is `Qtcm`. An instance of `Qtcm`, in this example `model`, is created the same way you create an instance of any class. When instantiating an instance of `Qtcm`, keyword parameters can be used to override any default settings. In the example above, the dictionary `inputs` specifying all keyword parameters is passed in on the instantiation of `model`.

The keyword parameter settings in Figure 3.1 have the following meanings:

- `runname`: This string ("test") is used in the output filename. QTCM1 writes mean and instantaneous output files to the directory given in `model.outdir.value`, with filenames *qm_runname.nc* for mean output and *qi_runname.nc* for instantaneous output.

- landon: When set to "0", the land is turned off and the run is an aquaplanet run. When set to "1", the land model is turned on.
- year0: The year the run starts on.
- month0: The month the run starts on (11 = November).
- day0: The day of the month the run starts on.
- lastday: The model runs from day 1 to lastday.
- mrestart: When set to "0", the run starts from default initial conditions (see Section 4.7.2 for a table of those values). When set to "1", the run starts from a restart file.
- compiled_form: This keyword sets what form the compiled QTCM1 model has, and its value is saved to the instance's compiled_form attribute. It is a string and can be set either to "parts" or "full". Most of the time, you will want to set it to 'parts'. This keyword is the only one that must be specified on instantiation; the model instance will at least instantiate using only the default settings for all the other keyword parameters (given in Appendix A). See Section 4.4 for details about what the compiled_form attribute controls.

By default, the SSTmode attribute, which controls whether the model will use climatological sea-surface temperatures (SST) or real SSTs, is set to the value "seasonal", thus giving a run with seasonal forcing at the lower-boundary over the ocean.

This example assumes that the boundary condition files, sea surface temperature files, and the model output directories are as specified in submodule defaults. Those values are described in Section A.1.

## 3.2 Managing Directories

Most of the time, your boundary condition files and output files will not be in the locations specified in Section A.1, or in the directory your Python script resides. The easiest way to tell your Qtcm instance where your input/output files are is to pass them in as keyword parameters on instantiation.

Figure 3.2 shows an example run where those directories are explicitly specified; in all other aspects, the run is identical to the one in Figure 3.1. In Figure 3.2, output from the model is directed to the directory described by string variable dirbasepath. dirbasepath is created by joining the current working directory

```
from qtcm import Qtcm
rundirname = 'test'
dirbasepath = os.path.join(os.getcwd(), rundirname)
inputs = {}
inputs['bnddir'] = os.path.join( os.getcwd(), 'bnddir',
                                 'r64x42' )
inputs['SSTdir'] = os.path.join( os.getcwd(), 'bnddir',
                                 'r64x42', 'SST_Reynolds' )
inputs['outdir'] = dirbasepath
inputs['runname'] = rundirname
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()
```

Figure 3.2: An example `qtcm` run showing detailed description of input and output directories.

with the run name given in string variable `rundirname`.[1] Setting keyword parameter `outdir` to `dirbasepath` sends output to `dirbasepath`. Keywords `bnddir` and `SSTdir` specify the directories where non-SST and SST boundary condition files, respectively, are found.

Interestingly, the default version of QTCM1 does *not* send all output from the model to `outdir`. The restart file *qtcm_yyyymmdd.restart* (where *yyyymmdd* is the year, month, and day of the model date when the restart file was written) is written into the current working directory, not the output directory. Thus, if you do multiple runs, you'll have to manually deal with the restart files that will proliferate.

Neither the QTCM1 model nor the `Qtcm` object create the directories specified in `bnddir`, `SSTdir`, and `outdir`. Failure to do so will create an error. I use Python's file management tools to make sure the output directory is created, and any old output files are deleted. Here's an example that does that, using the `dirbasepath` and `rundirname` variables from Figure 3.2:

```
if not os.path.exists(dirbasepath):  os.makedirs(dirbasepath)
qi_file = os.path.join( dirbasepath, 'qi_'+rundirname+'.nc' )
qm_file = os.path.join( dirbasepath, 'qm_'+rundirname+'.nc' )
if os.path.exists(qi_file):  os.remove(qi_file)
if os.path.exists(qm_file):  os.remove(qm_file)
```

## 3.3 Model Field Variables

The term "field" variable refers to QTCM1 model variables that are accessible at both the compiled Fortran QTCM1 model-level as well as the Python `Qtcm` instance-level. Field variables are all instances of the `Field` class, and are stored as attributes of the `Qtcm` instance.[2]

`Field` class instances have the following attributes:

- `id`: A string naming the field (e.g., "Qc", "mrestart"). This string should contain no whitespace.
- `value`: The value of the field. Can be of any type, though typically is either a string or numeric scalar or a numeric array.
- `units`: A string giving the units of the field.
- `long_name`: A string giving a description of the field.

[1]The Python `os` module enables platform-independent handling of files and directories. The `os.path.join` function resolves paths without the programmer needing to know all the possible directory separation characters; the function chooses the correct separation character at runtime. The `os.getcwd` function returns the current working directory.

[2]Note non-field variables can also be instances of `Field`, and that `Qtcm` instances have other attributes that are not equal to `Field` instances.

`Field` instances also have methods to return the rank and typecode of `value`.

Remember, if you want to access the value of a `Field` object, make sure you access that object's `value` attribute. Thus, for example, to assign a variable `foo` to the `lastday` value for a given `Qtcm` instance `model`, type the following:

```
foo = model.lastday.value
```

For scalars, this assignment sets `foo` by value (i.e., a copy of the value of attribute `model.lastday` is set to `foo`). In general, however, Python assigns variables by reference. Use the `copy` module if you truly want a copy of a field variable's value (such as an array), rather than an alias. For more details about field variables, see Section 4.7.

## 3.4 Run Sessions

### 3.4.1 What is a Run Session?

A run session is a unit of simulation where the model is run from day 1 of simulation to the day specified by the `lastday` attribute of a `Qtcm` instance. A run session is a "complete" model run, at the beginning of which all compiled QTCM1 model 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 in the Fortran model, and a Python-accessible snapshot is taken of the model variables that were written to the restart file.

### 3.4.2 Changing Variables

Between run sessions, changing any field variable is as easy as a Python assignment. For instance, to change the atmosphere mixed layer depth to 100 m, just type:

```
model.ziml.value = 100.0
```

When changing arrays, be careful to try to match the shape of the array.[3] You can use the NumPy `shape` function on a NumPy array to check its shape.

### 3.4.3 Continuing a Model Run

Figure 3.3 shows an example of two run sessions, where the second run session is a continuation of the first.

[3]At the very least, match the rank of the array, which is required for the routines in `setbypy` to properly choose which Fortran subroutine to use in reading the Python value. I haven't tested if only the rank is needed, however, for the passing to work, for a continuation run (my hunch is it won't).

```
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)
```

Figure 3.3: An example of two `qtcm` run sessions where the second run session is a continuation of the first. Assume `inputs` is a dictionary, and that earlier in the script the run name and all input and output directory names were added to the dictionary.

The first run session runs from day 1 to day 10. The second run session runs the model for another 30 days. Setting the `init_with_instance_state` of `model` to `True` tells the model to use the the values of the instance attributes (for prognostic variables, right-hand sides, and start date) are currently stored `model` as the initial values for the run_session.[4] The `cont` keyword in the second `run_session` call specifies a continuation run, and the value gives the number of additional days to run the model.

The set of runs described above would produce the exact same results as if you had gone into the Fortran model after 10 days, doubled the first baroclinic mode zonal velocity, and continued the run for another 30 days. With the Python example above, however, you didn't need to know you were going to do that ahead of starting the model run (which is what a compiled model requires you to do). Section 4.5 describes continuation runs in detail.

### 3.4.4 Passing Restart Snapshots Between Run Sessions

The pure-Fortran QTCM1 uses a restart file to enable continuation runs. A `Qtcm` instance can also make use of that option, through setting the `mrestart` attribute value (see Section 4.5 and Neelin et al. [4] for details). It's easier, however, instead of using a restart file, to pass along a "snapshot" dictionary.

The `Qtcm` instance method `make_snapshot` copies the variables that would be written out to a restart file into a dictionary that is saves as the instance attribute

[4]Unless overridden, by default, `init_with_instance_state` is set to True on `Qtcm` instance instantiation.

snapshot. This snapshot can be saved separately, for later recall. Note that snapshots are automatically made at the end of a run session.

The following example shows a model run_session call, following which the snapshot is saved to the variable snapshot:[5]

```
model.run_session()
mysnapshot = model.snapshot
```

After taking the snapshot, you might continue the run a while, and then decide to return to the snapshot you saved. To do so, use the sync_set_py_values_to_snapshot method to reset the model instance values to mysnapshot before your next run session:

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

See Section 4.5.5 for details regarding the use of snapshots, as well as for a list of what variables are saved in a snapshot.

## 3.5 Creating Multiple Models

### 3.5.1 Model Instances

Creating a new QTCM1 model is as simple as creating another Qtcm instance. For instance, to instantiate two QTCM1 models, model1 and model2, type the following:

```
from qtcm import Qtcm
model1 = Qtcm(compiled_form='parts')
model2 = Qtcm(compiled_form='parts')
```

model1 and model2 do *not* share any variables in common, including the extension modules holding the Fortran code. In creating the instances, a copy of the extension modules are saved in temporary directories.

### 3.5.2 Passing Snapshots To Other Models

The snapshots described in Section 3.4.4 can also be passed around to other model instances, enabling you to easily branch a model run:

[5]Remember Python assignment defaults to assignment by reference, so in this example the variable mysnapshot is a pointer to the model.snapshot attribute. (However, note that model.snapshot itself is not a reference, but a distinct copy of those variables; to do otherwise would result in a non-static snapshot.) If the model.snapshot attribute is dereferenced, then mysnapshot will become the sole pointer to the dictionary.

```
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()
```

The state of `model` after its run session is used to start `model1` and `model2`. This is an easy way to save time in spinning-up multiple models.

## 3.6 Run Lists

This feature of `Qtcm` objects is what really gives `Qtcm` model instances their flexibility. A run list is a list of strings and dictionaries that specify what routines to run in order to execute a particular part of the model. Each element of the run list specifies the method or subroutine to execute, and the order of the elements specifies their execution order.

For instance, the standard run list for initializing the the atmospheric portion of the model is named "qtcminit", and equals the following list:

```
['__qtcm.wrapcall.wparinit',
'__qtcm.wrapcall.wbndinit',
'varinit',
{'__qtcm.wrapcall.wtimemanager': [1]},
'atm_physics1']
```

This list is stored as an entry in the `runlists` dictionary (with key `'qtcminit'`). `runlists` is an attribute of a `Qtcm` instance. Table 4.3 lists all standard run lists.

When the run list element in the list is a string, the string gives the name of the routine to execute. The routine has no parameter list. The routine can be a compiled QTCM1 model subroutine for which an interface has been written (e.g., `__qtcm.wrapcall.wparinit`), a method of the of the Python model instance (e.g., `varinit`), or another run list (e.g., `atm_physics1`).

When the run list element is a 1-element dictionary, the key of the dictionary element is the name of the routine, and the value of the dictionary element is a list specifying input parameters to be passed to the routine on call. Thus, the element:

```
{'__qtcm.wrapcall.wtimemanager': [1]}
```

calls the `__qtcm.wrapcall.wtimemanager` routine, passing in one input parameter, which in this case is the value 1.

If you want to change the order of the run list, just change the order of the list. To add or remove routines to be executed, just add and remove their names from the

| `Qtcm` **Attribute Name** | **NetCDF Output Name** |
|---|---|
| `'Qc'` | `'Prec'` |
| `'FLWut'` | `'OLR'` |
| `'STYPE'` | `'stype'` |

Table 3.1: NetCDF output names for `Qtcm` field variables that are different from the `Qtcm` and compiled QTCM1 model variable names. The netCDF names are case-sensitive.

run list. Python provides a number of methods to manipulate lists (e.g., `append`). Since lists are dynamic data types in Python, you do not have to do any recompiling to implement the change.

The `compiled_form` attribute must be set to `'parts'` in the `Qtcm` instance in order to take advantage of the run lists feature of the class. Run lists are not available for `compiled_form='full'`, because subroutine calls are hardwired in the compiled QTCM1 model Fortran code in that case.

## 3.7 Model Output

### 3.7.1 NetCDF Output

Model output is written to netCDF files in the directory specified by the `Qtcm` instance attribute `outdir`. Mean values are written to an output file beginning with *qm_*, and instantaneous values are written to an output file beginning with *qi_*.

The frequency of mean output is controlled by `ntout`, and the frequency of instantaneous output is controlled by `ntouti`. `ntout.value` gives the number of days over which to average (and if equals `-30`, monthly means are calculated). `ntouti.value` gives the frequency in days that instantaneous values are output (monthly if it equals `-30`). (See Section 4.7.2 for a description of other output-control variables, and see the QTCM1 manual [4] for a detailed description of how these variables control output.)

Figure 3.4 gives an example of a block of code to read netCDF output, where `datafn` is the netCDF filename, and `id` is the string name of the field variable (e.g., `'u1'`, `'T1'`, etc.). (Note that the netCDF identifier for field variables is the same as the name in `Qtcm`, except for the variables given in Table 3.1.)

In the code in Figure 3.4, the array value is read into `data`, and the longitude values, latitude values, and time values are read into variables `lon`, `lat`, and `time`, respectively. As netCDF files also hold metadata, a description and the units of the variable given by `id`, and each dimension, are read into variables ending in `_name` and `_units`, respectively.

**NB:** All netCDF array output is dimensioned (time, latitude, longitude) when read into Python using the `Scientific` package. This differs from the way `Qtcm`

```
import numpy as N
import Scientific as S

fileobj = S.NetCDFFile(datafn, mode='r')

data = N.array(fileobj.variables[id].getValue())
data_name = fileobj.variables[id].long_name
data_units = fileobj.variables[id].units

lat = N.array(fileobj.variables['lat'].getValue())
lat_name = fileobj.variables['lat'].long_name
lat_units = fileobj.variables['lat'].units

lon = N.array(fileobj.variables['lon'].getValue())
lon_name = fileobj.variables['lon'].long_name
lon_units = fileobj.variables['lon'].units

time = N.array(fileobj.variables['time'].getValue())
time_name = fileobj.variables['time'].long_name
time_units = fileobj.variables['time'].units

fileobj.close()
```

Figure 3.4: Example of Python code to read netCDF output. See text for description.

saves field variables, which follows Fortran convention (longitude, latitude). Please be careful when relating the two types of arrays. Section 4.7.4 for a discussion of why there is this discrepancy.

### 3.7.2 Visualization

The `plotm` method of `Qtcm` instances creates line plots or contour plots, as appropriate, of model output of average fields of run session(s) associated with the instance. Some examples, assuming `model` is an instance of `Qtcm` and has already executed a run session:

- `model.plotm('Qc', lat=1.875)`: A time vs. longitude contour plot is made for the full range of time and longitude, at the latitude 1.875 deg N, for mean precipitation.

- `model.plotm('Qc', time=10)`: A latitude vs. longitude contour plot of precipitation is made for the full spatial domain at day 10 of the model run.

- `model.plotm('Evap', lat=1.875, lon=[100,200])`: A contour plot of time vs. longitude of evaporation is made for the longitude points between 100 and 200 degrees E, at the latitude 1.875 deg N.

- `model.plotm('cl1', lat=1.875, lon=[100,200], time=20)`: A deep cloud amount vs. longitude line plot is made for the longitude points between 100 and 200 degrees east, at the latitude 1.875 deg N, at day 20 of the model run.

In these examples, the number of days over which the mean is taken equals `model.ntout.value`. Also, the `plotm` method automatically takes into account the `Qtcm`/netCDF variable differences described in Table 3.1.

## 3.8 Documentation

Section 1.4 gives the online locations of the transparent copies of this manual. Model formulation is fully described in Neelin & Zeng [3] and model results are described in Zeng et al. [5] ([3] is based upon v2.0 of QTCM1 and [5] is based on QTCM1 v2.1). Additional documentation you'll find useful include:

- The `qtcm` Package API Documentation[6]

- The Pure-Fortran QTCM1 Manual[7] [4]

[6]http://www.johnny-lin.com/py_pkgs/qtcm/doc/html-api/
[7]http://www.atmos.ucla.edu/∼csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

# Chapter 4

# Using qtcm

## 4.1 Introduction

Now that you've successfully run your first model instances, in this chapter I provide detailed explanations regarding the features of qtcm. I present these explanations in a documentary rather than didactic fashion; my goal is to document how the features work. More details are given in the code docstrings. At the end of the chapter, in Section 4.10, I provide a few cookbook ideas/examples of ways to use the model.

## 4.2 Model Instances

An instance of a Qtcm model is created in qtcm the same way you create an instance of any class. For instance, to instantiate two Qtcm models, model1 and model2, I type the following:

```
from qtcm import Qtcm
model1 = Qtcm(compiled_form='full')
model2 = Qtcm(compiled_form='parts')
```

In the above example, model1 uses the compiled QTCM1 model that runs the model (essentially) using the Fortran driver, while model2 uses the compiled QTCM1 model where execution order and content all the way down to the atmospheric timestep level is controlled by Python run lists. (Section 4.4 has more details about the difference between compiled forms.)

For each instance of Qtcm, copies of all needed extension modules (e.g., *.so* files) are copied to a temporary directory that is automatically created by Python. The full path name of that directory is saved in the instance attribute sodir. These extension modules are then associated with the specific instance through private instance attributes, and thus every instance of Qtcm has its own separate variable and name space on both the Fortran and Python sides.[1] The temporary directory and all of its contents are deleted when the model instance is deleted.

On instantiation, `Qtcm` instances set all scalar field variables to their default values as given in the submodule `defaults` (and listed in Section A.1), and assign the fields as instance attributes. The instance attribute `init_with_instance_state` is set to True by default, unless overridden on instantiation.

## 4.3 Initializing a Model Run

In the pure-Fortran QTCM1, there are three broad classes of initialized variables:

1. Those that are read-in using a namelist,
2. Those that the are read-in from a restart file, and
3. Those that are set by assignment in the Fortran code.

These variables are a combination of scalars and arrays.

For `qtcm`, interfaces were built so that all classes of initialized variables that could be user-controlled are accessible and changeable at the Python-level. For `qtcm`, the set of variables that could be changed is also expanded, to include not just the first and second classes of pure-Fortran QTCM1 initialized variables. This was done to make `qtcm` more flexible. All variables that can be passed between the compiled QTCM1 model and Python model levels are called field variables, and are described in detail in Section sec:field.variables.

As it happens, all the namelist-set variables are scalars. In the pure-Fortran QTCM1, those variables are given default values prior to reading in of the namelist. To duplicate this functionality, on model instantiation, all scalar fields are set to their default values as given in the submodule `defaults` and listed in Section A.1. Most of the default values in `defaults` are the same as in the pure-Fortran QTCM1, but there are a few differences.[2] This setting of scalar defaults is the same for both `compiled_form = 'full'` and `compiled_form = 'parts'` instances. Of course, all `qtcm` fields are user-controllable, both via keyword input parameters at model instantiation as well as through direct manipulation of the instance attribute that stores the field variable.

The pure-Fortran QTCM1 initialized prognostic variables and right-hand sides are set in the Fortran subroutine `varinit`. Their they are read-in from a restart file or, as default, set by assignment. In `qtcm`, the same variables are initialized by a `Qtcm` instance method of the same name, `varinit`, for the case when `compiled_form = 'parts'`.

[1]The private instance attribute is `__qtcm`. See Section 6.7.3 for details about private `Qtcm` instance attributes.

[2]One difference being `mrestart`, which in `qtcm` will have the value of 0 in contrast to the pure-Fortran QTCM1 where the default is the 1.

For the case of `compiled_form = 'full'`, the compiled QTCM1 subroutine that is the same as in the pure-Fortran case is used, and that routine is inaccessible at the Python level. See Section 4.5.5's listing of snapshot variables, which also includes the prognostic variables and right-hand sides that are set in `varinit` (both Fortran and Python).

## 4.4 The `compiled_form` Keyword

The `qtcm` package is a Python wrap of the Fortran routines that make up QTCM1. The wrapping layer adds flexibility and functionality, but at the cost of speed. Thus, I created two types of extension modules from the Fortran QTCM1 code, one which permits very little control over the compiled Fortran *routines* at the Python level, and one that allows the Python-level to control model execution in the compiled QTCM1 model all the way down to the atmospheric timestep level.[3] The former extension module corresponds to `compiled_form = 'full'` and the latter extension module to `compiled_form = 'parts'`.

For `compiled_form = 'full'`, the compiled portion of the model encompasses (nearly) the entire QTCM1 model as a whole. Thus, the only compiled QTCM1 model modules or subroutines that Python should interact with is the `driver` routine (which executes the entire model) and the `setbypy` module (which enables communication between the compiled model and the Python-level of model fields.[4]

For `compiled_form = 'parts'`, the compiled portion of the model does not encompasses the model as a whole, but rather is broken up into separate units (as appropriate) all the way down to an atmosphere timestep. Thus, compiled QTCM1 model modules/subroutines that are accessible at the Python-level include those that are executed within an atmosphere timestep on up.

Because the difference in compiled forms fundamentally affects how the `Qtcm` instance facilitates Python-Fortran communication, this attribute must be set on instantiation via a keyword input parameter.

In the rest of this section, to avoid being verbose, when I write `'full'`, I mean the situation where `compiled_form = 'full'`. Likewise, when I write `'parts'`, I mean the situation where `compiled_form = 'parts'`.

### 4.4.1 Initialization for `compiled_form = 'full'`

For a model run of this case, the `Qtcm` instance will initialize the model using the Fortran `varinit` subroutine in the compiled QTCM1 model. This subroutine does the following:

[3]That control is via run lists, which are described in Section 4.6.

[4]The `setbypy` Python module is the wrap of the Fortran QTCM1 `SetByPy` module.

- If `mrestart = 1`, the restart file is used to initialize all prognostic variables. In terms of start date, the following rules are used:

  1. Variable `dateofmodel` is read from the restart file.
  2. If `day0`, `month0`, and `year0` are negative, or otherwise invalid (e.g., `month0` greater than 12), the invalid value is replaced with the day, month, and/or year of the day *after* that given by `dateofmodel`. If the value of `day0`, `month0`, or `year0` is not invalid in this sense, it is not replaced.

  Thus, if the restart file gives `dateofmodel` equal to 101102 (year 10, month 11, day 2), and `day0 = -1`, `month0 = -1`, `year0 = -1`, and `mrestart = 1`, the model will start running from year 10, month 11, day 3. If `dateofmodel` equals to 101102, and `day0 = -1`, `month0 = 3`, `year0 = -1`, the model will start running from year 10, month 3, day 3.

- If `mrestart = 0`, all prognostic variables and right-hand sides are set to an initial value (which for most of those variables is zero). In terms of start date, `day0` is set to 1 (and thus the value of `day0` previously input is ignored), and both `month0` and `year0` are set to 1 if their previously input values are invalid (where invalid means less than 1, or, for `month0`, greater than 12). Otherwise, `month0` and `year0` are left unchanged. Variable `dateofmodel` has the value it had when the variable was declared (which is determined by the compiler and usually is zero; `dateofmodel` will not be properly set until subroutine `TimeManager` is called.

  Thus, if `day0 = -1`, `month0 = -1`, `year0 = -1` is input into the model (say from a namelist) and `mrestart = 0`, the model will start running from year 1, month 1, day 1, and `dateofmodel` at the exit of subroutine `varinit` will equal its compiler-set default. If `day0 = 14`, `month0 = 3`, `year0 = 11`, and `mrestart = 0` on input into the model, the model will start running from year 11, month 3, day 1, and `dateofmodel` at the exit of subroutine `varinit` will equal its compiler-set default.

  Note that `dateofmodel` can thus be inconsistent with `month0` and `year0` at the exit of subroutine `varinit`.

This behavior with respect to initializing the start date is different than in QTCM1 versions 1.0 and 2.1. Please see the source code from those earlier QTCM1 versions for details.

### 4.4.2 Initialization for `compiled_form = 'parts'`

For `'parts'` model, the methodology of how initialized prognostic variables, right-hand sides, and start date related variables are set is controlled by the `Qtcm` instance attribute/flag `init_with_instance_state`. The initialization is (mostly) executed in the Python `varinit` method in the following way:

- If `init_with_instance_state` is False: The method as described for initialization for the `'full'` case is generally followed, with the exception that dateofmodel is set to match `day0`, `month0`, `year0`, prior to exit of `varinit`.

- If `init_with_instance_state` is True: the model object will initialize the model based on the current state of the model instance. This enables you to set a model run session's initial conditions based upon the state of the prognostic variables and parameters stored at the Python level, which is accessible at runtime.

Since the `init_with_instance_state = False` case is mainly described by the initialization method for the `'full'` case, I refer the reader to Section 4.4.1. For the case of `init_with_instance_state` is True, however, the task is more complicated. Specifically, for that case, initialization includes the following:

1. If not currently defined, variable `dateofmodel` is set to a default value of 0, which is specified in the module defaults.

2. The `mrestart` flag is ignored for variable initialization.

3. All prognostic variables and right-hand sides are set to an initial value (which for most of those variables is zero), unless the variable is defined at the Python level, in which case the inital value is set to the Python level defined value.

4. If `dateofmodel` is greater than 0, `day0`, `month0`, and `year0` are overwritten with values derived from `dateofmodel` in order to set the run to start the day *after* `dateofmodel`.

5. If `dateofmodel` is less than or equal to 0, `day0`, `month0`, and `year0` are set to their respective instance attribute values, if valid. For invalid instance attribute values, the invalid `day0`, `month0`, and/or `year0` is set to 1.

6. Variable `dateofmodel` is recalculated and overwritten to match `day0`, `month0`, `year0`, prior to exit of `varinit`.

As a result, for `init_with_instance_state` is True, the way you indicate to the model that a run session is a brand-new run is by setting, before the `run_session` method call, `dateofmodel` to a value less than or equal to 0, and `day0`, `model0`, and `year0` to the day you want the model to begin the run session. To indicate to the model you wish to continue a run, set `dateofmodel` to the day *before* you want the model to start running from.

Examples:

- If `day0 = -1`, `month0 = -1`, `year0 = -1`, and `dateofmodel = 0` is input into the model the model will start running from year 1, month 1, day 1, and variable `dateofmodel` at the exit of subroutine `varinit` will equal 10101.

- If `day0 = 14`, `month0 = 3`, `year0 = 11`, and `dateofmodel = 0` is input into the model, the model will start running from year 11, month 3, day 14, and variable `dateofmodel` at the exit of subroutine `varinit` will equal 110314.

- If `day0 = 14`, `month0 = 3`, `year0 = 11`, and `dateofmodel = 341023` is input into the model, the model will start running from year 34, month 10, day 24, and at the exit of subroutine `varinit`, `dateofmodel` will equal 341024, with `day0 = 24`, `month0 = 10`, and `year0 = 34`.

### 4.4.3 Communication Between Python and Fortran-Levels

After initialization, the second major difference between a `'full'` and `'parts'` model is how and when communication between the Python and Fortran levels can occur. For the `'full'` case, except for the passing in and out of variables before and after a run session, all variable passing and subroutine calling happens in the compiled QTCM1 model, with no control at the Python level. For the `'parts'` case, variables can be passed between the Python and Fortran-levels at all levels down to the atmospheric timestep, and many Fortran QTCM1 subroutines can be called from the Python-level.

#### Passing Variables

For all `compiled_form` cases, variables are passed back and forth between the Python `Qtcm` instance level and the compiled QTCM1 model Fortran-level using the `Qtcm` instance methods `get_qtcm1_item` and `set_qtcm1_item`:[5]

- `get_qtcm1_item`(*key*): Returns the value of the field variable given by the string *key*. If the compiled QTCM1 model variable given by *key* is unreadable, the custom exception `FieldNotReadableFromCompiledModel` is thrown. The value returned is a copy of the value on the Fortran side, not a reference to the variable in memory.

- `set_qtcm1_item`: Sets the value of a field variable in the compiled QTCM1 model *and at the Python-level,* automatically overriding any previous value at both levels. Thus, calling this method will change/create the `Qtcm` instance

[5]All Fortran routines used to pass variables back and forth are defined in the `setbypy` module of the *.so* extension module stored in the `Qtcm` instance variable `__qtcm`. All Fortran wrappers that enable Python to call compiled QTCM1 model subroutines are defined in the `wrapcall` module stored in the `Qtcm` instance variable `__qtcm`. These modules are described in detail in Sections 6.3.1 and 6.3.2, respectively.

attribute corresponding to the field variable. When the compiled QTCM1 model variable is set, a copy of the Python value is passed to the Fortran model; the variable is *not passed by reference.* This value comes from the `set_qtcm1_item` calling parameter list, *not* from the `Qtcm` instance attribute corresponding to the field variable.

The `set_qtcm1_item` method has two calling forms, one with one argument and the other with two arguments:

- One argument: The method is called as `set_qtcm1_item`(*arg*), where *arg* is either a string giving the name of the field variable or a `Field` instance.

- Two arguments: The method is called as `set_qtcm1_item`(*key*, *value*), where *key* is the string giving the name of the field variable and *value* is the value to set the model field variable to (note *value* can be a `Field` instance).

In either calling form, if no value given, the default value as defined in module `defaults` is used.

Some compiled QTCM1 model variables are not in a state where they can be set. An example is a compiled QTCM1 model pointer variable, prior to the pointer being associated with a target (an attempt to set would yield a bus error). In such cases, the `set_qtcm1_item` method will throw a `FieldNotReadableFromCompiledModel` exception, nothing will be set in the compiled QTCM1 model, and the Python counterpart field variable (if it previously existed) would be left unchanged.[6]

Examples, typed in at a Python prompt, and assuming that `model` is a `Qtcm` instance:

- `dtvalue = model.get_qtcm1_item('dt')`: Retrieves the value of field variable `dt` (timestep) from the compiled QTCM1 Fortran model and sets it to the Python variable `dtvalue`.

- `model.set_qtcm1_item('dt')`: Sets the value of field variable `dt` in the compiled QTCM1 Fortran model to the default value (as given in `defaults`), and sets the value of Python attribute `model.dt` also to that default value. Remember that `model.dt` is a `Field` instance.

- `model.set_qtcm1_item('dt', 2000.)`: Sets the value of field variable `dt` in the compiled QTCM1 Fortran model to 2000 (as a real), and sets the value of Python attribute `model.dt` also to 2000.

[6]We handle this situation in this way to enable the `Qtcm` instance to store variables even if the compiled model is not yet ready to accept them.

**Calling Compiled QTCM1 Model Subroutines**

All compiled QTCM1 model subroutines that can be called (except `driver` and `varptrinit`) are in the `setbypy` or `wrapcall` modules of the `Qtcm` instance private attribute `__qtcm`. (On `Qtcm` instance instantiation, `__qtcm` is set to the *.so* extension module that is the compiled QTCM1 Fortran model.) Thus, to call `wmconvct` in `wrapcall` at the Python-level, just type `model.__qtcm.wrapcall.wmconvct()` (where `model` is a `Qtcm` instance). For `driver` and `varptrinit`, these subroutines are not contained in a `__qtcm` module, and thus can be called directly (e.g., just type `model.__qtcm.driver()`). See Sections 6.3.1 and 6.3.2 for more information on the `setbypy` and `wrapcall` modules.

For the `'full'` case, the only compiled QTCM1 model subroutine you can usefully call during a run session is `driver`. For the `'parts'` case, while you can essentially call any subroutine given in a run list, you usually will not directly call a compiled QTCM1 model subroutine but will instead call it through including it in a run list. For example, if you have the following run list in a `'parts'` model:

```
[ 'qtcminit', '__qtcm.wrapcall.woutpinit' ]
```

Running this list using the `Qtcm` instance method `run_list` will result in `Qtcm` instance method `qtcminit` first being run, then the compiled QTCM1 Fortran model subroutine `woutpinit` in Fortran module `wrapcall` being run. See Section 4.6 and Table 4.3 for a discussion and list of the standard run lists that control routine execution content and order in the `'parts'` case.

## 4.5 Restart and Continuation Run Sessions

### 4.5.1 Restart Runs In the Pure-Fortran QTCM1

To enable restart of a model run, the pure-Fortran QTCM1 model writes out a restart file with the state of the prognostic variables and select right-hand sides at that point in the run (for a list of the variables, see Section 4.5.5). This binary file can then be read in by later model runs. The Fortran `mrestart` flag is passed in via a namelist; if `mrestart` is 1, the run uses the restart file (named *qtcm.restart*).

One of the problems with using the restart file to do a continuation run is that the continuation run will not be perfect. In other words, a 15 day run followed by a 25 day run based on the restart file generated at the end of the 15 day run will *not* give the exact same output as a continuous 40 day run.

### 4.5.2 Overview of Restart/Continuation Options In `qtcm`

For a `Qtcm` instance, in contrast to the pure-Fortran QTCM1, more than one method of continuation is available. Thus, for a continuation run, you need to tell the model "continue from what?" The `Qtcm` class provides three choices for restart/continuing a run:

1. From a restart file: Move/rename a QTCM1 restart file to the current working directory to *qtcm.restart*.
2. From a snapshot from another run session (see Sections 3.4.4 and 4.5.5).
3. From the values of the `Qtcm` instance you will be calling `run_session` from.

Restart/continuation methods 1 and 2 both suffer from the same problem as the pure-Fortran QTCM1 restart process: They do not produce perfect restarts (see Section sec:puref90.restart for details). In this section, I discuss the restart/continuation options for each `compiled_form` option.

Methods 1 and 2 are best used when making a run session from a newly instantiated `Qtcm` instance. Method 3 is best used when executing a run session using a `Qtcm` instance that has already gone through at least one run session. Regardless of which method you use, however, please note that anytime you execute a run session using a `Qtcm` instance that already has made a previous run session, some variables *cannot be updated* between run sessions. This feature is most noticeable with the output filename, and occurs because the name persists in the compiled QTCM model, and is stored in the extension module (*.so* files in `sodir`) associated with the instance. If you wish to control all variables possible from the Python level (including output filename), you need do the run session from a new model instance.

### 4.5.3 Restart/Continuation for `compiled_form='full'` Model Instances

The only option for restart when using `compiled_form='full'` model instances is method 1, to use a QTCM1 restart file.[7] To use this option, the value of the `mrestart` attribute must equal 1, the restart file must be named *qtcm.restart*, and the restart file must be in the current working directory. As with the pure-Fortran QTCM1 restart process, this method does not produce perfect restarts.

### 4.5.4 Restart/Continuation for `compiled_form='parts'` Model Instances

For the `compiled_form='parts'` case, all three restart/continuation methods described in Section 4.5.2 are available.

[7]The `cont` keyword parameter in `run_session` and the value of the `init_with_instance_state` attribute have no effect if `compiled_form='full'`. With `'full'`, the call to initialize variables all happens at the Fortran level (via the Fortran `varinit`, not the Python `varinit`), with no reference to the Python field states (or even existing Fortran field states, if present).

### Method 1: From a QTCM1 Restart File

To use the QTCM1 restart file mechanism, not only must the `mrestart` attribute have a value to 1, but the `init_with_instance_state` flag also has to be `False`, otherwise the `mrestart` attribute value will be ignored. As with the pure-Fortran QTCM1 restart process, this method does not produce perfect restarts.

### Method 2: From a `Qtcm` Instance Snapshot

You can take snapshots of the model state of a `Qtcm` instance by the `make_snapshot` instance method. This snapshot saves a copy of all the variables saved to a QTCM1 restart file (see Section 4.5.5 for the full list of fields), which then can be passed to other `Qtcm` instances for use in other run sessions.

The key difference between this method and method 3 (described below) is that `run_session` calls using the snapshot are done *without* the `cont` keyword input parameter (by default, `cont` is False). If the `cont` keyword is not False, it says the run session is a continuation run that uses the state of the compiled QTCM1 model for all variables that are not specified at, and read-in from, the Python level. If the `cont` keyword is False, the run session initializes as if it were a new run.

See Section 3.4.4 for details and an example of using snapshots to initialize a run session. Note that as with the pure-Fortran QTCM1 restart process, this method does not produce perfect restarts.

### Method 3: From the Calling `Qtcm` Instance

This method is used when you want to make a run session that is a "true" continuation run, i.e., one that uses the current state of the compiled QTCM1 model for all variables that are not read-in from the Python level (remember that `Qtcm` instances hold a subset of the variables defined at the Fortran level). The key reason to use this method for a continuation run session is that the continuation is byte-for-byte the same (if no fields are changed) as if the run just went straight on through. Thus, the continuation would be perfect: A 15 day run followed by a 25 day run using the same `Qtcm` instance with the `cont` keyword will give the exact same output as a continuous 40 day run. This is not the case when making a new instance and passing a restart file or a snapshot, because a separate extension module is used for those new instances.

Control of this method is accomplished through the `cont` keyword input parameter to the `run_session` method and the `init_with_instance_state` attribute of a `Qtcm` instance:

- `cont`: If set to False, the run session is not a continuation of the previous run, but a new run session. If set to True, the run session is a continuation of the previous run session. If set to an integer greater than zero, the run session is a continuation just like `cont = True`, but the value `cont` is set to is used for `lastday` and replaces `lastday.value` in the `Qtcm` instance.

- `init_with_instance_state`: If True, for a `run_session` call using the `cont` keyword, whatever the field values are in the Python instance are used in the run session. If False, model variables are set and initialized as described in Section 4.4.2. In that case, previous compiled QTCM1 model values will likely be overwritten. Thus, if you want a continuation run that uses the state of all field variables except for those you explicitly change at the Python-level, make sure `init_with_instance_state` is True.

(Note that the `cont` keyword has no effect if `compiled_form` is `'full'`. The default value of `cont` in a `run_session` call is False. The value of keyword `cont` is stored as private instance attribute `_cont`, in case you really need to access it elsewhere; see Section 6.7.3 for more details).

The example described in Section 3.4.3 is an example of method 3 in the list above: The second run session is continued from the state of `model`, with the values of `model`'s instance variables overriding any values in the compiled QTCM1 model in initializing the second run session.

This method has a few caveats worthy of note:

- The `init_with_instance_state` attribute value will have no effect unless the instance prognostic variables are set, i.e., unless a previous run session has been done. Another way to put it is for an initial run session right after a `Qtcm` instance is created, `varinit` will use the same initial values for prognostic variables (defined in `defaults` module variable `init_prognostic_dict`)[8] as it would with for both `init_with_instance_state` set to True or False).

- Continuation run sessions using this method have to continue with the next day from wherever the last run session left off, contiguously.[9] If you want to do a non-contiguous run, create a new `Qtcm` instance initialized with a snapshot instead of the continuation method describe in this section. will use restart rules to run a new model.

- When making a continuation run session using this method, you cannot change some variables, for instance, `outdir` and any of the date related variables. In fact, the only thing you should change for your continuation run session are the prognostic and diagnostic variables and `lastday`. This is because some variables cannot be updated between run sessions. As noted in Section 4.5.2, if you wish to control all variables possible from the Python level (including output filename), you need to execute the run session from a new model instance.

[8]`init_prognostic_dict` is the dictionary giving the default initial values of each prognostic variable and right-hand side (as defined by the restart file specification).

[9]For continuation run sessions, you keep the same extension module (the compiled *.so* library), and all the values that define the state where it left off.

### 4.5.5 Snapshots of a Qtcm Instance

The snapshot dictionary (briefly described in Section 3.4.4), saved as the Qtcm instance attribute snapshot, and generated by the method make_snapshot, saves the current state of the following instance field variables:

| Field | Shape | Units | Description |
|---|---|---|---|
| T1 | (64, 44) | K | |
| Ts | (64, 42) | K | Surface temperature |
| WD | (64, 42) | | |
| dateofmodel | | | Date of model coded as an integer as yyyymmdd |
| psi0 | (64, 43) | | |
| q1 | (64, 44) | K | |
| rhsu0bar | (3,) | | |
| rhsvort0 | (64, 42, 3) | | |
| title | | | A descriptive title |
| u0 | (64, 44) | m/s | Barotropic zonal wind |
| u0bar | | | |
| u1 | (64, 44) | m/s | Current time step baroclinic zonal wind |
| v0 | (64, 43) | m/s | Barotropic meridional wind |
| v1 | (64, 43) | m/s | |
| vort0 | (64, 42) | | |

These are the same variables saved to a QTCM1 restart file, and so a snapshot duplicates the restart functionality in the Python environment, but with more flexibility. Since the snapshot dictionary is a Python variable like any other, you can manipulate it and alter it to fit any condition you wish.

## 4.6 Creating and Using Run Lists

Section 3.6 provides an introduction to the role and use of run lists. A run list is a list of methods, Fortran subroutines, and other run lists that can be executed by the Qtcm instance run_list method. Run lists are stored in the Qtcm instance attribute runlists, which is a dictionary of run lists. The names of run lists should not be preceeded by two underscores (though elements of a run list may be very private variables), nor should names of run lists be the same as any instance attribute. Run lists are not available for compiled_form = 'full'.

The run_list method takes a single input parameter, a list, and runs through that list of elements that specify other run lists or instance method names to execute. Methods with private attribute names are automatically mangled as needed to become executable by the method. Note that if an item in the input run list is an instance method, it should be the entire name (not including the instance name) of the callable method, separated by periods as appropriate.

Elements in a run list are either strings or 1-element dictionaries. Consider the following example, where model is a Qtcm instance, and run_list is called using mylist as input:

```
model = Qtcm(...)
mylist = [ {'varinit':None},
           'init_model',
           '__qtcm.driver',
           {'set_qtcm1_item':  ['outdir', '/home/jlin']} ]
model.run_list(mylist)
```

The first element in mylist refers to a method that requires no positional input parameters be passed in (as shown by the None). The second and third elements in mylist also refers to methods that require no positional input parameters be passed in. The last element in mylist refers to a method with two input parameters. Note that while I use the term "method" to describe the elements, the strings/keys do not have to be only Python instance methods. The second element, for instance, refers to another run list, and the third element refers to a compiled QTCM1 model subroutine (note the __qtcm attribute).

When the run_list method is called, the items in the input run list are called in the order given in the list. For each element, the run_list method first checks if the string or dictionary key name corresponds to the key of an entry in the Qtcm instance attribute runlists. If so, run_list is called using that run list (i.e., it is a "recursive" call). If the string or dictionary key name does not refer to another run list, the run_list method checks if the string or dictionary key name is a method of the Qtcm instance, and if so the method is called. Any other value throws an exception.

If input parameters for a method are of class Field, the run_list method first tries to pass the parameters into the method as is, i.e., as Field object(s). If that fails, the run_list method passes its parameters in as the value attribute of the Field object.

If you want a variable that is being passed into a run list to be continuously updated, you have to set the parameter in the run list to a Field instance that is a Qtcm instance attribute, not just to the value of the field variable (or to a non-Field object). Otherwise, subsequent calls to that run list element will not use the updated values as input parameters.

For instance, if you had a run list element:

```
{'__qtcm.timemanager':[model.coupling_day,]}
```

and `model.coupling_day` were an integer (it's not by default, but pretend it was), then `run_list` calling `__qtcm.timemanager` will pass in a scalar integer rather than a binding to the variable `model.coupling_day`. In such a situation, if the variable `model.coupling_day` were updated in time, the `run_list` call of `__qtcm.timemanager` would not be updated in time. This happens because when the dictionary that is the run list element is created, the value of list element(s) attached to the dictionary element is set to the scalar value of `model.coupling_day` at that instant.

You can get around this feature by setting `Qtcm` instance attributes that will change with model execution to `Field` instances, and then referring to those attributes in the parameter list in the run list element. In that case:

```
{'__qtcm.timemanager':[model.coupling_day,]}
```

will use the current value of `model.coupling_day` anytime `__qtcm.timemanager` is called by `run_list`, if `model.coupling_day` is a `Field` object.

When `run_list`, encounters a calling input parameter that is a `Field` object, it will first try to pass the entire `Field` object to the method/routine being called. If that raises an exception, it will then try to pass just the value of the entire `Field` object. This is done to enable `run_list` to be used for both pure-Python and compiled QTCM Fortran model routines. Fortran cannot handle `Field` objects as input parameters, only values.

Table 4.3 shows all standard run lists stored in the `runlists` attribute upon instantiation of a `Qtcm` instance.

Of course, feel free to change the contents of any of the run lists after instantiation, or to add additional run lists to the `runlists` attribute dictionary. The ability to alter run lists at runtime gives the `qtcm` package much of its flexibility.

## 4.7 Field Variables and the `Field` Class

The term "field" variable refers to QTCM1 model variables that are accessible at both the compiled Fortran QTCM1 model-level as well as the Python `Qtcm` instance-level. Field variables are all instances of the `Field` class (though non-field variables can also be instances of `Field`).

Section 3.3 gives a brief introduction to the attributes and methods in a `Field` instance. A nitty gritty description of the class is found in its docstrings.

### 4.7.1 Creating Field Variables

To create a `Field` instance whose value is set to the default, instantiate with the field id as the only positional input argument. Thus:

| **Run List Name/Description** | **List Element(s) Name(s)** | **# Arg(s)** |
|---|---|---|
| atm_bartr_mode (calculate the atmospheric barotropic mode at the barotropic timestep) | __qtcm.wrapcall.wsavebartr | None |
| | __qtcm.wrapcall.wbartr | None |
| | __qtcm.wrapcall.wgradphis | None |
| atm_oc_step (calculate the atmosphere and ocean models at a coupling timestep) | _first_method_at_atm_oc_step | None |
| | __qtcm.wrapcall.wtimemanager | 1 |
| | __qtcm.wrapcall.wocean | 2 |
| | qtcm | None |
| | __qtcm.wrapcall.woutpall | None |
| atm_physics1 (calculate atmospheric physics at one instant) | __qtcm.wrapcall.wmconvct | None |
| | __qtcm.wrapcall.wcloud | None |
| | __qtcm.wrapcall.wradsw | None |
| | __qtcm.wrapcall.wradlw | None |
| | __qtcm.wrapcall.wsflux | None |
| atm_step (calculate the entire atmosphere at one atmosphere timestep) | atm_physics1 | None |
| | __qtcm.wrapcall.wsland1 | None |
| | __qtcm.wrapcall.wadvctuv | None |
| | __qtcm.wrapcall.wadvcttq | None |
| | __qtcm.wrapcall.wdffus | None |
| | __qtcm.wrapcall.wbarcl | None |
| | _bartropic_mode_at_atm_step | None |
| | __qtcm.wrapcall.wvarmean | None |
| init_model (initialize the entire model, i.e., the atmosphere and ocean components and output) | qtcminit | None |
| | __qtcm.wrapcall.woceaninit | None |
| | __qtcm.wrapcall.woutpinit | None |
| qtcminit (initialize the atmosphere portion of the entire model) | __qtcm.wrapcall.wparinit | None |
| | __qtcm.wrapcall.wbndinit | None |
| | varinit | None |
| | __qtcm.wrapcall.wtimemanager | 1 |
| | atm_physics1 | None |

Table 4.3: Standard run lists stored in the `runlists` attribute upon instantiation of a `Qtcm` instance. The run list and list element names are stored as strings.

```
foo = Field('lastday')
```

will return foo as a Field instance with foo.value set to the value listed in Section A.1. The value of all Field instances upon creation are specified in the defaults submodule of package qtcm, and listed in Sections A.1 and A.2.

To create Field instances whose attributes are set different from their defaults, you can specify the different settings in the instantiation parameter list, or change the attributes once the instance is created. See the Field docstring for details.

## 4.7.2 Initial Field Variables

Field variables include both model parameters that do not change for a Qtcm instance as well as prognostic variables that do change during model integration. As a result, many field variables have values different from the default values listed in Sections A.1 and A.2. In this section, I list the *initial* values of all field variables. The "initial" values are the settings for Qtcm field variables execution of the run_session method, but prior to cycling through an atmosphere-ocean coupling timestep. This is in contrast to "default" values, which the field variables are given on instantiation, if no other value is specified. Numerical values are rounded as per the conventions of Python's %g format code.

### Scalars

For the fields that give the input/output directory names, and the run name, the entry "value varies" is provided in the "Value" column.

| Field | Value | Units | Description |
|---|---|---|---|
| SSTdir | value varies | | Where SST files are |
| SSTmode | seasonal | | Decide what kind of SST to use |
| VVsmin | 4.5 | m/s | Minimum wind speed for fluxes |
| bnddir | value varies | | Boundary data other than SST |
| dateofmodel | 10101 | | Date of model coded as an integer as yyyymmdd |
| day0 | 1 | dy | Starting day; if < 0 use day in restart |
| dt | 1200 | s | Time step |
| eps_c | 0.000138889 | 1/s | 1/tau_c NZ (5.7) |
| interval | 1 | dy | Atmosphere-ocean coupling interval |
| it | 1 | | Time of day in time steps |
| landon | 1 | | If not 1: land = ocean with fake SST |
| lastday | 0 | dy | Last day of integration |
| month0 | 1 | mo | Starting month; if < 0 use mo in restart |

| Field | Value | Units | Description |
|---|---|---|---|
| mrestart | 0 | | =1: restart using qtcm.restart |
| mt0 | 1 | | Barotropic timestep every mt0 timesteps |
| nastep | 1 | | Number of atmosphere time steps within one air-sea coupling interval |
| noout | 0 | dy | No output for the first noout days |
| nooutr | 0 | dy | No restart file for the first nooutr days |
| ntout | -30 | dy | Monthly mean output |
| ntouti | 0 | dy | Monthly instantaneous data output |
| ntoutr | 0 | dy | Restart file only at end of model run |
| outdir | value varies | | Where output goes to |
| runname | value varies | | String for an output filename |
| title | value varies | | A descriptive title |
| u0bar | 0 | | |
| visc4x | 700000 | $m^2/s$ | Del 4 viscocity parameter in x |
| visc4y | 700000 | $m^2/s$ | Del 4 viscocity parameter in y |
| viscxT | 1.2e+06 | $m^2/s$ | Temperature diffusion parameter in x |
| viscxq | 1.2e+06 | $m^2/s$ | Humidity diffusion parameter in x |
| viscxu0 | 700000 | $m^2/s$ | Viscocity parameter for u0 in x |
| viscxu1 | 700000 | $m^2/s$ | Viscocity parameter for u1 in x |
| viscyT | 1.2e+06 | $m^2/s$ | Temperature diffusion parameter in y |
| viscyq | 1.2e+06 | $m^2/s$ | Humidity diffusion parameter in y |
| viscyu0 | 700000 | $m^2/s$ | Viscocity parameter for u0 in y |
| viscyu1 | 700000 | $m^2/s$ | Viscocity parameter for u1 in y |
| weml | 0.01 | m/s | Mixed layer entrainment velocity |
| year0 | 1 | yr | Starting year; if $<$ 0 use year in restart |
| ziml | 500 | m | Atmosphere mixed layer depth $\sim$ cloud base |

**Arrays**

| Field | Shape | Max | Min | Units | Description |
|---|---|---|---|---|---|
| Evap | (64, 42) | 1502.56 | 223.552 | | |
| FLW | (64, 42) | 74.5136 | 74.5136 | | |
| FLWds | (64, 42) | 206.424 | 206.424 | | |
| FLWus | (64, 42) | 429.708 | 429.708 | | |
| FLWut | (64, 42) | 148.771 | 148.771 | | |

| Field | Shape | Max | Min | Units | Description |
|---|---|---|---|---|---|
| `FSW` | (64, 42) | 147.767 | 0 | | |
| `FSWds` | (64, 42) | 410.895 | -6.99713 | | |
| `FSWus` | (64, 42) | 356.831 | -4.49983 | | |
| `FSWut` | (64, 42) | 332.431 | 0 | | |
| `FTs` | (64, 42) | 930.115 | 138.383 | | |
| `Qc` | (64, 42) | 0 | 0 | K | Precipitation |
| `S0` | (64, 42) | 534.264 | 0 | | |
| `STYPE` | (64, 42) | 3 | 0 | | Surface type; ocean or vegetation type over land |
| `T1` | (64, 44) | -100 | -100 | K | |
| `Ts` | (64, 42) | 295 | 295 | K | Surface temperature |
| `WD` | (64, 42) | 350 | 0 | | |
| `WD0` | (4,) | 500 | 0 | | Field capacity SIB2/CSU (approximately) |
| `arr1` | (64, 42) | 0 | 0 | | Auxiliary optional output array 1 |
| `arr2` | (64, 42) | 0 | 0 | | Auxiliary optional output array 2 |
| `arr3` | (64, 42) | 0.138699 | 0.138699 | | Auxiliary optional output array 3 |
| `arr4` | (64, 42) | 0 | 0 | | Auxiliary optional output array 4 |
| `arr5` | (64, 42) | 0 | 0 | | Auxiliary optional output array 5 |
| `arr6` | (64, 42) | 0 | 0 | | Auxiliary optional output array 6 |
| `arr7` | (64, 42) | 0 | 0 | | Auxiliary optional output array 7 |
| `arr8` | (64, 42) | 0 | 0 | | Auxiliary optional output array 8 |
| `psi0` | (64, 43) | 0 | 0 | | |
| `q1` | (64, 44) | -50 | -50 | K | |
| `rhsu0bar` | (3,) | 0 | 0 | | |
| `rhsvort0` | (64, 42, 3) | 0 | 0 | | |
| `taux` | (64, 42) | 0 | 0 | | |
| `tauy` | (64, 42) | 0 | 0 | | |
| `u0` | (64, 44) | 0 | 0 | m/s | Barotropic zonal wind |
| `u1` | (64, 44) | 0 | 0 | m/s | Current time step baroclinic zonal wind |
| `v0` | (64, 43) | 0 | 0 | m/s | Barotropic meridional wind |

| Field | Shape | Max | Min | Units | Description |
|---|---|---|---|---|---|
| v1 | (64, 43) | 0 | 0 | m/s | |
| vort0 | (64, 42) | 0 | 0 | | |

### 4.7.3 Passing Fields Between the Python and Fortran-Levels

Section 4.4.3 discusses the differences between how the `'full'` and `'parts'` compiled forms pass field variables between the Python and Fortran-levels. That discussion gives a detailed description of the methods used for passing fields to and from the Python and Fortran-levels (i.e., the `get_qtcm1_item` and `set_qtcm1_item` methods).

Please note the following regarding field variables as you pass them back and forth between the Python and Fortran-levels:

- Field variables with ghost latitudes, such as `u1`, on the Python end are always the full variables (i.e., including the ghost latitudes). On the Fortran end, variables like `u1` also always have the ghost latitudes while in the model, but when stored as restart files, do not have the ghost latitudes; the end points are not saved in restart files or written to the netCDF output files. See the QTCM1 manual[10] [4] for details about ghost latitudes.

- You should assume there is only a full synchronizing between compiled QTCM1 model and Python model field variables at the beginning and end of a run session.

- If you have a variable at the Python-level, but at the compiled QTCM1 Fortran model-level the variable is not readable, if you try to call `set_qtcm1_item` on the variable, nothing is done, and the Python-level value is left alone. If you have a compiled QTCM1 model variable, but no Python-level equivalent, if you call `set_qtcm1_item` on the variable, the Python-level variable (as an attribute) is created.

- To be precise, only compiled QTCM1 model variables can be passed pass back and forth between the Python and Fortran-levels; there are many `Qtcm` instance attributes that do not have any counterparts at the Fortran-level.[11]

- Although `dayofmodel` is described in module `setbypy` as an option for the `get_qtcm1_item` and `set_qtcm1_item` methods to operate on, in reality those methods cannot operate on `dayofmodel`, but `dayofmodel` is not defined in `defaults`.[12]

[10] http://www.atmos.ucla.edu/~csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

[11] I use the term "field variables" to refer to compiled QTCM1 model variables that can be passed back and forth to the Python level.

[12] All field variables must be defined in `defaults` in order for the proper Fortran routine to be called according to the variable's type.

### 4.7.4 Field Variable Shape

Normally, Python arrays have a different dimension order than Fortran arrays. While Fortran arrays are dimensioned (col, row, slice), with adjacent columns being contiguous, then rows, and then slices, Python arrays are dimensioned (slice, row, col), with adjacent columns being contiguous, then rows, and then slices. Based on this, you would think that everytime you passed an array between the Python and Fortran-levels you would need to transpose the array.

Thankfully, we don't have to do this because `f2py` handles array dimension order transparently so we can refer to each element the same way whether we're in Python or Fortran. Thus, the array `Qc` in Fortran is dimensioned (longitude, latitude), (64,42) by default, and the Python `Qtcm` instance attribute `Qc` has a `value` attribute also dimensioned (longitude, latitude), (64,42) by default. And at both the Fortran and Python-levels, the first longtude, second latitude element is referred to as `Qc(1,2)`.

In contrast, however, netCDF output saved by the compiled QTCM1 model and read into Python (using the `Scientific` package) is *not* in Fortran array order. Arrays read from netCDF output into Python are in Python array order, and are dimensioned (latitude, longitude) or (time, latitude, longitude). The `Qtcm` routines that manipulate netCDF data (e.g., `plotm`), however, automatically adjust for this, so you only need to be aware of this when reading in output for your own analysis (see Section 4.8).

## 4.8 Model Output

Section 3.7 gives an overview of how to use `qtcm` model output to netCDF files.

All netCDF array output is dimensioned (time, latitude, longitude) when read into Python using the `Scientific` package. This differs from the way `Qtcm` saves field variables, which follows Fortran convention (longitude, latitude). Thus, the shapes in Section 4.7.2, Appendix A, etc., are not the shapes of arrays read from the netCDF output. See Section 4.7.4 for a discussion of why there is this discrepancy.

Because netCDF files allow you to specify an "unlimited" dimension, it is possible to close a netCDF file, reopen it, and add more slices of data to the file. Thus, continuous `Qtcm` run sessions (i.e., those that use the `cont` keyword input parameter in the `run_session` method) will automatically append output to the netCDF output files.

Field variables with ghost latitudes, such as `u1`, on the Python and Fortran ends are always the full variables (i.e., including the ghost latitudes). The ghost latitudes are not written to the netCDF output files, however. See the QTCM1 manual[13] [4] for details about ghost latitude structure.

`Qtcm` instances have a few built-in tools to visualization model output. These are briefly described in Section 3.7.2. Note that the `plotm` method is linked to a specific

[13]http://www.atmos.ucla.edu/∼csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

```
inputs = {}
inputs['runname'] = 'test'
inputs['landon'] = 0
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 30
inputs['mrestart'] = 0
inputs['init_with_instance_state'] = True
inputs['compiled_form'] = 'parts'
```

Figure 4.1: The initial definition of the `inputs` dictionary for examples given in Section 4.10. These settings imply that a run session will start on November 1, Year 1, last for 30 days, and will be an aquaplanet run.

`Qtcm` instance. Do not use `plotm` outside of the instance it is linked to. It must also be used only after a successful run session (i.e., not in the middle of a run session).

## 4.9 Miscellaneous

A few miscellaneous items/issues about the model:

- The land model runs at same timestep as the atmosphere.
- If the land model runs less often than `sflux` in `physics1`, the calculation of evaporation over the land needs to be fixed in sflux.
- The units of some field variables are not what you would expect. For instance, `Qc` is in energy units, i.e., K, and not mm/day. See the QTCM1 manual[14] [4] for details.

## 4.10 Cookbook of Ways the Model Can Be Used

This cookbook of a few ways to use the model is arranged by science tasks, i.e., certain types of runs we want to do. For some of the examples below, I assume that the dictionary `inputs` is initially defined as given in Figure 4.1. All examples assume that `from qtcm import Qtcm` has already been executed.

**Plain model run:** Here I just want to make a single model run. Tasks: Instantiate a fresh model and execute a run session. The code to run the model is just:

[14]http://www.atmos.ucla.edu/~csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

```
inputs['init_with_instance_state'] = False
model = Qtcm(**inputs)
model.run_session()
```

where `inputs` is initialized with the code in Figure 4.1.

**Explore parameter space with a set of models:** Here I want to create an entire suite of separate models, in order to determine the sensitivity of the model to changing a parameter. To do this, I instantiate multiple fresh models, and execute a run session for each instance, all within a `for` loop:

```
import os
inputs['init_with_instance_state'] = False
for i in xrange(0,1002,10):
    iname = 'ziml-' + str(i) + 'm'
    ipath = os.path.join('proc', iname)
    os.makedirs(ipath)
    model = Qtcm(**inputs)
    model.ziml.value = float(i)
    model.runname.value = iname
    model.outdir.value = ipath
    model.run_session()
    del model
```

The loop explores mixed-layer depth `ziml` from 0 m to 1000 m, in 10 m intervals. I create the `outdir` directory before every model call, since the compiled QTCM1 model requires the output directory exist, specifying the run name and output directory as the string `iname`. The output directories are assumed to all be in the *proc* sub-directory of the current working directory. `inputs` is initialized with the code in Figure 4.1.

**Conditionally explore parameter space:** Here I want to conditionally explore the parameter space, on the basis of some mathematical criteria. To do this, I instantiate a model, evaluate results using that criteria, and run another fresh model depending on the results (passing the previous model state via a snapshot), all within a `while` loop. Note that this type of investigation is very difficult to automate if all you can use are shell scripts and Fortran. See Figure 4.2 for a detailed example.

**With interactive adjustments at run time:** The example in Figure 3.4.3 illustrates this type of run. In this example, I instantiate a fresh model, execute a run session, analyze the output, change variables in the model instance, and then execute a continuation run session.

**Test alternative parameterizations:** I've already described how we can use run lists to arbitrarily change model execution order and content at run time. We can take advantage of Python's inheritance abilities, along with run lists, to simplify this. Figure 4.3 provides an example of this use.

Of course, you can use pre-processor directives and shell scripts to accomplish the same functionality seen in Figure 4.3 using just Fortran. The Python solution, however, shortcuts the compile/linking step, and enables you to easily do run time swapping between subroutine choices based upon run time calculated tests (see Figure 4.2 for an example of such tests).

```
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
```

Figure 4.2: This code explores different values of mixed-layer depth `ziml` for 30 day runs, as a function of maximum `u1` magnitude, until it finds a case where the maximum `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 to initialize (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). The `inputs` dictionary is initialized with the code in Figure 4.1.

```
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
```

Figure 4.3: Let's say we have 9 different cloud physics schemes we wish to try out in 9 different runs. The easiest way to do this is to create a new class `NewQtcm` that inherits everything from `Qtcm`, and to which we'll add the additional cloud schemes (`cloud0`, `cloud1`, etc.). In the `for` loop, I change the cloud model run list entry in the run list that governs atmospheric physics at one instant to whatever the cloud model is at this point in the loop. The `inputs` dictionary is initialized with the code in Figure 4.1. Of course, we could do the same thing by running the 9 models separately, but this set-up makes it easy to do hypothesis testing with these 9 models. For instance, we can create a test by which we will choose which of the 9 models to use: Within this framework, the selection of those models can be altered by changing a string.

# Chapter 5

# Troubleshooting

## 5.1 Error Messages Produced by `qtcm`

`Error-Value too long in SetbyPy module getitem_str for` ***key*****:** This message is produced by the Fortran subroutine `getitem_str` in the module `SetbyPy` in the compiled QTCM1 Fortran code. The code is in the file *setbypy.F90*. This error occurs when the Fortran variable whose name is given by the string *key* has a value that is greater than the local parameter `maxitemlen` in `getitem_str`. To fix this, you have to go into *setbypy.F90* and change the value of `maxitemlen`.

`Error-real_rank1_array should be deallocated`**:** Fortran module `SetByPy`'s subroutine `getitem_real_array` generates this message (or a similar message for other ranks) if the Fortran variable for the input *key* are allocated on entry to the routine. This may indicate the user has written another Fortran routine to access the `real_rank1_array` variable outside of the standard interfaces..

`Error-Bad call to SetbyPy module ...`**:** Often times, this error occurs because a get or set routine in `SetByPy` tried to act on a variable for which the corresponding input *key* is not defined. The solution is to add that case in the if/then construct for the get and set routines in `SetByPy` and rebuild the extension modules.

## 5.2 Other Errors

**Python cannot find some packages:** This error often happens when the version of Python in which you have installed all your packages is not the version that is called at the Unix command line by typing in `python`. To get around this, define a Unix alias that maps `python2.4` (or whichever version of Python has all your packages installed) to `python`. If you have multiple Python's installed on your system, you might have to use a more specific name for the Python executable. As a result, you may have to change the test scripts in *test* in the `qtcm` distribution directory.

**`get_qtcm1_item` and compiled QTCM1 model pointer variables:** If you try to use the `get_qtcm1_item` method on a compiled QTCM1 model pointer variable (i.e., `u1`, `v1`, `q1`, `T1`), before the compiled model `varinit` subroutine is run, you'll get a bus error with no additional message.

**Mismatch between Python and Fortran array field variables:** You change an array field variable on the Python side, but it seems like the wrong elements are changed on the Fortran side. Or you type in the same index address for accessing a `qtcm` netCDF output array as well as its `Qtcm` instance attribute counterpart, and find you get different answers. Some possible reasons and fixes:

- This will occur if you haven't accounted for the difference in how field variables are saved at the Python-level, Fortran-level, and in a netCDF file. All netCDF array output is dimensioned (time, latitude, longitude) when read into Python using the `Scientific` package. This differs from the way `Qtcm` saves field variables, *both* at the Python- and Fortran-levels, which follows Fortran convention (longitude, latitude).

  Note that the way `Qtcm` saves field variables at the Python- and Fortran-levels is different than the default way Python and Fortran save arrays. Section 4.7.4 for more information.

- You may have forgotten that array indices in Python start at 0, while indices in Fortran (generally) start at 1. Also, ranges in Python are exclusive at the upper-bound, while ranges in Fortran are inclusive at the upper-bound. (Both Python and Fortran array indice ranges are inclusive at the lower-bound.)

- You may have forgotten some field variables have ghost latitudes, and thus there are extra latitude bands when the array is stored as a Python or Fortran field variable, but there are *no* extra latitude bands when the array is stored as netCDF output (the QTCM1 output routines strip off the ghost latitudes when writing those field variables out). See the QTCM1 manual[1] [4] for details about ghost latitudes.

  The safest and easiest way to tell whether the variable has a ghost latitudes is to look at its shape. A call to the `Qtcm` instance method `get_qtcm1_item` will give you the array, and the use of NumPy's `shape` function will give you the shape.

[1]http://www.atmos.ucla.edu/∼csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

# Chapter 6

# Developer Notes

## 6.1 Introduction

This section describes programming practices and issues related to the `qtcm` package that might be of interest to users wishing to add/change code in the package. Please see the package API documentation,[1] which includes the source code, for details.

## 6.2 Changes to QTCM1 Fortran Files

The source code used to generate the shared object files used in this Python package is unchanged from the pure-Fortran QTCM1 model source code, except in the following ways:

- The suffix of all source code files has been changed from *.f90* to *.F90*, in order to ensure the compiler preprocesses the source code. Some compilers use the capitalization to tell whether or not to run the code through a preprocessor.

- In all *.F90* files, occurrences of:

  ```
  Character(len=130)
  ```

  are changed to:

  ```
  Character(len=305)
  ```

  This enables the model to properly deal with longer filenames. The number "305" is chosen to make search and replace easier.

- In *qtcmpar.F90*, the `eps_c` variable is changed from an unchangable parameter to a changeable real, so that it can be changed in the model at runtime.

[1]http://www.johnny-lin.com/py_pkgs/qtcm/doc/html-api/

- All occurrences of an underscore ("_") character in a subroutine or function name are removed. The presence of the underscore messes up the dynamic lookup mechanism for the `f2py` generated extension module. The following names are changed, both in subroutine definitions and calls:
  - `out_restart` to `outrestart`,
  - `save_bartr` to `savebartr`,
  - `grad_phis` to `gradphis`.
- *driver.F90* is changed so that program `driver` becomes a subroutine, and subroutine `driverinit` is deleted (along with all calls to it) because basic model initialization is handled at the Python level.
- In *clrad.F90*, subroutine `cloud`, the first `COUNTCAP` preprocessor macro, a comment line for that ifdef is moved to prevent a warning message during building with `f2py`.
- The order of subroutine `qtcminit` is changed. The original pure-Fortran QTCM1 `qtcminit` code has the following calling sequence:

  ```
  Call parinit !Initialize model parameters
  Call varinit !Initialize variables
  Call TimeManager(1) !mm set model time
  Call bndinit !input boundary datasets
  Call physics1 !diagnostic fields for initial condition
  ```

  For the `qtcm` package, I've altered this order so `bndinit` comes after `parinit` but before `varinit`:

  ```
  Call parinit !Initialize model parameters
  Call bndinit !input boundary datasets
  Call varinit !Initialize variables
  Call TimeManager(1) !mm set model time
  Call physics1 !diagnostic fields for initial condition
  ```

  This is done because `STYPE` is not read in for the `landon True` case until `bndinit`, but in `varinit STYPE` is used to calculate the original values of `WD` for the non-restart case. This also corrects the conflicting order found in the pure-Fortran QTCM1 manual (compare pp. 29 and 32). As far as I can tell, `bndinit` has no dependencies that require it to come after `timemanager` or `varinit`.

In addition, the Fortran files *setbypy.F90*, *wrapcall.F90*, and *varptrinit.F90* are added. The routines in these files, however, just add more flexibility and functionality to the model; they do not automatically affect any model computations. See Section 6.3 for details.

## 6.3 New Interfaces and Fortran Functionality

As described in Section 6.2, the Fortran files *setbypy.F90*, *wrapcall.F90*, and *varptrinit.F90* are added to the QTCM1 source directory. The first two files define the Fortran 90 modules (`SetbyPy` and `WrapCall`) needed to interface the Python and Fortran levels. The last file defines a new Fortran subroutine `varptrinit` that associates QTCM1 model pointer variables at the Fortran level. In a pure-Fortran run of QTCM1, this occurs in subroutine `varinit`; for a `compiled_form = 'parts'` run, since the functionality of the Fortran `varinit` is now in the Python `varinit` method, a separate Fortran pointer association subroutine needed to be defined. The Fortran subroutine `varptrinit` is called as the `varptrinit` function of the `compiled_form = 'parts'` *.so* extension module.

### 6.3.1 Fortran Module `SetbyPy`

#### Design Description

This module defines functions and subroutines used to read variables from the Fortran-level to the Python-level, and in setting Fortran-level variables using the Python-level values. These routines are used by `Qtcm` methods `get_qtcm1_item` and `set_qtcm1_item` (and dependencies thereof) to "get" and "set" the Fortran-level variables. Note that the Fortran module `SetbyPy` is referred to in lowercase at the Python level, i.e., as the attribute `__.qtcm.setbypy` of a `Qtcm` instance.

Because Fortran variables are not dynamically typed, separate Fortran functions and subroutines need to be defined to get and set variables of different types.[2] The `Qtcm` methods `get_qtcm1_item` and `set_qtcm1_item` know which one of the Fortran routines to call on the basis of the type and rank of the value for the field variable in the `defaults` submodule. This is why all field variables need to have defaults defined in `defaults`. For array variables, the field variable defaults also provide the rank of the Fortran-level variable being gotten or set. However, the array default values do *not* have to have the same shape as the Fortran-level variables; on the Python-side, variable shape adjusts to what is declared on the Fortran-side. Thus, if you change the resolution of the compiled QTCM1 model, you do not have to change the dimensions of the field variable values in `defaults`.

The `Qtcm` method `get_qtcm1_item` directly calls the `SetByPy` routines. The `Qtcm` method `set_qtcm1_item` makes use of private instance methods that make the calls

[2]The `interface` construct in Fortran 90 is supposed to provide a way to create a single interface to multiple routines, e.g.:

```
Interface setitem
  Module Procedure setitem_real, setitem_int, setitem_str
End Interface
```

This construct, however, causes a bus error (Mac OS X 10.4, Intel). Thus, I put the same functionality in the Python code.

to the `SetByPy` routines.

For scalar field variables, `SetByPy` provides functions and subroutines that provide the value of the variable on output. For array field variables, `SetByPy` dynamic *module* arrays are used to pass array variables in and out; I could not get the `SetByPy` Fortran routines to set locally defined dynamic arrays (that is, locally within a function or subroutine).[3] In the `SetByPy` module, these dynamic arrays are defined as follows:

```
Real, allocatable, dimension(:)  ::  real_rank1_array
Real, allocatable, dimension(:,:)  ::  real_rank2_array
Real, allocatable, dimension(:,:,:)  ::  real_rank3_array
```

For all field variables, scalar or array, the `SetByPy` module has a fourth module variable defined, `is_readable`, that the Fortran get and set routines will set to `.TRUE.` if the variable is readable and `.FALSE.` if not (it's declared as a logical variable). This Fortran variable can be used to prevent Python from accessing pointer variables that aren't yet associated to targets.

In general, `SetByPy` routines make use of Fortran constructs to enable them to accomodate all possible variables of a given type and shape. However, for string scalars, the `SetByPy` function `getitem_str` has to have a return value of a predefined length, in order to work properly. That length is given by the parameter `maxitemlen` and is set to 505 (the value is chosen to be larger than all filename variables described in Section 6.2 and to be easily found in the *.F90* files).

### Module Structure

If you're a Fortran programmer, you can probably get all the information in this section from just reading the *setbypy.F90* file directly. This description of the module structure, however, permits me to highlight what you need to do if you want to make additional compiled QTCM1 variables accessible to Python `Qtcm` objects.

- All `Use` statements are given in the beginning of the `SetByPy` module. These statements cover nearly all of the QTCM1 Fortran modules that contain variables of interest. If the QTCM1 variable you're interested in isn't in a module listed here, you'll have to add your own `Use` statement of that module here.

- Next comes the definitions for the `real_rank1_array`, `real_rank2_array`, and `real_rank3_array` dynamic array variables, and the `is_readable` boolean variable.

- The `Contains` block of the module defines the module routines called by the `Qtcm` instance methods to set and get the compiled QTCM1 model variables. The routines are:

[3] I tried to implement Fortran subroutine `getitem_real_array` using traditional array dimension passing (e.g., `subroutine foo(nx, ny, a)`) as well as declaring the allocatable array inside the subroutine, but neither option worked on my `f2py` (version 2_3816) and Python (version 2.4.3).

- Function `getitem_real`
- Subroutine `getitem_real_array`
- Function `getitem_int`
- Function `getitem_str`
- Subroutine `setitem_real`
- Subroutine `setitem_real_array`
- Subroutine `setitem_int`
- Subroutine `setitem_str`

Each of the routines in the module `Contains` block is essentially a list of `if`/`elseif` statements. The list tests for the name of the variable of interest (a string), and gets or sets the compiled QTCM1 model variable corresponding to that name. For pointer array variables, a test is also made as to whether or not the variable has been associated. If not, the variable is not readable and `is_readable` is set to `.FALSE.` accordingly.

If you wish to add another compiled QTCM1 model variable to be accessible to `Qtcm` instance methods `get_qtcm1_item` and `set_qtcm1_item`, just add another `if`/`elseif`, like the other `if`/`elseif` blocks, in the Fortran set and get routines corresponding to the QTCM1 variable type (scalar vs. array, and real, integer, or string). On the Python side, add an entry in `defaults` corresponding to the new field variable you've created access to. I would strongly recommend making the Python name of your new field variable (given in `defaults`) be the same as the compiled QTCM1 model variable name.

### 6.3.2 Fortran Module `WrapCall`

Most of the time, if you want to call a compiled QTCM1 model subroutine from the Python level, you will use the version of the subroutine that is found in this Fortran module. Note that the Fortran module `WrapCall` is referred to in lowercase at the Python level, i.e., as the attribute `__.qtcm.wrapcall` of a `Qtcm` instance.

All the routines in this module do is wrap one of the compiled QTCM1 model routines. For instance, `WrapCall` subroutine `wadvcttq` is defined as just:

```
Subroutine wadvcttq
   Call advcttq
End Subroutine wadvcttq
```

All subroutines in this module begin with “w”, with the rest of the name being the Fortran QTCM1 subroutine name. The calling interface for the “w” version is the same as the Fortran QTCM1 original version. There are no subroutines in this module that do not have an exact counterpart in the Fortran QTCM1 code, and thus this module's subroutines sole purpose is to call other subroutines in the compiled QTCM1 model.

These wrapper routines are needed because `f2py`, for some reason I can't figure out, will not properly wrap Fortran routines (that are then callable at the Python level) that create local arrays using parameters obtained through a Fortran `use` statment. Thus, as an example, a Fortran subroutine `foo` with the following definition:

```
subroutine foo
  use dimensions
  real a(nx,ny)
  [...]
end subroutine foo
```

where `nx` and `ny` are defined in the module varsdimensions, will return an error, with the result that the extension module will not be created, or an extension modules that yields output that is different from running the pure-Fortran version of QTCM1.

By wrapping these calls into this file, I also avoid having to separate out the Fortran QTCM1 subroutines into separate *.F90* files. For Fortran subroutines that you want callable from the Python level, `f2py` seems to require each Fortran subroutine to be in its own file of the same name (e.g., the version of *driver.F90* for this package). If several Fortran subroutines are all found in a single *.F90* files, `f2py` seems unable to create wrappers for those subroutines.

## 6.4 Python `qtcm` and Pure-Fortran QTCM1 Differences

This section describes differences between how the `qtcm` package and the pure-Fortran QTCM1 assign some varables. A list of changes to the QTCM1 Fortran Files for use in the `qtcm` package is found in Section 6.2.

### 6.4.1 QTCM1 `driverinit`

In the pure-Fortran version of QTCM1, by default, the following variables are set by reference (as given below), not by value, in the `driverinit` routine:[4]

[4]In the pure-Fortran version of QTCM1, this routine is found in *driver.F90*.

```
lastday = daysperyear
viscxu0 = viscU
viscyu0 = viscU
visc4x = viscU
visc4y = viscU
viscxu1 = viscU
viscyu1 = viscU
viscxT = viscT
viscyT = viscT
viscxq = viscQ
viscyq = viscQ
```

Thus, in pure-Fortran QTCM1, if you change `daysperyear`, `viscU`, etc. and recompile (as needed), you will automatically change `lastday`, `viscxu0`, etc. (Though, in the pure-Fortran QTCM1, the default values may be overwritten by namelist input values.)

The `driverinit` routine is eliminated in the Python `qtcm` package. Instead, inital values of field variables are specified in the `defaults` submodule and set by value to attributes of the `Qtcm` instance. Thus, for instance, in a `Qtcm` instance, `lastday` is set to `365` by default, not to some variable `daysperyear`. For the diffusion and viscosity terms, the `Qtcm` instance attributes corresponding to those terms are set to literals.[5]

In contrast, in the pure-Fortran QTCM1, `driverinit` declares local variables `viscU`, `viscT`, and `viscQ`, and reads values into those variables via the input namelist. Those values are then used to set `viscxu0`, `viscyu0`, etc., as described above. In pure-Fortran QTCM1, `viscU`, `viscT`, and `viscQ` are not directly accessed anywhere else in the model. Thus, `viscU`, `viscT`, and `viscQ` are not defined as field variables in the `qtcm` package, and `Qtcm` instances do not have attributes corresponding to those names. Additionally, if you wish to change a viscosity parameter `visc*` (given above), the parameter for each direction must be set one-by-one even if the flow is isotropic.

### 6.4.2 The `varinit` Routine

One of the functions of the pure-Fortran QTCM1 `varinit` subroutine is to associate the pointer variables `u1`, `v1`, `q1`, and `T1`. For the extension modules in the `qtcm` package, a Fortran subroutine `varptrinit` is added that can also do this association. This subroutine is called in the `Qtcm` instance method `varinit`[6] (which duplicates and extends the function of its pure-Fortran counterpart, enabling alternative ways of handling restart).

The `varptrinit` is not accessed via `wrapcall`. Remember that `wrapcall` contains only those routines that were in the original pure-Fortran QTCM1 code, and that we want to have access to at the Python level.

[5]Those literals are defined by `defaults` private module variables `__viscT`, `__viscQ`, and `__viscU`.
[6]http://www.johnny-lin.com/py_docs/qtcm/doc/html-api/qtcm.qtcm.Qtcm-class.html#varinit

### 6.4.3 The `qtcm` Method of `Qtcm`

The `Qtcm` method `qtcm` duplicates the functionality of the `qtcm` subroutine in the pure-Fortran QTCM1 model. There are a few differences, however. First, the `qtcm` method for `Qtcm` instances does not include a call to `cplmean`, which uses mean surface flux for air-sea coupling. This state is consistent with the pure-Fortran QTCM1 preprocessor macro `CPLMEAN` being off. Thus, if you wish to use mean surface flux for air-sea coupling, you will have to revise the `qtcm` method of `Qtcm` to call `cplmean`. You'll also have to check for any other code additions needed that are associated with the `CPLMEAN` macro.

Second, the `qtcm` method for `Qtcm` instances does not include the option of not using the atmospheric boundary layer model. This is consistent with macro `NO_ABL` being off. If you wish to have no atmospheric boundary layer model, change the run list `atm_bartr_mode` so that the `wsavebartr` and `wgradphis` routines are not called. You'll also have to check for any other code additions needed that are associated with the `NO_ABL` macro.

### 6.4.4 Miscellaneous Differences

- In Python `Qtcm` instances, `dateofmodel` is set to 0 by default. In contrast, in the compiled QTCM1 model, the default (i.e., initial value) is calculated from `day0`, `month0`, and `year0`. See Section 4.4.1 for details.

- The `Qtcm` instance attribute `__qtcm` is not copyable using `copy.deepcopy`.

- In general, when executing a `Qtcm` instance method, if you change a `Qtcm` instance attribute that has a counterpart in the compiled QTCM1 model, the compiled QTCM1 counterpart is not changed until the end of the method. Likewise, if you call a compiled QTCM1 model subroutine and change a compiled QTCM1 model variable with a `Qtcm` instance counterpart, the `Qtcm` instance counterpart is not changed until the end of the subroutine.

- In general, even though some of the compiled QTCM1 model Fortran subroutines/functions have counterparts in `Qtcm` that duplicate the former's functionality, the Fortran versions are kept intact so that the `compiled_form = 'full'` case will work.

## 6.5 Considerations When Adding Fortran Code

In this section I describe issues to consider if you wish to add your own compiled code to the package as separate extension modules. (This is different from creating new standard extension modules, which is described in Section 6.6.):

- The `Qtcm` class assumes that the directory path to the original shared object file is the same as for the `package_version` module.
- If you want to be able to pass other Fortran variables in and out to/from Python, please see the Section 6.3.1 discussion of the Fotran `SetByPy` module.
- Fortran and Python routines to get and set compiled QTCM1 model arrays are currently written only for floating point array.
- If you ever change `Qtcm` instance method `_set_qtcm_array_item_in_model` to work with non-floating point values, you will also have to change the array handling section in `set_qtcm1_item`.
- The restart mechanism in the pure-Fortran QTCM1 model is *not* bit-for-bit correct. Thus, if you compare the final output from a 40 day run with a 30 day run restarted from a 10 day run, the output will not be the same. This behavior has been duplicated in `Qtcm` instances when the `mrestart` flag is used and applicable.
- When creating new extension modules using the *src* makefile, be sure you first use the `make clean` command to clean-up any old files.

## 6.6 Creating New Standard Extension Modules

The steps involved in creating the standard extension modules (e.g., *_qtcm_full_365.so*, etc.) on installation are given in Section 2.4. The makefile provided in */buildpath/src* uses a Fortran compiler to create the object code, runs `f2py` to create the shared object file in *src*, and moves the shared object files into *../lib*, overwriting any pre-existing files of the same name. In this section, I describe the makefile and `f2py` in a little more detail, in case you wish to create standard extension modules with additions from the ones the default makefile creates.

### 6.6.1 Makefile Rules

This section describes the rules of the makefile found in the *src* directory of the `qtcm` distribution. This makefile is used by the Python package to create the extension module (*.so* files) imported and used by `qtcm` objects (as described in Section 2.4). The makefile will, in general, be used only during `qtcm` installation, but if you wish to recompile the QTCM1 libraries and make changes in the Python extension module, you'll want to use/change this makefile.

**clean** Removes old files in preparation for compiling new extension modules.

**libqtcm.a** Creates library *libqtcm.a* that contains all QTCM1 object files in the directory *src*,, except *setbypy.o*, *wrapcall.o*, *varptrinit.o*, and *driver.o*. This archive is compiled with the netCDF libraries. Previous versions of *libqtcm.a* are overwritten.

**_qtcm_full_365.so** Creates the extension module *_qtcm_full_365.so*. `f2py` is run on applicable code in *src*, and the extension module is moved to *../lib*. Any previous versions of *../lib/_qtcm_full_365.so* are overwritten.

**_qtcm_parts_365.so** Creates the extension module *_qtcm_parts_365.so*. `f2py` is run on applicable code in *src*, and the extension module is moved to *../lib*. Any previous versions of *../lib/_qtcm_parts_365.so* are overwritten.

### 6.6.2 Using `f2py`

This section briefly describes how `f2py` is used in the makefile during the creation of the extension modules. `F2py` is a program that generates shared object libraries that allow you to call Fortran routines in Python. `F2py` comes with Python's NumPy array handling package, so you do not need to install anything extra if you have NumPy already installed.

To create the extension modules in `qtcm` using the makefile described in Section 6.6.1, I use a method similar to the "Quick and Smart Way,"[7] described in the `f2py` manual. For the *_qtcm_full_365.so* extension module, the `f2py` call is:

```
f2py --fcompiler=$(FC) -c -m _qtcm_full_365 driver.F90 \
        setbypy.F90 libqtcm.a $(NCLIB)
```

and for the *_qtcm_parts_365.so* extension module, the call is:

```
f2py --fcompiler=$(FC) -c -m _qtcm_parts_365 \
        varptrinit.F90 wrapcall.F90 setbypy.F90 \
        libqtcm.a $(NCLIB)
```

For both calls, `FC` and `NCLIB` are the environment variables in the makefile specifying the Fortran compiler and netCDF libraries, respectively. The `-m` flag specifies the extension module name (without the *.so* suffix). The *.F90* files specify the files that have modules and routines that will be accessible at the extension module level, and the rest of the Fortran files in QTCM1 are compiled and archived in a library *libqtcm.a*. For `f2py` to work properly, the *.F90* files may define *only one* module or routine.

If you add Fortran files containing new modules, and you wish those modules to be accessible at the Python level, compile your new code with `f2py`. If we have a file of such new code, *newcode.F90*, the `f2py` call to create the *_qtcm_parts_365.so* extension module will become:

[7] http://cens.ioc.ee/projects/f2py2e/usersguide/index.html#the-quick-and-smart-way

```
    f2py --fcompiler=$(FC) -c -m _qtcm_parts_365 \
            varptrinit.F90 wrapcall.F90 setbypy.F90 \
            newcode.F90 \
            libqtcm.a $(NCLIB)
```

If you write new Fortran code for the compiled QTCM1 model that will *not* be accessed from the Python-level, just add the object code filename to the variable `QTCMOBJS` in the makefile; you don't have to do anything else. If you are adding Fortran code to existing Fortran modules, it's even easier: You don't need change the makefile. Note that for 64 bit processor machines, you may have to use `f2py` with the `-fPIC` flag; see Section 2.8.4 for details on how the lines above will change.

### 6.6.3 Two Examples

**A Function:** Let's say you have written a piece of Fortran code called *myfunction.F90* that contains one function called `myfunction`, and you want to have this function callable from the Python level through the `Qtcm` instance method `__qtcm.myfunction`. Do the following:

1. Move *myfunction.F90* to *src* in the `qtcm` distribution directory */buildpath*.

2. Add `myfunction.o` to the end of the object file list lines after the target names `_qtcm_full_365.so` and `_qtcm_parts_365.so`.

3. In the `_qtcm_full_365.so` and `_qtcm_parts_365.so` target descriptions, add `myfunction.F90` to the beginning of the list of *.F90* names in the `f2py` lines.

**A Module:** Let's say you have written a piece of Fortran code called *mymodule.F90* that contains the Fortran module `MyModule` containing multiple routines and variables. You want to have those routines and variables callable from the Python level through the `Qtcm` instance attribute `__qtcm.mymodule`. The steps to add `MyModule` to the extension modules are exactly the same as for a single function, with `mymodule` being substituted in the makefile everywhere you have `myfunction`.

## 6.7 Attributes and Methods in `Qtcm` Instances

In this section I describe some attributes, particularly private ones, that may be of interest to developers. As is the convention in Python, private attributes and methods are prepended by one or two underscores, with two underscores being the "more" private attribute. Please see the package API documentation[8] for details about all variables, including private variables.

[8]http://www.johnny-lin.com/py_pkgs/qtcm/doc/html-api/

### 6.7.1 Public `num_settings` Submodule Attributes/Methods

- `typecode`: This module function returns the type code of the data array passed in as its argument.

- `typecodes`: This dictionary is the same as the NumPy (or Numeric and `numarray`) dictionary `typecodes`, except that the character `'S'` and `'c'` are added to the `typecodes['Character']` entry, if absent. This functionality is added because I found `typecodes['Character']` had different values in Mac OS X and Ubuntu GNU/Linux.

### 6.7.2 Private `qtcm` Submodule Attributes

This submodule of the package `qtcm` is the module that defines the `Qtcm` class.

- `_init_prog_dict`: This dictionary contains the default values of all prognostic variables and right-hand sides that can be initialized. In the submodule `qtcm`, it is set to the `init_prognostic_dict` module variable in submodule `defaults`.

- `_init_vars_keys`: List of all keys in `_init_prog_dict`, plus `'dateofmodel'` and `'title'`. These names correspond to the field variables that are usually written out into a restart file.

- `_test_field`: `Field` object instance used in type tests.

### 6.7.3 Private `Qtcm` Attributes

- `_cont`: A boolean attribute that is `True` if the run session is a continuation run session and `False` if not. Set the value passed in by the keyword `cont` when the `run_session` method is executed.

- `_monlen`: Integer array of the number of days in each month, assuming a 365 day year.

- `__qtcm`: The extension module that is the compiled QTCM1 Fortran model for this instance. This attribute is unique for every instance: The extension module *.so* file is first copied to a temporary directory (given by the `sodir` instance attribute) and then imported to the `Qtcm` instance. This private attribute is set on instantiation.

- `_qtcm_fields_ids`: Field ids for all default field variables, set on instantiation.

- `_runlists_long_names`: Dictionary holding the descriptions of the standard run lists. The keys of the dictionary are the names of the standard run lists.

## 6.8 Creating Documentation

The distribution of `qtcm` comes with the full set of documentation in readable form (PDF and HTML). The documentation consists of two kinds: this User's Guide and the API documentation. The User's Guide is written in LaTeX. The PDF version is generated directly from LaTeX, and the HTML version is created by LaTeX2HTML.

I use the *make_docs* shell script in *doc* creates all these documents. Briefly, that script does the following:

- In the *doc/latex* directory, uses `python` to run *code_to_latex.py*, which generates the LaTeX files describing the current `qtcm` package settings, including text in the manual which gives all uses of the current version number.

- LaTeX is run on the LaTeX files in the *doc/latex* directory. The PDF generated by the run is moved from *doc/latex* to *doc*.

- LaTeX2HTML is run on the LaTeX files in *doc/latex*. The HTML files generated by the run are moved to *doc/html*.

- `epydoc` is run on the `qtcm` package libraries. This is run in *doc*, to make use of the *epydoc* configuration file present there. The syntax from the command line is:

  ```
  epydoc -v --config epydocrc [name]
  ```

  `[name]` is either `qtcm`, if the `qtcm` package is installed in a directory listed in `sys.path`, or `[name]` is the name of the directory the `qtcm` package is located in (e.g., */usr/lib/python2.4/site-packages/qtcm*).

The *make_docs* script cannot be used without customizing it to your system, so please **DO NOT USE IT** if you do not know what you are doing. You could easily wipe out all your documentation by mistake.

# Chapter 7

# Future Work

This section describes the features and fixes I plan to work on in this package. The most urgent items are listed closer to the begining of the lists.

- Automate the installation using Python's `distutils`[1] utilities.
- Describe a way of using job control (either via the operating system or IPython's `jobctrl` module) to do a quick-and-dirty parallelization of multiple `Qtcm` instance run sessions. Or use some sort of threading to fire up two simulataneously running models. Check that the simultaneously running models have different memory space.
- Add capability for *create_benchmark.py* to overwrite existing benchmark files.
- Make `compiled_form` set to `'parts'` as the default instantiation. Change documentation accordingly.
- Currently, the `Qtcm` `plotm` method works only on 3-D output (time, latitude, longitude). Some of the fields in the netCDF output files are 2-D. Add the capability to `plot_netcdf_output` in the `plot` submodule to handle 2-D fields.
- Add documentation about removing temporary files. Add documentation in Section 4.2 of details of what occurs during instantiation of a `Qtcm` instance.
- Add the units and long names for all field variables in the `defaults` module.
- Create a keyword to automatically change precipitation and evaporation units to mm/day (or similar).
- Add ability to calculate and plot fields at different pressure levels. Create another module like defaults that specifies the vertical fields and gives the equation to use to calculate those fields; call the module "derivfields" or something similar.

[1]http://docs.python.org/dist/dist.html

- Throughout the `qtcm` package I use the condition `N.rank(`*arg*`) = 0` to test whether *arg* is a scalar. This works fine for `numpy` objects, but it does not work properly for `Numeric` and `numarray` arrays. In those array packages, `rank('abc')` returns the value 1. This is not a problem, as long as everyone has `numpy`, but in order to make the package interoperable, I need to find a better way of testing for scalars. The definitions of isscalar need to be changed in `num_settings`.

- `num_settings` needs to be changed to truly enable me to test whether `qtcm` works for `numarray` and `Numeric` arrays. The tests do not do this right now, because `num_settings` defaults to `numpy`, if it exists.

- Create makefiles for other platforms.

- A few fields (e.g., `u1`) have data for extra latitude bands, due to the use of "ghost latitudes" as part of the implementation of the numerics. Details are found in the QTCM1 manual[2] [4].

  Though adjusting to this idiosyncracy is not that difficult, in the future I hope to implement a method of handing fields with ghost latitudes so that they have the same dimensions as the other gridded output variables. In order to do this, I plan to write a Python method to read the Fortran generated binary restart file.

- Change the `set_qtcm_item` method so that it can automatically accomodate setting Fortran real variables if integer values are input.

- Currently, the `get_item_qtcm` and `set_item_qtcm` methods will not work on integer and character arrays, only scalars and real arrays. Add that missing functionality to those methods.

- Currently, the `make_snapshot` method duplicates the functionality of the pure-Fortran QTCM1 restart file mechanism. However, the restart file mechanism itself does not do a true restart. A continuous run does not provide the same results as two runs over the same period, joined by the restart file.

  To see whether saving more variables would do the trick, I altered `make_snapshot` to store all Python level variables (i.e., `self._qtcm_fields_ids`). However, the restart failing described above still continued. In the future, I hope to figure out exactly how many variables are needed in order to make the restart feature do a true restart.

- Add a test of using the `mrestart = 1` restart option. Does the *qtcm.restart* file need to be in the current working directory or another?

[2]http://www.atmos.ucla.edu/∼csi/qtcm_man/v2.3/qtcm_manv2.3.pdf

- Add a test in the unit test scripts to confirm that the `init_with_instance_state` attribute setting only has an effect if `compiled_form = 'parts'`.

- Document `tmppreview` keyword in `plot.plot_ncdf_output`.

- Confirm and document that for netCDF output, time is model time since dd-mm-yyyy.

- Add to the `plotm` method the ability to plot as text onto the figure the runname string and the calling line for the plotm method.

- Couple with the CliMT[3] climate modeling toolkit.

- Enable Python to set `arr1name`, etc., which are string variables at the Python level. I haven't really thought through how `arr1` variables work with the Python `Qtcm` instance.

- Possible: In the `Qtcm` method `__setattr__`, add a test to raise an exception if the instance tries to set `viscU`, `viscT`, or `viscQ` as attributes. Also create a method `isotropic_visc` that will set all viscosity parameters non-dependent on direction. See Section 6.4.1 for details.

- Go through the manual and create HTML-only versions of tables that have table numbers (use a similar construct as in figure environments).

- Go through documentation to check that output variable names are capitalized consistently.

- Create way to redirect stdout.

- Create a step method to run an arbitrary number of timesteps at the atmosphere level.

[3]http://maths.ucd.ie/~rca/climt/

# Bibliography

[1] A. K. Betts and M. J. Miller. A new convective adjustment scheme. Part II: Single column tests using GATE wave, BOMEX, ATEX and Arctic air-mass data sets. *Quart. J. Roy. Meteor. Soc.*, 112:693–709, 1986.

[2] Johnny Wei-Bing Lin. *The Effects of Evaporation-Wind Feedback, Mid-Latitude Storms, and Stochastic Convective Processes on Tropical Intraseasonal Variability.* Ph.D. Dissertation, University of California, Los Angeles, 2000.

[3] J. David Neelin and Ning Zeng. A quasi-equilibrium tropical circulation model—formulation. *J. Atmos. Sci.*, 57(11):1741–1766, June 1, 2000.

[4] J. David Neelin, Ning Zeng, Chia Chou, Johnny Lin, Hui Su, Matthias Munnich, Katrina Hales, and Joyce Meyerson. *The Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (QTCM1), Version 2.3.* UCLA Department of Atmospheric Sciences, Los Angeles, 2002.

[5] Ning Zeng, J. David Neelin, and Chia Chou. A quasi-equilibrium tropical circulation model—implementation and simulation. *J. Atmos. Sci.*, 57:1767–1796, 2000.

# Appendix A

# Field Settings in `defaults`

## A.1 Scalar Field Variables

This table lists the default settings for scalar `qtcm` fields as set by the `defaults` submodule. All fields are of class `Field`. Numerical values are rounded as per the conventions of Python's `%g` format code. To create a `Field` instance whose value is set to the default, instantiate with the field id as the argument

| Field | Value | Units | Description |
|---|---|---|---|
| `SSTdir` | ../bnddata/SST_Reynolds | | Where SST files are |
| `SSTmode` | seasonal | | Decide what kind of SST to use |
| `VVsmin` | 4.5 | m/s | Minimum wind speed for fluxes |
| `bnddir` | ../bnddata | | Boundary data other than SST |
| `dateofmodel` | 0 | | Date of model coded as an integer as yyyymmdd |
| `day0` | -1 | dy | Starting day; if $< 0$ use day in restart |
| `dt` | 1200 | s | Time step |
| `eps_c` | 0.000138889 | 1/s | 1/tau_c NZ (5.7) |
| `interval` | 1 | dy | Atmosphere-ocean coupling interval |
| `it` | 1 | | Time of day in time steps |
| `landon` | 1 | | If not 1: land = ocean with fake SST |
| `lastday` | 365 | dy | Last day of integration |
| `month0` | -1 | mo | Starting month; if $< 0$ use mo in restart |

| Field | Value | Units | Description |
|---|---|---|---|
| mrestart | 0 | | =1: restart using qtcm.restart |
| mt0 | 1 | | Barotropic timestep every mt0 timesteps |
| nastep | 1 | | Number of atmosphere time steps within one air-sea coupling interval |
| noout | 0 | dy | No output for the first noout days |
| nooutr | 0 | dy | No restart file for the first nooutr days |
| ntout | -30 | dy | Monthly mean output |
| ntouti | 0 | dy | Monthly instantaneous data output |
| ntoutr | 0 | dy | Restart file only at end of model run |
| outdir | ../proc/qtcm_output | | Where output goes to |
| runname | runname | | String for an output filename |
| title | QTCM default title | | A descriptive title |
| u0bar | 0 | | |
| visc4x | 700000 | $m^2/s$ | Del 4 viscocity parameter in x |
| visc4y | 700000 | $m^2/s$ | Del 4 viscocity parameter in y |
| viscxT | 1.2e+06 | $m^2/s$ | Temperature diffusion parameter in x |
| viscxq | 1.2e+06 | $m^2/s$ | Humidity diffusion parameter in x |
| viscxu0 | 700000 | $m^2/s$ | Viscocity parameter for u0 in x |
| viscxu1 | 700000 | $m^2/s$ | Viscocity parameter for u1 in x |
| viscyT | 1.2e+06 | $m^2/s$ | Temperature diffusion parameter in y |
| viscyq | 1.2e+06 | $m^2/s$ | Humidity diffusion parameter in y |
| viscyu0 | 700000 | $m^2/s$ | Viscocity parameter for u0 in y |
| viscyu1 | 700000 | $m^2/s$ | Viscocity parameter for u1 in y |

| Field | Value | Units | Description |
|---|---|---|---|
| weml | 0.01 | m/s | Mixed layer entrainment velocity |
| year0 | 0 | yr | Starting year; if $< 0$ use year in restart |
| ziml | 500 | m | Atmosphere mixed layer depth $\sim$ cloud base |

## A.2 Array Field Variables

This table lists the default settings for array `qtcm` fields as set by the `defaults` submodule. All fields are of class `Field`. Numerical values are rounded as per the conventions of Python's `%g` format code.

| Field | Shape | Max | Min | Units | Description |
|---|---|---|---|---|---|
| Evap | (1, 1) | 0 | 0 | | |
| FLW | (1, 1) | 0 | 0 | | |
| FLWds | (1, 1) | 0 | 0 | | |
| FLWus | (1, 1) | 0 | 0 | | |
| FLWut | (1, 1) | 0 | 0 | | |
| FSW | (1, 1) | 0 | 0 | | |
| FSWds | (1, 1) | 0 | 0 | | |
| FSWus | (1, 1) | 0 | 0 | | |
| FSWut | (1, 1) | 0 | 0 | | |
| FTs | (1, 1) | 0 | 0 | | |
| Qc | (1, 1) | 0 | 0 | K | Precipitation |
| S0 | (1, 1) | 0 | 0 | | |
| STYPE | (1, 1) | 0 | 0 | | Surface type; ocean or vegetation type over land |
| T1 | (1, 1) | 0 | 0 | K | |
| Ts | (1, 1) | 0 | 0 | K | Surface temperature |
| WD | (1, 1) | 0 | 0 | | |
| WD0 | (1,) | 0 | 0 | | Field capacity SIB2/CSU (approximately) |
| arr1 | (1, 1) | 0 | 0 | | Auxiliary optional output array 1 |
| arr2 | (1, 1) | 0 | 0 | | Auxiliary optional output array 2 |
| arr3 | (1, 1) | 0 | 0 | | Auxiliary optional output array 3 |
| arr4 | (1, 1) | 0 | 0 | | Auxiliary optional output array 4 |

| **Field** | **Shape** | **Max** | **Min** | **Units** | **Description** |
|---|---|---|---|---|---|
| arr5 | (1, 1) | 0 | 0 | | Auxiliary optional output array 5 |
| arr6 | (1, 1) | 0 | 0 | | Auxiliary optional output array 6 |
| arr7 | (1, 1) | 0 | 0 | | Auxiliary optional output array 7 |
| arr8 | (1, 1) | 0 | 0 | | Auxiliary optional output array 8 |
| psi0 | (1, 1) | 0 | 0 | | |
| q1 | (1, 1) | 0 | 0 | K | |
| rhsu0bar | (1,) | 0 | 0 | | |
| rhsvort0 | (1, 1, 1) | 0 | 0 | | |
| taux | (1, 1) | 0 | 0 | | |
| tauy | (1, 1) | 0 | 0 | | |
| u0 | (1, 1) | 0 | 0 | m/s | Barotropic zonal wind |
| u1 | (1, 1) | 0 | 0 | m/s | Current time step baroclinic zonal wind |
| v0 | (1, 1) | 0 | 0 | m/s | Barotropic meridional wind |
| v1 | (1, 1) | 0 | 0 | m/s | |
| vort0 | (1, 1) | 0 | 0 | | |